Machine Learning Notes

Lessons Learned

Ladislav Šulák <laco.sulak@gmail.com>

April 28, 2021

Contents

1 General

1.1 The Field of AI, CI, and Machine Learning

Intelligence

- Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.
- Webster's New Collegiate Dictionary defines intelligence as:
  - The ability to learn or understand or to deal with new or trying situations - reason. Also, the skilled use of reason.
  - The ability to apply knowledge to manipulate one's environment or to think abstractly as measured by objective criteria (as tests).
- Davig Fogel's definition:
  - The capability of a system to adapt its behavior (to implement decisions) to meet its goals in a range of environments. It is a property of all purpose-driven decision makers.

Artificial Intelligence

- John McCarthy (a co-founder of the field)[1]
  - It is the science and engineering of **making intelligent machines**, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable. (however it can be also a hundreds of thousand lines of conditions IF)
- TL;DR; **AI is intelligence demonstrated by machines**[2].
- AI is defined as the study of the computation required for intelligent behavior (mostly humans, but also animals and so on) and the attempt to duplicate such computation using computers.
- AI is a superset of CI and ML. AI is a very broad term, and some subsets (applications) can be interrelated together, for example ANN are CI and also ML. In literature, various definitions and terms in the field of AI are still not used homogeneously[3].
- Branches of AI (not full list, because some were not yet identified): logical AI, search, **pattern recognition**, representation, inference, common sense knowledge and reasoning, learning from experience, planning, epistemology, ontology, heuristics, genetic programming.
- Applications of AI (some of them): game playing, speech recognition, understanding natural language, computer vision, expert systems, heuristic classification.
- Anything typical human can do in less than 1 second of thought, we can possibly now or soon automate with AI[4].
- AI can be divided into two branches:
  - **ANI** (Artificial Narrow Intelligence) - these have one purpose, such as smart reader, self-driving car, web search, and so on.
  - **AGI** (Artificial General Intelligence) - this concept is about building AI that can do whatever a human can do, or maybe even more things.

[1] http://www-formal.stanford.edu/jmc/whatisai/whatisai.html
[2] https://www.iep.utm.edu/art-inte/
[3] https://blog.leanix.net/en/what-is-the-difference-between-artificial-intelligence-and-machine-learning
[4] https://www.youtube.com/watch?v=21EiKfQYZXc

Machine Learning

- Arthur Samuel (1959).
  - Field of study that gives computer the ability to learn without being explicitly programmed.
- Tom Mitchell (1998).
  - A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its **performance** on T, as measured by P, **improves with experience** E.

    Example: playing checkers.
    - E = the experience of playing many games of checkers
    - T = the task of playing checkers.
    - P = the probability that the program will win the next game.
- **Evolved from** the study of **pattern recognition** and **computational learning theory** in **AI**.
- **Data-driven predictions or decisions** = building a model from sample input. Machine learning systems automatically learn programs from data.
- A dumb algorithm with lots and lots of data beats a clever one with modest amounts of it. After all, machine learning is all about letting data do the heavy lifting.
- **Machine learning tasks by learning style**
  - Supervised learning - in every example of dataset we have a correct answer about the example. In other words, we are given a dataset and already know what our correct output should look like, having the idea that there is a relationship between the input and the output. These problems can be generalized into:
    - Semi-supervised learning: the computer is given only an incomplete training signal: a training set with some (often most) of the target outputs are missing. The goal is the same as supervised learning algorithm. By adding more examples you add more information about your problem - larger sample reflects better the probability distribution the data we labeled came from.
    - Active learning: special case of semi-supervised learning. Instead of assuming that all of the training examples are given at the start, active learning algorithms interactively collect new examples, typically by making queries to a human user (or some other information source). Often, the queries are based on unlabeled data, which is a scenario that combines semi-supervised learning with active learning.

The computer can only obtain training labels for a limited set of instances (based on a budget), and also has to optimize its choice of objects to acquire labels for. When used interactively, these can be presented to the user for labeling.

  - Structured prediction: when the desired output value is a complex object, such as a parse tree or a labeled graph, then standard methods must be extended. It is an umbrella term for supervised machine learning techniques that involves predicting structured objects, rather than scalar discrete or real values.
  - Learning to rank: when the input is a set of objects and the desired output is a ranking of those objects, then again the standard methods must be extended. It is a supervised learning problem.

- Unsupervised learning - find a structure from data itself where we do not know the effect of the variables. In other words, this allows us to approach problems with little or no idea what our results should look like. No labels are given to the learning algorithm. For example clustering algorithms deals with this. Or dimensionality reduction algorithms, or outlier detection (here, the output is a real number that indicates how x is different from a "typical" example in the dataset).
- Reinforcement learning[5] - training data (in form of rewards and punishments) is given only as a feedback to the program's actions in a dynamic environment, such as playing a game against an opponent. The machine "lives" in an environment and is capable of perceiving the state of that environment as a vector of features. The machine can execute actions in every state, and different actions bring different rewards and could also move the machine to another state of the environment. The goal of reinforcement learning algorithm is to learn a policy (this is a function - similar to a model in supervised learning - that takes the feature vector of a state as input and outputs an optimal action to execute in that state). The action is optimal if it maximizes the **expected average reward**. Reinforcement learning solves a particular kind of problem where decision making is sequential and the goal is long-term.
  - Here belong algorithm such as Monte Carlo, Q-learning, or SARSA[6].

- Another way of **categorizing machine learning tasks** is when **we consider** the **similarities in their function**. Chapters in this document basically group machine learning algorithms based on how they work. However it is not perfect either because some algorithms can be in multiple categories. This was inspired from machinelearningmastery[7] and Scikit-Learn API reference[8].

[5] https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/
[6] https://en.wikipedia.org/wiki/Reinforcement_learning
[7] https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/
[8] http://scikit-learn.org/stable/modules/classes.html#api-reference

Computational Intelligence

- Two key concepts and requirements - **adaptation and self-organization**. Algorithms and implementations that enable or facilitate appropriate **actions** (intelligent behavior) **in complex and changing environments**.
- CI usually refers to the ability of a computer to learn a specific task from data or experimental observation. Even though it is commonly considered a synonym of **soft computing**, there is still no commonly accepted definition of computational intelligence[9].
- Generally, CI is a set of nature-inspired computational methodologies and approaches to address complex real-world problems.
- Although Artificial Intelligence and Computational Intelligence seek a similar long-term goal: reach general intelligence, which is the intelligence of a machine that could perform any intellectual task that a human being can; there's a clear difference between them. According to Bezdek (1994), **Computational Intelligence is a subset of Artificial Intelligence**.
- Main principles:
  - Fuzzy logic (uncertainty like in human experiences - thus reflected as the behavioral level of the organism; fuzziness is not resolved by observation or measurement
  - Neural Networks (so they belong to CI and ML)
  - Evolutionary computation (includes genetic algorithms, genetic programming, evolutionary programming, evolution strategies)
  - Learning theory
  - Probabilistic methods

Soft Computing

- Same as CI, expanded with Probabilistic Reasoning, Swarm Intelligence, and partly Chaos Theory.
- Soft Computing is thus tolerant of imprecision, uncertainty and partial truth.

[9] https://en.wikipedia.org/wiki/Computational_intelligence

Knowledge Discovery in Databases

- KDD is a process of discovering interesting patterns and knowledge from large amounts of data.
- KDD involves the following process stages (iterative steps):[10]
  1. Development and understanding of the application domain.
  2. Creating a target data set.
  3. Data cleaning and preprocessing.
  4. Data reduction and projection.
  5. Matching process objectives (e.g. summarization, classification, regression, clustering).
  6. Modeling and exploratory analysis and hypothesis selection: choosing the algorithms or data mining, and select the method or methods to be used in the search for patterns of data.
  7. Data Mining: the search for patterns of interest in a particular representational form or a set of these representations, including classification rules or trees, regression and clustering. It can be descriptive (=unsupervised learning) or predictive (=supervised learning).
  8. Pattern evaluation - interpreting mined patterns.
  9. Acting on the discovered knowledge: using the knowledge directly, incorporating the knowledge in another system for further action, or simply document the results. It is knowledge presentation - post-processing in a way that is easily understandable by users, usually using visualizations.

Data Mining

- Process of discovering patterns in large datasets involving methods at the **intersection of machine learning, statistics,** and **database systems**.
- Data mining is just one process (raw analysis) of more ones in the overall **KDD** (knowledge discovery in databases) - however, a lot of people thinks that Data Mining = KDD.

Data Analysis Process

- Process of ordering and organizing **raw** data in order to determine useful insights and decisions.

[10] https://www.smartdatacollective.com/difference-between-knowledge-discovery-and-data-mining/

- This is a superset of Data Mining that involves extracting, cleaning, transforming, modeling and visualization of data with an intention to uncover meaningful and useful information that can help in deriving conclusion and take decisions.[11] It requires the intersection of **computer science, machine learning, statistics**, and **mathematics**.
- The process has 5 steps:
  1. Define Your Questions
  2. Set Clear Measurement Priorities (what and how to measure)
  3. Collect Data
  4. Analyze Data
  5. Interpret Results

[11] https://www.educba.com/data-mining-vs-data-analysis/

Survey of major AI application areas

- **Computer Vision**
  - **Image classification / Object recognition** (here belongs **Face recognition** for instance) - yes or no answers, if 2 images shows the same person for example.
  - **Object detection** - positions of found objects. For example position (in rectangle box) of a pedestrian or a car.
  - **Image segmentation** - each pixel is classified into a class. It tells us, if a given pixel is a part of a car or pedestrian and so on. There is an excellent dataset named COCO (from 2018, but may be updated) here[12].
  - **Tracking** - also tracks for example people in a video. Basically a object recognition on a video.
- **Natural Language Processing**
  - **Text classification** - given an input email, is it spam or not? Or for example, classify from a given product description, which product category does it belongs to. Here belongs also **Sentiment classification** (automatically rate a review of some product).
  - **Information retrieval** - e.g. web search.
  - **Name entity recognition** - find people's name in a sentences, automatic extraction of names of companies, phones numbers, and so on.
  - **Machine translation** - translation of text from one language into another language.
  - **Part-of-speech** - given an input sentence, which words are verbs, which are nouns, and so on. This can be important for building sentiment classification system. We can thus help AI system to figure out which of the words to pay more attention to.
  - **Parsing** - helping a group the words together to make a phrase.
- **Speech**
  - **Speech recognition** - is speech-to-text translation.
  - **Trigger word/wakeword detection**
  - **Speaker ID** - get a identity of a speaker.
  - **Speech synthesis** - speech-to-text, also known as TTS.
- **Robotics**
  - **Perception** - figure out what's in he world around you based on the senses you have (cameras, radar, sensors, ...).

[12]COCO is a large-scale object detection, segmentation, and captioning dataset.

  - **Motion planning** - finding a path for the robot to follow.
  - **Control** - sending commands to motors to follow a path.

- **General machine learning**
  - Unstructured data - images, text, audio, ...
  - Structured data - tables, structures, ...

ML-related Job Positions

- **Software Engineer**
  - Writes specialized software for executing a functionality based/with some learned model; making software reliable.

- **Machine Learning Engineer**
  - Responsible for learning ML models or for building other ML algorithms.
  - Machine Learning Engineers often focus more on producing great data-driven products than they do answering operational questions for a company.

- **Machine Learning Researcher**
  - Extending state-of-the-art in ML. These people may publish papers (for some companies).

- **Applied ML scientist**
  - Stands between ML Researcher and ML Engineer. They are responsible for going to the academic literature or the research literature and finding the state-of-the-art techniques and finding ways to adapt them to a problem they are facing.

- **Data Engineer**
  - Organize data and making sure that the data is saved in an easily, accessible, secure, and cost effective way.
  - Since you'd be (one of) the first data hires, heavy statistics and machine learning expertise is less important than strong software engineering skills. As a result, you'll have great opportunities to shine and grow via trial by fire, but there will be less guidance and you may face a greater risk of flopping or stagnating.
  - Data Engineers are the link between the management's big data strategy and the data scientists that need to work with data.
  - This can be basically (it can be said, that BA is a "beginning position" and DM is "more skilled")

    * **Business Analyst** - using business intelligence to translate the analyzed data into visual insights. It is also needed to know data mining and AB testing. Average salary in 2018 is 67.350 dollars/year.
    * **Data Analyst** - data preparation and cleaning, entry-level position in data science. More math, software tools are used in this role. Average salary in 2018 is 73.250 dollars/year.
    * **Data Modeler** - developing systems and models that can be applied to a company's databases. This role can be similar to ML Engineer. Average salary in 2018 is 81.850 dollars/year. A more advanced "version" of this role, that is more about research, is **Data Scientist**.

- **Data Scientist**
  - Examining data and provide insights. Make presentations to team/executive. However, this position can be very similar to ML Engineer.
  - Data scientist is often used as a blanket title to describe jobs that are drastically different! Data science combines several disciplines, including statistics, data analysis, machine learning, and computer science.
  - This can be the same name as **Data Analyst** role (plus machine learning, but usually more skilled). But, if you are data analyst, your job might consist of tasks like pulling data out of SQL databases, becoming an Excel or Tableau master, and producing basic data visualizations and reporting dashboards.
  - Once you have a handle on your day-to-day responsibilities, a company like this can be a great environment to try new things and expand your skillset.
  - They are math experts. They use linear algebra and multi-variable calculus to create new insight from existing data.
  - Average salary in 2018 is 121.350 dollars/year.
- **AI Product Manager**
  - Decide what to build, what is feasible and valuable.
  - Similar role is **Data Science Manager**, but these are both less technical roles. They involve presentation of results and communication with internal organs or other managers / departments. Average salary in 2018 is 134.850 dollars/year.

Data Science Project Workflow

1. Identify the question (Data Scientist is responsible for this part, but Data Scientist role is responsible for all the roles)
2. Prepare the data (Data Analyst or Data Modeler)
3. Analyze the data (Data Modeler)

4. Visualize the insights (Business Analyst)
5. Present your findings (Business Analyst or Data Science Manager)

Big Data

The term "big data" does not refer only to a big volume. There are 4 important things: Volume, velocity, variety, and veracity.

- **Volume** (size) how much data you have.
- **Velocity** (speed) - how fast data are getting to you. How much needs to be processed or is coming into the system. This is where the whole concept of streaming data and real-time processing comes in to play.
- **Variety** - how different your data are.
- **Veracity** (credibility) - how reliable are your data. The issue with big data is that they are very unreliable. You cannot really trust the data.

Use Big Data only if you need to! That means, only if you are running into scaling issues! Big Data is a very expensive thing. For example, a Hadoop cluster for instance needs at least five servers to work properly (more is better). Also, maintenance and development of top big data tools can be expensive.

- Batch processing
- Stream processing
  - In stream processing sometimes it is absolutely all right to drop messages, other times it is not. Sometimes it is fine to process a message multiple times, other times that needs to be avoided.
  - That is why there are different strategies of streaming: processing message at least once, exactly once, or at most once.

Hadoop Platforms

- When people talk about big data, one of the first things come to mind is Hadoop.
- You will see that Hadoop has evolved from a platform into an ecosystem. Its design allows a lot of Apache projects and 3rd party tools to benefit from Hadoop.
- Hadoop is a platform for distributed storing and analyzing of very large data sets. Hadoop has four main modules: Hadoop common, HDFS, MapReduce and YARN. The way these modules are woven together is what makes Hadoop so successful.
- Note: Hadoop is still relevant in 2019 even if you look into serverless tools.

- You use Apache Kafka to ingest data, and store the it in HDFS. You do the analytics with Apache Spark and as a backend for the display you store data in Apache HBase. To have a working system you also need YARN for resource management. You also need Zookeeper, a configuration management service to use Kafka and HBase. Each project is closely connected to the other. Spark for instance, can directly access Kafka to consume messages. It is able to access HDFS for storing or processing stored data. Want to store data from Kafka directly into HDFS without using Spark? No problem, there is a project for that. Apache Flume has interfaces for Kafka and HDFS.
- **Alternatives**
  - Often times it does not make sense to deploy a Hadoop cluster, because it can be overkill. Hadoop does not run on a single server.
  - You basically need at least five servers, better six to run a small cluster. Because of that. the initial platform costs are quite high.
  - One option you have is to use a specialized systems like Cassandra, MongoDB or other NoSQL DB's for storage. Or you move to Amazon and use Amazon's Simple Storage Service, or S3.
  - Guess what the tech behind S3 is. Yes, HDFS. That's why AWS also has the equivalent to MapReduce named Elastic MapReduce. The great thing about S3 is that you can start very small. When your system grows you don't have to worry about S3's server scaling.
- **HDFS document store**
  - HDFS works different to a typical file system - HDFS is hardware independent. Not only does it span over many disks in a server. It also spans over many servers.
  - HDFS will automatically place your files somewhere in the Hadoop server collective.
  - It will not only store your file, Hadoop will also replicate it two or three times (you can define that). Replication means replicas of the file will be distributed to different servers.
  - This gives you superior fault tolerance. If one server goes down, then your data stays available on a different server.
  - Another great thing about HDFS is, that there is no limit how big the files can be.
  - HDFS physically stores files different then a normal file system. It splits the file into blocks. These blocks are then distributed and replicated on the Hadoop cluster. The splitting happens automatically. In the configuration you can define how big the blocks should be (megabytes or gigabyte, it's up to you).

- **MapReduce**
  - How MapReduce is working is, that it processes data in 2 phases: the map phase and then the reduce phase.
  - The map phase - the framework is reading data from HDFS. Each dataset is called an input record. The whole map and reduce process relies heavily on using key-value pairs. That's what the mappers are for. In the map phase input data, for instance a file, gets loaded and transformed into key-value pairs. When each map phase is done it sends the created key-value pairs to the reducers where they are getting sorted by key. This means, that an input record for the reduce phase is a list of values from the mappers that all have the same key.
  - The reduce phase - actual computation is done and the results are stored. The storage target can either be a database or back HDFS or something else. Reduce phase is doing the computation of key and its values from map phase and outputs the results.
  - The map and reduce phases are parallelised. What that means is, that you have multiple map phases (mappers) and reduce phases (reducers) that can run in parallel on your cluster machines.
  - How many mappers and reducers can you use in parallel? The number of parallel map and reduce processes depends on how many CPU cores you have in your cluster. Every mapper and every reducer is using one core.
  - MapReduce is awesome for simpler analytics tasks, like counting stuff. It just has one flaw: It has only two stages Map and Reduce.
    - The problem with MapReduce is that there is no simple way to chain multiple map and reduce processes together.
    - At the end of each reduce process the data must be stored somewhere. This fact makes it very hard to do complicated analytics processes. You would need to chain MapReduce jobs together. Chaining jobs with storing and loading intermediate results just makes no sense.
  - Another issue with MapReduce is that it is not capable of streaming analytics. Jobs take some time to spin up, do the analytics and shut down. Basically minutes of wait time are totally normal. This is a big negative point in a more and more real time data processing world.

Apache Spark

- Spark is a complete in-memory framework. Data gets loaded from, for instance HDFS, into the memory of workers.
- There is no longer a fixed map and reduce stage as it is in MapReduce. Your code can be as complex as you want.

- Once in memory, the input data and the intermediate results stay in memory (until the job finishes). They do not get written to a drive like with MapReduce. This makes Spark the optimal choice for doing complex analytics. It allows you for instance to do iterative processes. Modifying a dataset multiple times in order to create an output is totally easy.
- Streaming analytics capability is also what makes Spark so great. Spark has natively the option to schedule a job to run every X seconds or X milliseconds. As a result, Spark can deliver you results from streaming data in "real time".
- Spark vs Hadoop is however a wrong question.
  - Hadoop is used to store data in the Hadoop Distributed File System (HDFS). It can analyze the stored data with MapReduce and manage resources with YARN. However, Hadoop is more than just storage, analytics and resource management. There's a whole ecosystem of tools around the Hadoop core.
  - Spark is "just" an analytics framework. It has no storage capability. Although it has a standalone resource management, you usually don't use that feature.
  - So Spark and Hadoop are not the same thing, and they can even work together. As Storage you use HDFS. Analytics is done with Apache Spark and YARN is taking care of the resource management.
  - From a platform architecture perspective, Hadoop and Spark are usually managed on the same cluster. This means on each server where a HDFS data node is running, a Spark worker thread runs as well.
  - In distributed processing, network transfer between machines is a large bottle neck. Transferring data within a machine reduces this traffic significantly. Spark is able to determine on which data node the needed data is stored. This allows a direct load of the data from the local storage into the memory of the machine, which significantly reduces network traffic.
  - So, the question is not whether to use Spark or Hadoop. The question has to be: Should you use Spark or MapReduce alongside Hadoop's HDFS and YARN.
  - If you are doing simple batch jobs like counting values or doing calculating averages, go with MapReduce. If you need more complex analytics like machine learning or fast stream processing: go with Apache Spark.
- Spark jobs can be programmed in a variety of languages. That makes creating analytic processes very user-friendly for data scientists.
- Data locality - you can and you should run Spark workers directly on the data nodes of your Hadoop cluster, because processing data locally where it is stored is the most efficient thing to do. Spark can then natively identify on what data node the needed data is stored. This enables Spark to use the worker running on the machine where the data is stored to load the data into the memory. The downside

of this setup is that you need more expensive servers. Because Spark processing needs stronger servers with more RAM and CPUs than a "pure" Hadoop setup.

- The machine learning library MLlib is included in Spark so there is often no need to import another library. But Spark can be integrated with TensorFlow for example.
- Managing resources - Spark has stand-alone resource manager. If Spark is running in an Hadoop environment you don't have to use Spark's own standalone resource manager. You can configure Spark to use Hadoop's YARN resource management. Having a single resource manager instead of two independent ones makes it a lot easier to configure the resource management.
- **Apache Nifi** - tool for creating data pipelines; you can do it directly in UI. You can read data from a RestAPI and post it to Kafka, or read data from Kafka and put it into a database.

Machine Learning in Production

- Machine learning in production can be done by using stream and batch processing. In the batch processing layer you are creating the models, because you have all the data available for training. In the stream in processing layer you are using created models and you are applying them to new data.
- This is a constant cycle: training, applying, re-training, pushing into production, and applying. What you don't want to do is doing this manually. You need to figure out a process of automatic retraining and automatic pushing to into production of models.
- Automation is harder than you think! Look at AWS machine learning for instance. The process is: build, train, tune deploy. Where's the loop of retraining? You can create models and then use them in production. But this loop is almost nowhere to be seen. In 2019, this is still a very big issue that needs to be solved.

Data Warehouse

Data Lake

1.2 Data Types

(from here[13])

- Qualitative (broader name for variables that can't be counted, output of classification models)
  - Nominal (a.k.a. categorical) are like labels. They are mutually exclusive, so no overlap. Memohelp - nominal sounds like name. There is no order, for instance MALE and FEMALE. They are discrete.
  - Ordinal in which the order matters, we can assign scores. Examples are BAD, OK and HAPPY.
  - Dichotomous (a.k.a. binary variable), that can only have 2 values.
- Quantitative (broader name for variables that can be counted, output of regression models)
  - Discrete can be mapped to limited int, which means that there can be only limited amount of possible values.
  - Continuous have unlimited number of values.
- Different angle, from experimentation with model
  - Confounding variable: extra variables that have a hidden effect on your experimental results.
  - Dependent variable: the outcome of an experiment. As you change the independent variable, you watch what happens to the dependent variable.
  - Independent variable: (also known as predictor variable) a variable that is not affected by anything that you, the researcher, does. Usually plotted on the x-axis. It is usually some input, feature, but it is also known as explanatory variable, regressor, and some others[14].
  - Endogenous variable: similar to dependent variables, they are affected by other variables in the system. Used almost exclusively in econometrics.
  - Exogenous variable: variables that affect others in the system.
  - Extraneous variables are any variables that you are not intentionally studying in your experiment or test.
  - Latent variable: a hidden variable that can't be measured or observed directly.

[13] http://www.statisticshowto.com/types-variables/
[14] https://en.wikipedia.org/wiki/Dependent_and_independent_variables

1.3 Data Distributions

(from here[15] and here[16])

- Bernoulli distribution - it has only two possible outcomes, namely 1 (success) and 0 (failure), and a single trial.

$$P(x) = \begin{cases} 1-p & x = 0 \\ p & x = 1 \end{cases} \tag{1.1}$$

  for $0 < p < 1$

  - A Bernoulli distribution is a special case of Binomial distribution (see below). Specifically, when $n = 1$ the **Binomial distribution** becomes Bernoulli distribution. Also, a generalization of Bernoulli distribution is called **Categorical distribution** (probability distribution for a set of discrete random variables).
  - The Bernoulli distribution serves as a building block for discrete distributions which model Bernoulli trials, such as **binomial distribution** and **geometric distribution**.
  - It is a probability distribution over a two-valued (=binary - true or false / 1 or 0) random variable.
  - **A Bernoulli trial, or Bernoulli experiment**, is an experiment satisfying two key properties:
    - There are exactly two complementary outcomes, success and failure.
    - The probability of success is the same every time the experiment is repeated.
  - **Example**: For a single coin toss, the probability you win one dollar is p. The random variable that represents your winnings after one coin toss is a Bernoulli random variable.
  - Another examples:[17]
    - A tennis player either wins or loses a match.
    - A newborn child is either male or female.
    - You either pass or fail an exam.

[15] https://www.analyticsvidhya.com/blog/2017/09/6-probability-distributions-data-science/
[16] https://medium.com/@laumannfelix/statistics-probability-fundamentals-2-cbb1239f9605
[17] https://brilliant.org/wiki/bernoulli-distribution/

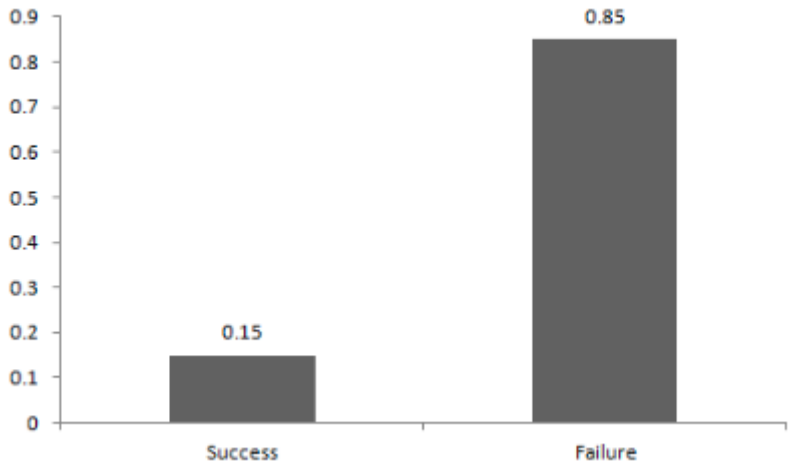


Figure 1.1: An example of Bernoulli Distribution of data. X-axis has 2 possible scenarios and y-axis is a probability.

- Binomial distribution - a distribution where only two outcomes are possible, such as success or failure, gain or loss, win or lose and where the **probability of success and failure is same for all the trials** is called a Binomial distribution.
  - If there are n Bernoulli trials, and each trial has a probability p of success, then the probability of exactly k successes is:

$$Pr(X = k) = \begin{pmatrix} k \\ n \end{pmatrix} p^k (1-p)^{n-k} \tag{1.2}$$

  denoting the probability that the random variable X is equal to k
  - In other words, an **experiment** with **only two possible outcomes repeated n number of times** is called **binomial**. The parameters of a binomial distribution are n and p where n is the total number of trials and p is the probability of success in each trial.
  - It is useful for analyzing the results of repeated independent trials.
  - A binomial experiment is a series of n Bernoulli trials, whose outcomes are independent of each other. A random variable, X, is defined as the number of successes in a binomial experiment. Finally, a binomial distribution is the probability distribution of X.
  - **Example**: if you toss the coin 5 times, your winnings could be any whole number of dollars from zero dollars to five dollars, inclusive. The probability that you win five dollars is p^5, because each coin toss is independent of the others, and for each coin toss the probability of heads is p. Probability of winning exactly three dollars in five tosses would require you to toss the coin five times, getting exactly three heads and two tails. This can be achieved with probability $\binom{5}{3} * p^3 * (1-p)^2$. And, in general, if there are n Bernoulli trials, then the sum of those trials is binomially distributed with parameters n and p.
  - Another example:[18]
    - Consider a fair coin. Flipping the coin once is a Bernoulli trial, since there are exactly two complementary outcomes (flipping a head and flipping a tail), and they are both $\frac{1}{2}$ no matter how many times the coin is flipped. Note that the fact that the coin is fair is not necessary; flipping a weighted coin is still a Bernoulli trial.
    - A binomial experiment might consist of flipping the coin 100 times, with the resulting number of heads being represented by the random variable X. The binomial distribution of this experiment is the probability distribution of X.
  - **Properties**:
    - Each trial (/attempt) is independent.

[18] https://brilliant.org/wiki/binomial-distribution/

  - There are only two possible outcomes in a trial - either a success or a failure.
  - A total number of n identical trials are conducted.
  - The probability of success and failure is same for all trials (trials are identical).

$$P(x) = \frac{n!}{(n-x)!x!} p^x * (1-p)^{n-x} \tag{1.3}$$

- The difference between Binomial and Bernoulli distributions are[19]:
  - A Bernoulli random variable has two possible outcomes: 0 or 1.
  - A Binomial distribution is the **sum of independent and identically distributed** Bernoulli random variables.
  - All Bernoulli distributions are Binomial distributions, but most Binomial distributions are not Bernoulli distributions.

[19] https://math.stackexchange.com/questions/838107/what-is-the-difference-and-relationship-between-the-binomial-and-bernoulli-distr

- Uniform distribution - all the n number of possible outcomes of a uniform distribution are equally probable, always defined with boundaries a and b.
  - A variable x is said to be uniformly distributed if the **the probability density function** is:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & x < a \; or \; x > b \end{cases} \tag{1.4}$$

And graph of a uniform distribution curve looks like:

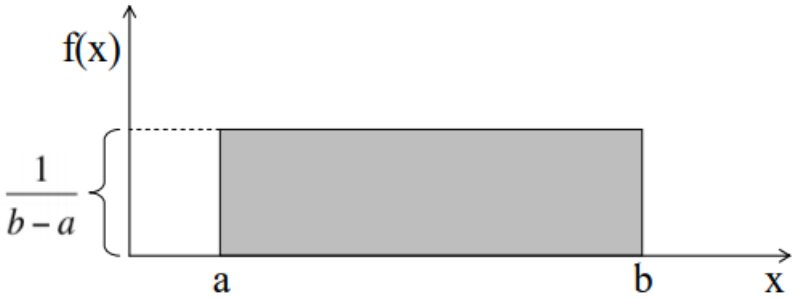


Figure 1.2: An example of Uniform distribution of data.

- Geometric distribution
  - The geometric distribution is useful for determining the likelihood of a success given a limited number of trials, which is highly applicable to the real world in which unlimited (and unrestricted) trials are rare.
  - For a geometric distribution with probability p of success, the probability that exactly k failures occur before the first success is:

$$Pr(X = k) = (1 - p)^k p \tag{1.5}$$

  - Note that the geometric distribution satisfies the important property of being memory-less, meaning that if a success has not yet occurred at some given point, the probability distribution of the number of additional failures does not depend on the number of failures already observed. For instance, suppose a die is being rolled until a 1 is observed. If the additional information were provided that the die had already been rolled three times without a 1 being observed, the probability distribution of the number of further rolls is the same as it would be without the additional information.[20]
  - An example: A programmer has a 90% chance of finding a bug every time he compiles his code, and it takes him two hours to rewrite his code every time he discovers a bug. What is the probability that he will finish his program by the end of his workday?
    - Assume that a workday is 8 hours and that the programmer compiles his code immediately at the beginning of the day.
    - In this instance, a success is a bug-free compilation, and a failure is the discovery of a bug. The programmer needs to have 0, 1, 2, or 3 failures, so his probability of finishing his program is:

      $Pr(X = 0) + Pr(X = 1) + Pr(X = 2) + Pr(X = 3) = (0.9)^0(0.1) + (0.9)^1(0.1) + (0.9)^2(0.1) + (0.9)^3(0.1) = 0.344$
    - This information is useful for determining whether the programmer should spend his day writing the program or performing some other tasks during that time.

[20] https://brilliant.org/wiki/geometric-distribution

- Normal (Gaussian) distribution - characterized by standard deviation and mean. This distribution represents the behavior of most of the situations in the universe.
  - **Properties**:
    - The mean, median and mode of the distribution coincide.
    - The curve of the distribution is bell-shaped and symmetrical about the line $x = \mu$.
    - The total area under the curve is 1.
    - Exactly half of the values are to the left of the center and the other half to the right.
  - The normal distribution is highly different from Binomial distribution. However, if the number of trials $n \to \infty$, then the shapes will be quite similar.
  - The normal distribution is also a limiting case of Poisson distribution with the parameter $\lambda \to \infty$.
  - It can be computed in the following way (so there are just 2 parameters: μ and σ):

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.6}$$

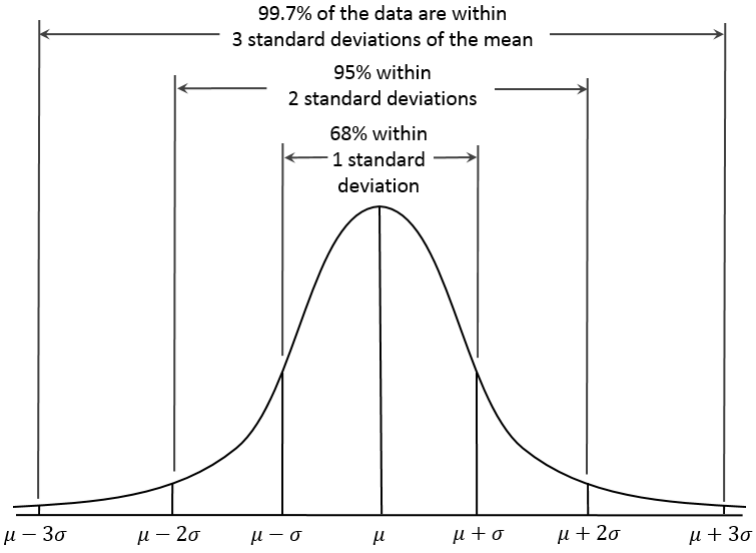


Figure 1.3: An example of 3 Normal distributions of data. X-axis has measured variable, and y-axis has density.

  - Owing largely to the central limit theorem, the normal distributions is an appropriate approximation even when the underlying distribution is known

to be not normal. This is convenient because the normal distribution is easy to obtain estimates with; the empirical rule states that 68% of the data modeled by a normal distribution falls within 1 standard deviation of the mean, 95% within 2 standard deviations, and 99.7% within 3 standard deviations. For obvious reasons, the empirical rule is also occasionally known as the 68-95-99.7 rule.[21]

[21] https://brilliant.org/wiki/normal-distribution

- Poisson distribution - used as an approximation to binomial with $p < 0,1$ and $n > 30$. Normal distribution is used as an approximation to binomial with sufficiently large p.
  - This is a discrete probability distribution of the number of events occurring in a given time period, given the average number of times the event occurs over that time period.
  - The Poisson distribution is applicable only when several conditions hold:
    * An event can occur any number of times during a time period.
    * Events occur independently. In other words, if an event occurs, it does not affect the probability of another event occurring in the same time period.
    * The rate of occurrence is constant; that is, the rate does not change based on time.
    * The probability of an event occurring is proportional to the length of the time period. For example, it should be twice as likely for an event to occur in a 2 hour time period than it is for an event to occur in a 1 hour period.
  - Given that a situation follows a Poisson distribution, there is a formula which allows one to calculate the probability of observing k events over a time period for any non-negative integer value of k. Let X be the discrete random variable that represents the number of events observed over a given time period. Let λ be the expected value (average) of X. If X follows a Poisson distribution, then the probability of observing k events over the time period is:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{1.7}$$

    where e is Euler's number
  - **Example**: suppose you work at a call center, approximately how many calls do you get in a day? It can be any number. Now, the entire number of calls at a call center in a day is modeled by Poisson distribution.
  - Another example: The Poisson distribution is appropriate for modeling the number of phone calls an office would receive during the noon hour, if they know that they average 4 calls per hour during that time period.
    * Although the average is 4 calls, they could theoretically get any number of calls during that time period.
    * The events are effectively independent since there is no reason to expect a caller to affect the chances of another person calling.
    * The occurrence rate may be assumed to be constant.

  - ∗ It is reasonable to assume that (for example) the probability of getting a call in the first half hour is the same as the probability of getting a call in the final half hour.

– Another example: In the World Cup, an average of 2.5 (this is our expected value λ) goals are scored each game. Modeling this situation with a Poisson distribution, what is the probability that k goals are scored in a game?

$P(X = 0) = \frac{2.5^0 e^{-2.5}}{0!} \approx 0.082$

$P(X = 1) = \frac{2.5^1 e^{-2.5}}{1!} \approx 0.205$

$P(X = 2) = \frac{2.2 e^{-2.5}}{2!} \approx 0.257$

$P(X = 3) = \frac{2.5^3 e^{-2.5}}{3!} \approx 0.213$

$P(X = 4) = \frac{2.5^4 e^{-2.5}}{4!} \approx 0.133$

And so on. There is no upper limit on the value of k for this formula, though the probability rapidly approaches 0 as k increases (so more far away we are from expected value, then the lower probability we receive).

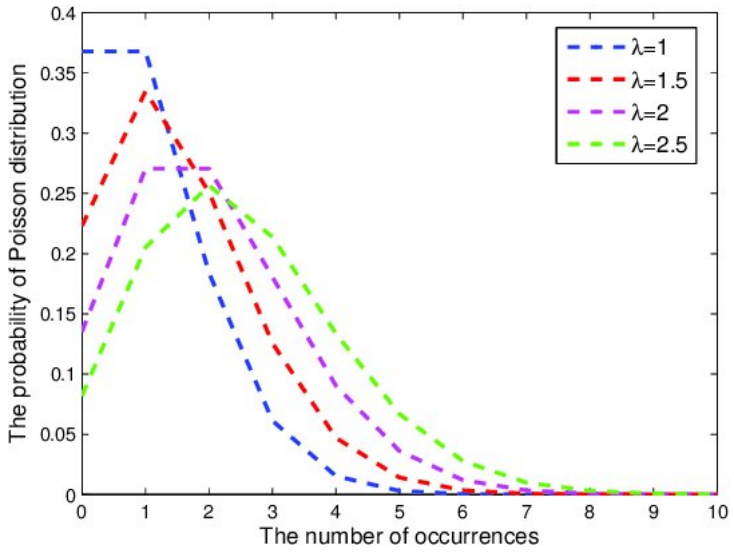


Figure 1.4: An example of Poisson distribution of data.

- Exponential distribution - probability distribution that describes the time between events in a Poisson point process, i.e. a process in which events occur continuously and independently at a constant average rate.
  - The exponential distribution is a continuous probability distribution which describes the amount of time it takes to obtain a success in a series of continuously occurring independent trials. It is a continuous analog of the geometric distribution.
  - **Example**: interval of time between the calls of call center.
  - **The probability density function**:

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1.8}$$

where $\lambda > 0$ is the parameter of the distribution, often called the rate parameter.

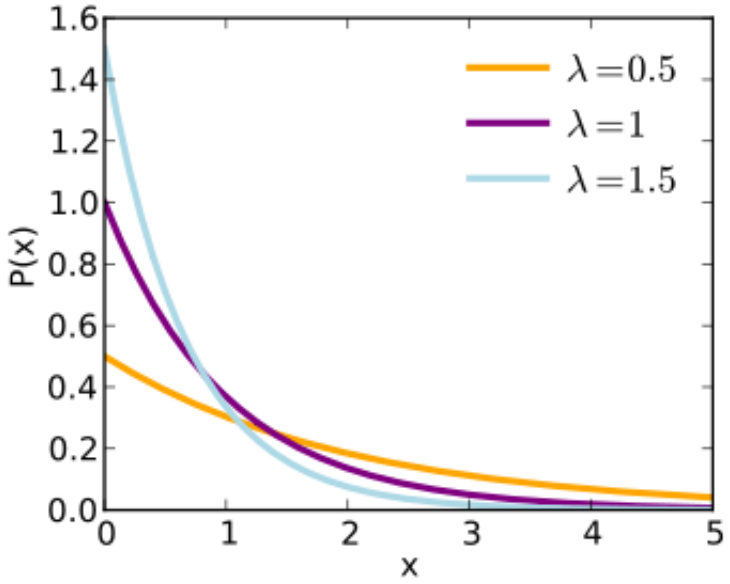


Figure 1.5: An example of Expoiential distribution of data.

- Log-normal distribution is the probability distribution of a random variable whose logarithm follows a normal distribution. It models phenomena whose relative growth rate is independent of size, which is true of most natural phenomena including the size of tissue and blood pressure, income distribution, and even the length of chess games.
  - Let Z be a standard normal variable, which means the probability distribution of Z is normal centered at 0 and with variance 1. Then a log-normal distribution is defined as the probability distribution of a random variable:

    $$X = e^{\mu+\sigma Z} \tag{1.9}$$

    where μ and σ are the mean and standard deviation of the logarithm of X, respectively.
  - The term "log-normal" comes from the result of taking the logarithm of both sides:

    $log(X) = \mu + \sigma Z$

    As Z is normal, $\mu + \sigma Z$ is also normal (the transformations just scale the distribution, and do not affect normality), meaning that the logarithm of X is normally distributed (hence the term log-normal).
  - For most natural growth processes, the growth rate is independent of size, so the log-normal distribution is followed. As a result, the log-normal distribution has heavy applications to biology and finance, two areas where growth is an important area of study. In particular, epidemics and stock prices tend to follow a log-normal distribution.

- Multivariate normal distribution
  - It is a vector in multiple normally distributed variables, such that any linear combination of the variables is also normally distributed. It is mostly useful in extending the central limit theorem to multiple variables, but also has applications to Bayesian inference and thus machine learning, where the multivariate normal distribution is used to approximate the features of some characteristics.[22]
  - A random vector $X = (X_1, X_2, ..., X_n)$ is multivariate normal if any linear combination of the random variables $X_1, X_2, ..., X_n$ is normally distributed. In other words, $a_1X_1 + a_2X_2 + ... + a_nX_n$ has a normal distribution for any constants $a_1, a_2, ..., a_n$.
  - Equivalently, multivariate distributions can be viewed as a linear transformation of a collection of independent standard normal random variables, meaning that if z is another random vector whose components are all standard random variables, there exists a matrix A and vector μ such that $x = Az + \mu$.
  - The multivariate normal distribution is useful in analyzing the relationship between multiple normally distributed variables.

1.4 Basic Terms

Classification algorithms: discrete valued output (e.g. 0 or 1). They can be divided into:

- Binary classification problems - only 2 discrete values. **Usually** - we give label '1' to rare class that we want to detect.
- Multi-class classification problems - more discrete values than 2. This is usually solved by using **one-vs-all** (or one-vs-rest) technique, where we have 2 classes (1 specific and all others) and then we iterate over all classes. So n classes, n binary classification problems, n trained classifiers as a result.

Regression algorithms: predict continuous valued output given an unlabeled example. The regression problem is solved by a regression learning algorithm that takes a collection of labeled examples as inputs and produces a model that can take an unlabeled example as input and output a target.

Image classification: is there a car on a picture? So, we are working with 1 object (car).

Classification with localization: localize (an select with bounding box) car where it is located on a picture. So again a single object (car). For example, localization of 4 landmarks of a human face (beginning and end of eyes). Each landmark is X and Y coordinate, so ground truth is a vector of 9 numbers - the first is if it is

[22] https://brilliant.org/wiki/multivariate-normal-distribution/

an object or not, and the other 8 are coordinates for all 4 landmarks. Then loss function is computed (it is still just 1 number) by summing over all such 9 numbers obtained by ground truth minus predicted values (maybe using power of 2 on each deviation) .

Detection: it is classification with localization in a given picture, but we may have multiple objects. All are selected, so we are working with many objects. This can be achieved with sliding window, but nowadays we don't use a traditional algorithm (computational cost). We use "convolutional implementation", see the next figure.

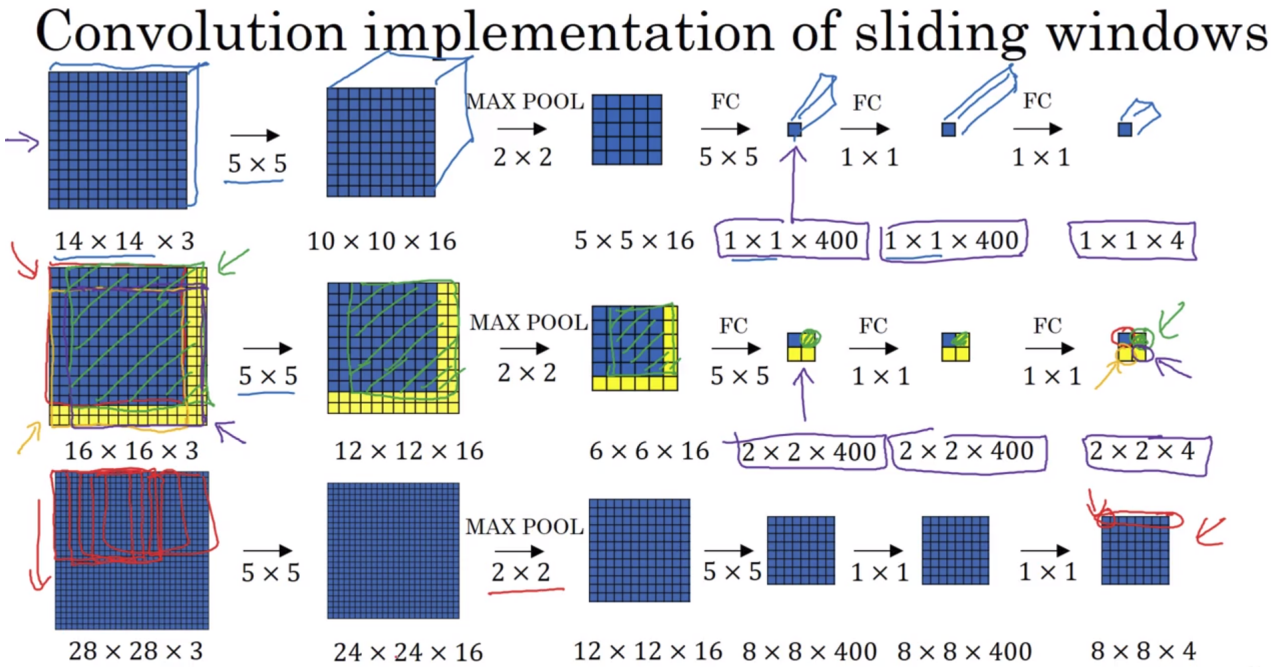


Figure 1.6: Convolutional implementation(s) of sliding window for object detection (2014). Instead of going through an input image step by step, with many sliding windows, convolutional layer will do this in more optimal way (all at once). This still can have a problem with outputting the most accurate bounding boxes (shape can be rectangular a bit as well).

Verification vs Recognition:

- verification: input is image/name/ID, output is whether the input image is that claimed person. This is 1:1 problem.
- recognition is much harder than verification: it has database of K people, and just an image on its input. Output is ID if the image is any of K people; "not recognized" otherwise. This is 1:K problem (so K verification steps). An example:

- Your face verification system is mostly working well. But since Kian got his ID card stolen, when he came back to the house that evening he couldn't get in!
- To reduce such shenanigans, you'd like to change your face verification system to a face recognition system. This way, no one has to carry an ID card anymore. An authorized person can just walk up to the house, and the front door will unlock for them!
- No more person's name on the input!

Fuzzy set

- It is a generalization of a set.
- For each element x in a fuzzy set S, there is a membership function $\mu_S(x) \in [0, 1]$ that defines the membership strength of x to the set S.
- We say that x weakly belongs to a fuzzy set S if $\mu_S(x)$ is close to 0. On the other hand, if $\mu_S(x)$ is close to 1, then x has a strong membership in S.
- If $\mu(x) = 1$ for all $x \in S$, then a fuzzy set S becomes equivalent to a normal, non-fuzzy set.

Shallow vs. Deep learning:

- A shallow learning algorithm learns the parameters of the model directly from the features of the training examples. Most supervised learning algorithms are shallow.
- The notorious exceptions are neural network learning algorithms, specifically those that build neural networks with more than one layer between input and output. Such neural networks are called deep neural networks. In deep neural network learning (or, simply, deep learning), contrary to shallow learning, most model parameters are learned not directly from the features of the training examples, but from the outputs of the preceding layers.

Model-Based vs. Instance-Based Learning

- Most supervised learning algorithms are **model-based**. Model-based learning algorithms use the training data to create a model that has parameters learned from the training data. After the model was built, the training data can be discarded.
- **Instance-based** learning algorithms use the whole dataset as the model. One example of algorithm is k-Nearest Neighbors (kNN). In classification, to predict a label for an input example the kNN algorithm looks at the close neighborhood of the input example in the space of feature vectors and outputs the label that it saw the most often in this close neighborhood.

One-Shot Learning

- This is one of the challenges of face recognition. For most face recognition applications you need to be able to **recognize** a person given just one single example example of that person's face.
  - So, we want to build a model that can recognize that two photos of the same person represent that same person. If we present to the model two photos of two different people, we expect the model to recognize that the two people are different.
  - To solve such a problem, we could go a traditional way and build a binary classifier that takes two images as input and predict either true (when the two pictures represent the same person) or false (when the two pictures belong to different people). However, in practice, this would result in a neural network twice as big as a typical neural network, because each of the two pictures needs its own embedding sub-network. Training such a network would be challenging not only because of its size, but also because the positive examples would be much harder to obtain than negative ones (highly imbalanced problem).
  - It's a common misconception that for one-shot learning we need only one example of each entity for training. In practice, we need more than one example of each person for the person identification model to be accurate. It's called one-shot because of the most frequent application of such a model: face-based authentication. For example, such a model could be used to unlock your phone. If your model is good, then you only need to have one picture of you on your phone and it will recognize you, and also it will recognize that someone else is not you.
- **For example**: we have let's say K faces of people in our database. So we are using softmax in our deep neural network for multi-task classification. If a new person come, we would need to retrain the whole network, which is not wise. Instead, a similarity function is being used. Bigger it is, more different faces are. This similarity function can be learned with ANN, more specifically **Siamese network** (see Section 5.7), that solves this problem effectively.

Zero-Shot Learning

Note: This is a relatively new research area, so there are no algorithms that proved to have a significant practical utility yet.

- In zero-shot learning (ZSL) we want to train a model to assign labels to objects. The most frequent application is to learn to assign labels to images. However, contrary to standard classification, **we want the model to be able to predict labels that we didn't have in the training data**.

- The trick is to use embeddings (for example word embeddings) no just to represent the input x, but also to represent the output y.
  - **Word embeddings** have such a property that each dimension of the embedding represents a specific feature of the meaning of the word. For example, if our word embedding has four dimensions (usually they are much wider, between 50 and 300 dimensions), then these four dimensions could represent such features of the meaning as animalness, abstractness, sourness, and yellowness (yes, sounds funny, but it's just an example). So the word bee would have an embedding like this $[1, 0, 0, 1]$, the word yellow like this $[0, 1, 0, 1]$, the word unicorn like this $[1, 1, 0, 0]$. The values for each embedding are obtained using a specific training procedure applied to a vast text corpus.
- Now, in our classification problem, we can replace the label y_i for each example i in our training set with its word embedding and train a multi-label model that predicts word embeddings. To get the label for a new example x, we apply our model f to x, get the embedding $\hat{y}$ and then search among all English words those whose embeddings are the most similar to $\hat{y}$ using cosine similarity.
- **An example**:
  - Take a zebra - it is white, it is a mammal, and it has stripes.
  - Take a clownfish - it is orange, not a mammal, and has stripes.
  - Now take a tiger - it is orange, it is a mammal, and it has stripes.
  - If these three features are present in word embeddings, the CNN would learn to detect these same features in pictures. Even if the label tiger was absent in the training data, but other objects including zebras and clownfish were, then the CNN will most likely learn the notion of mammalness, orangeness, and stripeness to predict labels of those objects.
  - Once we present the picture of a tiger to the model, those features will be correctly identified from the image and most likely the closest word embedding from our English dictionary to the predicted embedding will be that of tiger.

Self-learning

- Historically, there were multiple attempts at solving semi-supervised learning problem. None of them could be called universally acclaimed and frequently used in practice.
- One frequently cited method is called **self-learning**. In self-learning, we use a learning algorithm to build the initial model using the labeled examples. Then we apply the model to all unlabeled examples and label them using the model. If the confidence score of prediction for some unlabeled example x is higher than some threshold (chosen experimentally), then we add this labeled example to our training set, retrain the model and continue like this until a stopping criterion is satisfied. We could stop, for example, if the accuracy of the model has not been improved during the last m iterations.

Grammar induction

- Also known as grammatical inference. It is a process of learning a formal grammar (usually as a collection of re-write rules or as finite state machine or automation of some kind) from a set of observations, thus constructing a model which accounts for the characteristics of the observed objects.
- More generally, grammatical inference is that branch of machine learning where the instance space consists of discrete combinatorial objects such as strings, trees and graphs.
- Finite state machines of various types, and context-free grammars are common.
- The applications of grammar induction are for example semantic parsing (task of converting a natural language utterance to a logical form: a machine-understandable representation of its meaning; semantic parsing can thus be understood as extracting the precise meaning of an utterance; applications of semantic parsing include machine translation, question answering and code generation), natural language understanding, example-based translation (this is essentially a translation by analogy), morpheme analysis, and so on. But also for loss-less data compression.

Speech recognition

- **In speech recognition problem**, you are given an input x (audio clip), and you job is to automatically find a text transcript y. Audio clip has **time** on X-axis, and what microphone does is to measure very small changes in **air pressure** (Y-axis). But a human ear doesn't process raw wave forms. Human ear has physical structures that measures the **amounts of intensity of different frequencies**. Thus, a common preprocessing step is to run an input audio clip and generate a spectogram (time on X-axis and frequencies on Y-axis) that shows intensities of different colors that shows the amount of energy (how loud is a sound at different frequencies and different times). And this spectogram is common preprocessing step for some algorithm, and a human ear is doing similar "preprocessing step". **Using a spectogram and optionally a 1D conv layer is a common preprocessing step prior to passing audio data to an RNN, GRU or LSTM.**
- **Attention models** (see Section 5.6) can be used for speech recognition. Or an alternative, **CTC models** (see below) also work well. In such end-to-end learning, there is no more need for phonemes (hand-engineered basic units of sound, historically very important, but were replaced by ANN with a lot of data - 100k hours in commercial systems for example, and maybe 300-3k hours in academia).
- **CTC** (Connectionist temporal classification) **cost** (from 2006) is used for speech recognition problems. An idea is to use RNN (in practice usually bidirectional LSTM) with many-to-many architecture when the number of input and output size is the same.

  - But notice that the number of timesteps is very large and in speech recognition, usually the **number of input time steps is much bigger than the number of output timesteps**. So, for example, if you have 10 seconds of audio and your features come at a 100 hertz. So, 100 samples per second then 10 second audio clip would end up with a 1k inputs. But your output might not have a 1k characters.
  - CTC allows to generate output like this "ttt_h_eee_____[ ]_____qqq___ ... " ('[ ]' denotes space character) for input "The quick ...". Basic rule for CTC cost function is to collapse repeated characters not separated by "blank" (underscore '_' character in this example).

- Trigger word detection system can be done with a smaller amount of data. See the next figure for more details.

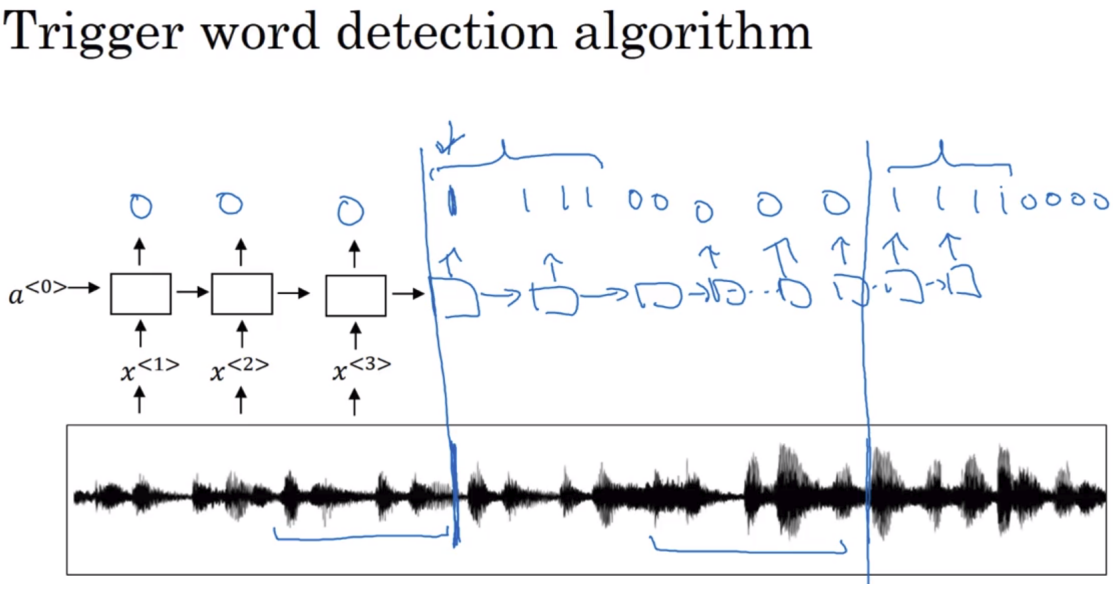


Figure 1.7: An example of trigger word detection system. RNN can be used on an audio clip (or spectogram). This is a supervised learning task, and where some trigger word ends, here a sequence of '1' symbols starts. When an trigger word starts, then a sequence of '1's ends and starts a sequence of '0's. $x^{<t>}$ are features of the audio (such as spectogram features) at time t.

Association rule learning algorithms: association rule learning methods extract rules that best explain observed relationships between variables in data.

Training sample: a training sample (or training instance or training example) is a **data point x** in an available training set that we use for working on a predictive modeling task. For example, if we are interested in classifying emails, one email in our dataset would be one training sample.

Skewed classes: if there are for instance 2 classes, and one is significantly bigger, for example we have 1% of examples in the first class, and 99% of examples in the second one.

Target function: in predictive modeling, we are typically interested in modeling a particular process; we want to learn or approximate a particular function that, for example, let's us distinguish spam from non-spam email. The target function f(x) = y is the **true function** f that **we want to model**.

Hypothesis

- A hypothesis is a certain function (denoted as "h" on Figure 1.8) that we believe (or hope) **is similar to the true function**, the target function that we want to model.
- In context of email spam classification, it would be the rule we came up with that allows us to separate spam from non-spam emails.

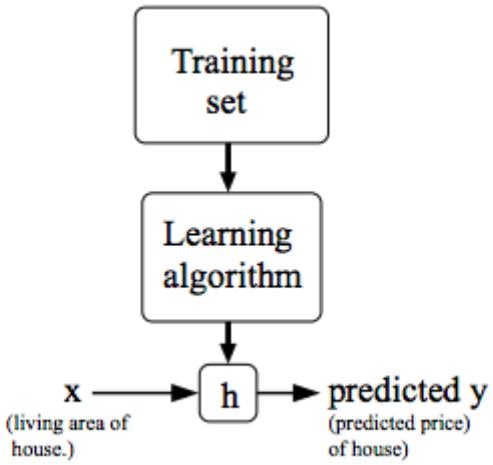


Figure 1.8: Hypothesis

- **Example** of a concrete hypothesis for univariate linear regression: $h(x) = 50+0.1x$
- For binary classification problems: the decision boundary is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.

Model: in machine learning field, the terms hypothesis and model are often used **interchangeably**.

Model types: from here[23]

[23] https://stackoverflow.com/questions/879432/what-is-the-difference-between-a-generative-and-discriminative-algorithm

- **Generative models** (e.g. Gaussian Mixture model, Hidden Markov Models, Naive Bayes, Restricted Boltzmann Machine, Generative Adversarial networks) generate all values for a given phenomenon. Discriminative models infer outputs based on inputs, while generative models generate both inputs and outputs, typically given some hidden parameters. They are trying to model a probability distribution of data. They are trying to find such distribution from which similar data could be generated. If the observed data are truly sampled from the generative model, then fitting the parameters of the generative model to maximize the data likelihood is a common method (Maximum Likelihood estimation - just multiplication of conditional probabilities and looking for maximization). **They learn JOINT probability distribution**.
- **Discriminative models** (e.g. Linear Regression, Logistic Regression, SVM, Perceptron, Neural Networks, or Random Forests) work differently, we don't model probability distribution of data, but they are trying directly to model something for distinguishing classes - direct estimation of probabilities if an item is in one class or another. They can use regularization (penalization for model complexity, it prevents overfitting). They are inherently supervised and cannot easily support unsupervised. **They learn CONDITIONAL probability distribution.**

Probabilistic Graphic Model (PGM): representation of conditional independent relationships between the nodes (random variables). The most important characteristic of PGMs is the ability of being translated from a graph to a joint distribution. These models can be discrete or continuous.

Classifier: this is a **special case of a hypothesis** (nowadays, often learned by a machine learning algorithm). A classifier is a hypothesis or **discrete-valued function** that is used to **assign (categorical) class labels to particular data points**.

- In the email classification example, this classifier could be a hypothesis for labeling emails as spam or non-spam.
- However, a hypothesis must not necessarily be synonymous to a classifier. In a different application, our hypothesis could be a function for mapping study time and educational backgrounds of students to their future SAT scores.
- **A classier is a function that maps an unlabeled instance to a label using internal data structures. An inducer** (an induction algorithm), **builds a classifier from a given dataset** [6]. Why inducer? Because learners do **induction** - from some particular observations or evidences, create something more general - some larger hypothesis (from data to knowledge). This is an opposite to **deduction**, from some general knowledge to some specific (from knowledge to data).

The Hadamard product: also known as **Schur product**, is simply element-wise multiplication of two vectors of the same dimension.

Arg Max

- Given a set of values $A = \{a_1, a_2, ..., a_n\}$, the operator $max_{a \in A} f(a)$ returns the highest value $f(a)$ for all elements in the set A (so, max returns the biggest value of a function result).
- On the other hand, the operator $arg\,max_{a \in A} f(a)$ returns such element of the set A, that maximizes $f(a)$ (so, $arg\,max$ returns the input value to some function, which is maximized because of this value; but we want to know what caused it).

Derivative and Gradient

- **A derivative** f' of function f is a function or a value that describes how fast f grows (or decreases). If the derivative is negative at some point x, that the function decreases at this point; if derivative is positive at some point, then the function increases there, and if the derivative is 0 at x, that means that the function's slope at x is horizontal (local or global optima).
- The process of finding a derivative is called **differentiation**.
- Derivatives for basic functions are known, and it a function is not basic, we can find its derivative using the **chain rule**.
- **Gradient** is the generalization of derivatives for functions that take several inputs (or one input in the form of a vector or some other complex structure). **So gradient of a function is a vector of partial derivatives**.
- **Gradient of function** f, denoted as ∇f is given by the vector $[\frac{\partial f}{\partial x^{(1)}}, \frac{\partial f}{\partial x^{(2)}}]$ where $\frac{\partial f}{\partial x^{(1)}}$(similarly for $\partial x^{(2)}$) is partial derivative of function f with respect to $x^{(1)}$.

Bag of words

- Text converted to a feature vector (this vector is called bag of words) by taking dictionary of English words (sorted alphabetically and having 20k words) and using it.
- For example, the first feature is equal to 1 if the email message contains the word "a", otherwise this feature is 0. Similarly for second word, and the algorithm will continue with the third word from dictionary, and so on until 20,000th word. Each data sample will thus have dimensionality of 20k with information about presence of a given word or not.

Neural style transfer: you want to recreate an image according to some painting for instance. For estimating how good is a generated image, we need custom cost function. Idea from 2015.

$$J(G) = \alpha.J_{content}(C, G) + \beta J_{style}(S, G) \tag{1.10}$$

- where these two cost functions inside measures a similarity between input images, and α and β are hyperparameters to specify a relative weighting between content cost and style cost.
- $J_{content} = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2$ is calculated as a similarity between activation of layer l on the images C and G. If these activations are similar, both images have similar content.
- J_{style} is calculated as correlation between activations across different channels in layer l (how often they occur together).
- Following the original NST paper, it can be used the VGG network. Specifically, VGG-19 (it is CNN), a 19-layer version of the VGG network. This model has already been trained on the very large ImageNet database, and thus has learned to recognize a variety of low level features (at the earlier layers) and high level features (at the deeper layers).
- How do you ensure the generated image G matches the content of the image C?
  - As we saw in lecture, the earlier (shallower) layers of a ConvNet tend to detect lower-level features such as edges and simple textures, and the later (deeper) layers tend to detect higher-level features such as more complex textures as well as object classes.
  - We would like the "generated" image G to have similar content as the input image C. Suppose you have chosen some layer's activations to represent the content of an image. In practice, you'll get the most visually pleasing results if you choose a layer in the middle of the network--neither too shallow nor too deep.
  - So, the content cost takes a hidden layer activation of the neural network, and measures how different $a^{(C)}$ and $a^{(G)}$ are. When we minimize the content cost later, this will help make sure G has similar content as C.

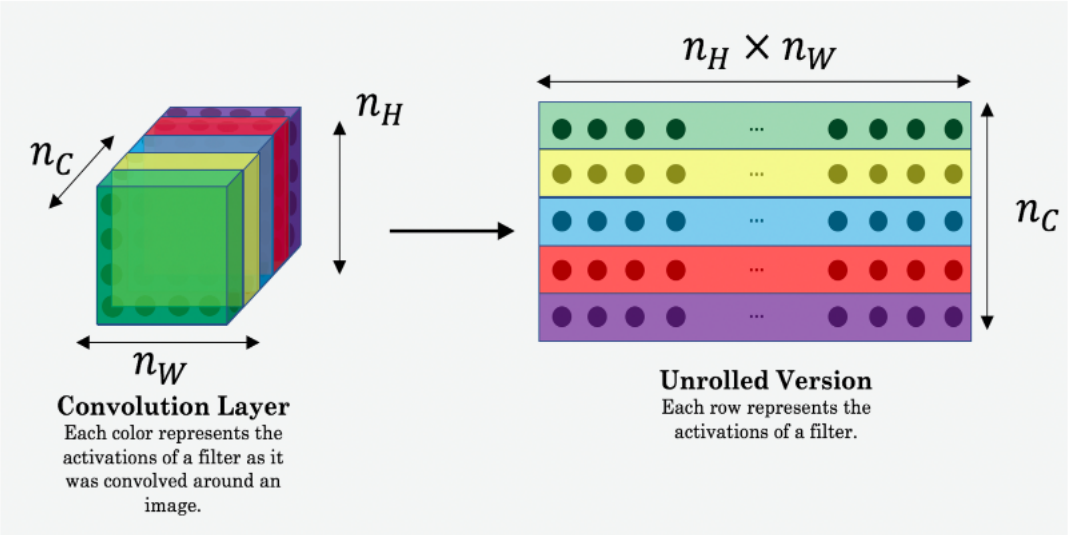


Figure 1.9: Unrolling step for easier computation (not a neccessary step though).

- Style matrix
  - The style matrix is also called a "Gram matrix." In linear algebra, the Gram matrix G of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are $G_{ij} = v_i^T v_j = np.dot(v_i, v_j)$. In other words, G_{ij} compares how similar v_i is to v_j: If they are highly similar, you would expect them to have a large dot product, and thus for G_{ij} to be large. The value G_{ij} measures how similar the activations of filter i are to the activations of filter j.
  - One important part of the gram matrix is that the diagonal elements such as G_{ii} also measures how active filter i is. For example, suppose filter ii is detecting vertical textures in the image. Then G_{ii} measures how common vertical textures are in the image as a whole: If G_{ij} is large, this means that the image has a lot of vertical texture.
  - By capturing the prevalence of different types of features (G_{ii}), as well as how much different features occur together (G_{ij}), the Style matrix G measures the style of an image.

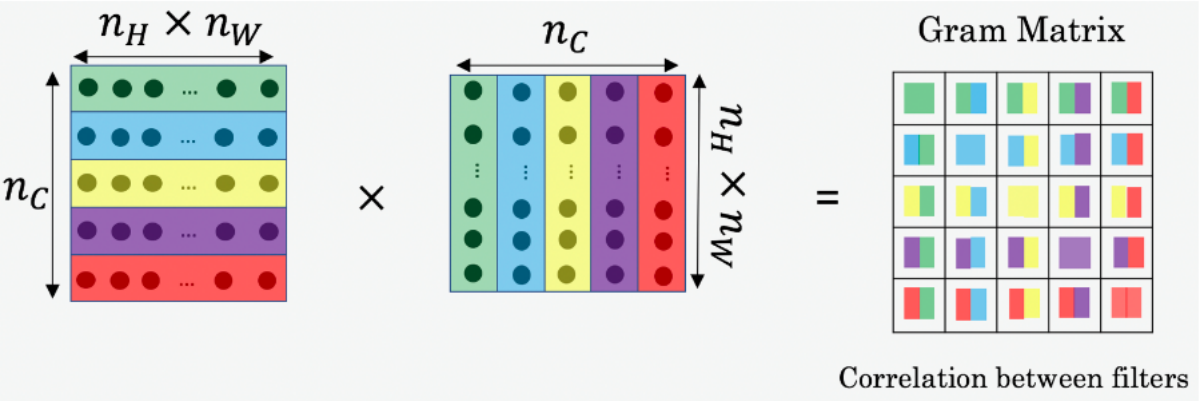


Figure 1.10: Unrolling step and computation of Gram (Style) Matrix.

- The next step is to compute Style cost (computed similarly as content cost). The goal will be to minimize the distance between the Gram matrix of the "style" image S and that of the "generated" image G
  - The style of an image can be represented using the Gram matrix of a hidden layer's activations. However, we get even better results combining this representation from multiple different layers. This is in contrast to the content representation, where usually using just a single hidden layer is sufficient.
  - Minimizing the style cost will cause the image G to follow the style of the image S
- The total cost is a linear combination of the content cost $J_{content}(C, G)$ and the style cost $J_{style}(S, G)$. α and β are hyperparameters that control the relative weighting between content and style.

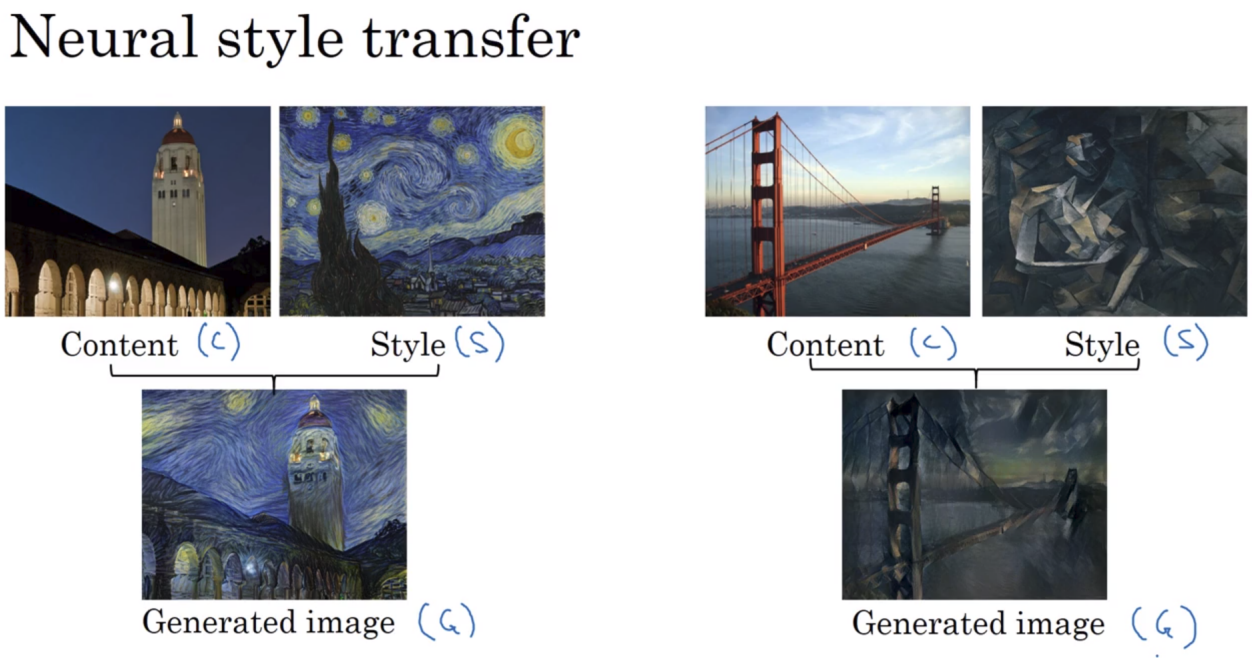


Figure 1.11: Two examples of neural style transfer between an image and piccasso paintings.

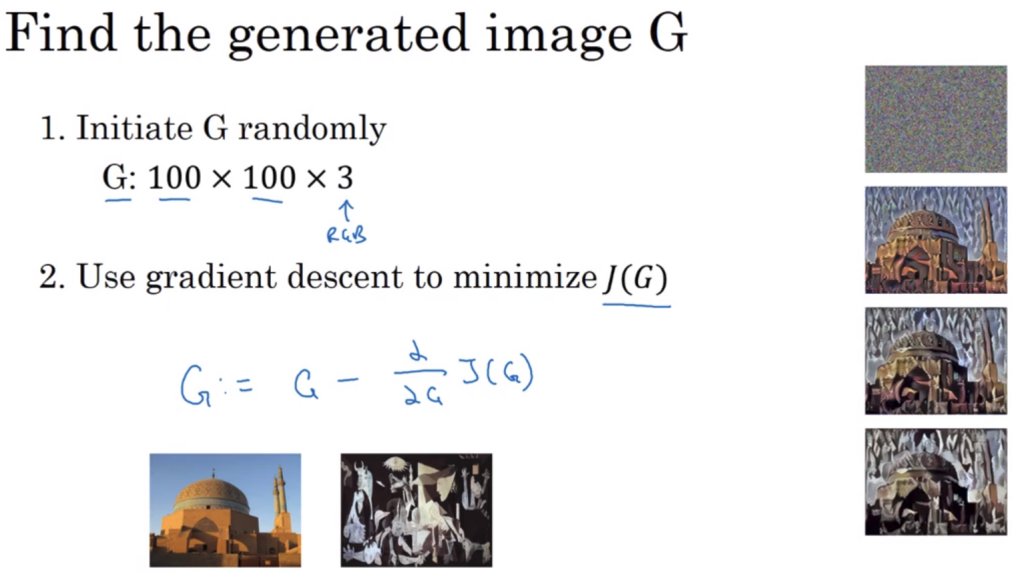


Figure 1.12: Computation of gradient descent using neural style transfer between an image and piccasso paintings. As you run gradient descent, you minimize the cost function $J(G)$ slowly through the pixel value so then you get slowly an image that looks more and more like your content image rendered in the style of your style image. So, just to be clear, we initialize the "generated" image as a noisy image created from the content_image. By initializing the pixels of the generated image to be mostly noise but still slightly correlated with the content image, this will help the content of the "generated" image more rapidly match the content of the "content" image.

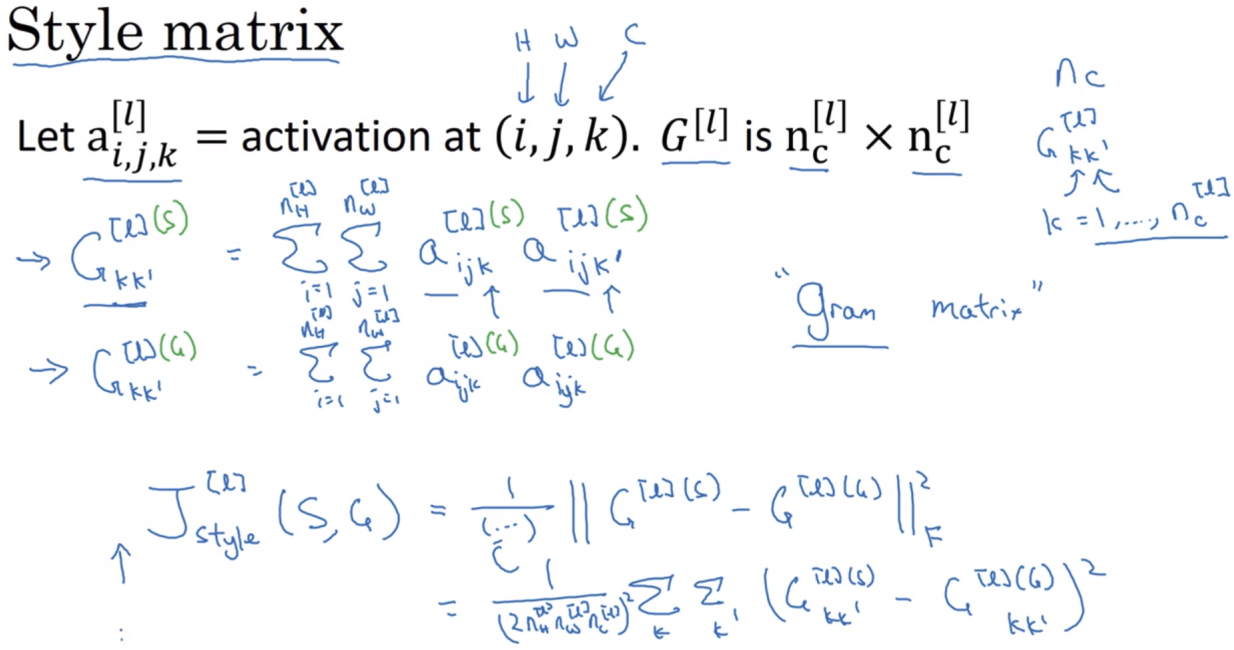


Figure 1.13: Computation of style matrix for neural style transfer.

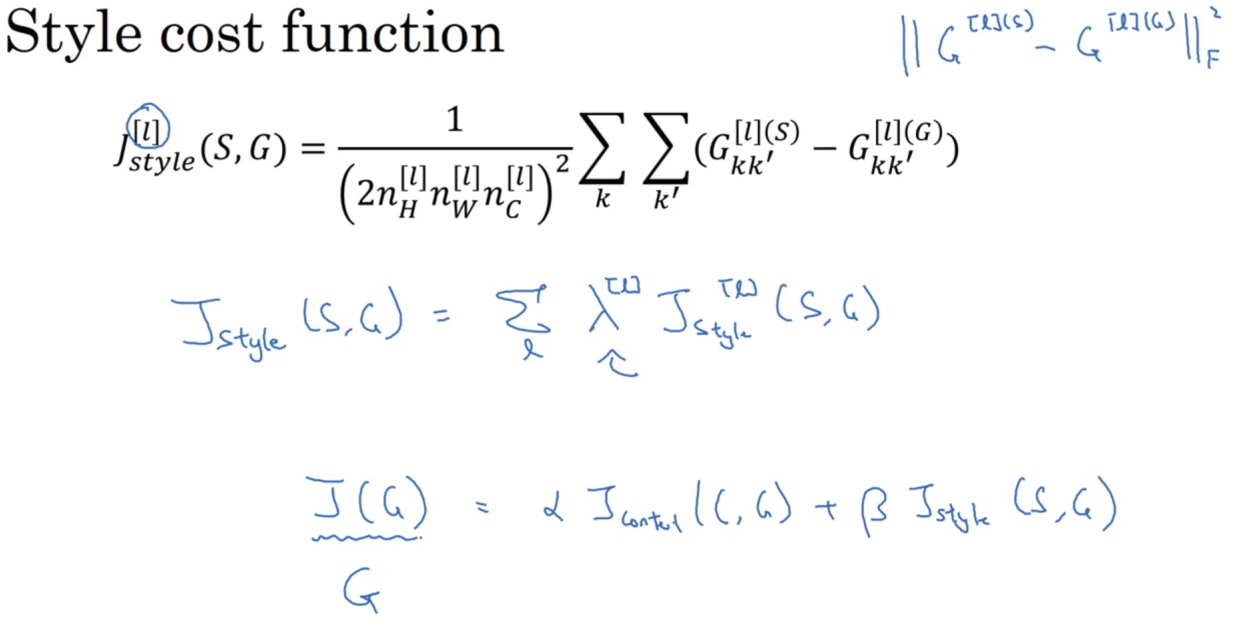


Figure 1.14: Computation of style cost function for particular layer l for neural style transfer. So, the pixel values of generated image G are updated in each iteration of the optimization algorithm.

Handling multiple inputs

- Often in practice, you will work with multimodal data. For example, your input could be an image and text and the binary output could indicate whether the text describes this image. It's hard to adapt shallow learning algorithms to work with multimodal data.
- You could train one shallow model on the image and another one on the text. Then you can use a model combination technique using ensemble learning. If you cannot divide your problem into two independent sub-problems, you can try to vectorize each input (by applying the corresponding feature engineering method) and then simply concatenate two feature vectors together to form one wider feature vector. For example, if your image has features $[i(1), i(2), \quad i(3)]$ and your text has features $[t(1), t(2), t(3), t(4)]$ your concatenated feature vector will be $[i(1), i(2), i(3), t(1), t(2), t(3), t(4)]$.
- With neural networks, you have more flexibility. You can build two subnetworks, one for each type of input. For example, a CNN subnetwork would read the image while an RNN subnetwork would read the text. Both subnetworks have as their last layer an embedding: CNN has an embedding of the image, while RNN has an embedding of the text. You can now concatenate two embeddings and then add a classification layer, such as softmax or sigmoid, on top of the concatenated embeddings. Neural network libraries provide simple to use tools that allow concatenating or averaging layers from several subnetworks.

Topic Modeling

- In text analysis, this is a prevalent **unsupervised learning problem**. **You have a collection of text documents, and you would like to discover topics present in each document**.
- **Latent Dirichlet Allocation (LDA)** is a very effective algorithm of topic discovery. You decide how many topics are present in your collection of documents, and the algorithm assigns a topic to each word in this collection. Then, to extract the topics from a document, you simply count how many words of each topic are present in that document.

Probabilistic Graphical Models

- **Conditional random fields** (CRF) is one example of probabilistic graphical models (PGMs). With CRF you can model the input sequence of words and relationships between the features and labels in this sequence as a sequential dependency graph.
- More generally, a PGM can be any graph. A graph is a structure consisting of a collection of nodes and edges that join a pair of nodes. Each node in PGM

represents some random variable (values of which can be observed or unobserved), and edges represent the conditional dependence of one random variable on another random variable. For example, the random variable "sidewalk wetness" depends on the random variable "weather condition". By observing values of some random variables, an optimization algorithm can learn from data the dependency between observed and unobserved variables. If you decide to learn more about PGMs, they are also known under names of **Bayesian networks, belief networks**, and **probabilistic independence networks**.

Markov Chain Monte Carlo

- If you work with graphical models and want to sample examples from a very complex distribution defined by the dependency graph, you could use Markov Chain Monte Carlo (MCMC) algorithms.
- MCMC is a class of algorithms for sampling from any probability distribution defined mathematically. Sampling from standard distributions, such as normal or uniform, is relatively easy because their properties are well known. However, the task of sampling becomes significantly more complicated when the probability distribution can have an arbitrary form defined by a complex formula.

Handling multiple outputs

- In some problems, you would like to predict multiple outputs for one input. Some problems with multiple outputs can be effectively converted into a multi-label classification problem. Especially those that have labels of the same nature (like tags) or fake labels can be created as a full enumeration of combinations of original labels. However, in some cases the outputs are multimodal, and their combinations cannot be effectively enumerated.
- Consider the following example: you want to build a model that detects an object on an image and returns its coordinates. In addition, the model has to return a tag describing the object, such as "person," "cat," or "hamster." Your training example will be a feature vector that represents an image. The label will be represented as a vector of coordinates of the object and another vector with a one-hot encoded tag.
  - To handle a situation like that, you can create one subnetwork that would work as an encoder. It will read the input image using, for example, one or several convolution layers. The encoder's last layer would be the embedding of the image. Then you add two other subnetworks on top of the embedding layer:
    * The first subnetwork will take the embedding vector as input and predicts the coordinates of an object. It can have a ReLU as the last layer, which is a good choice for predicting positive real numbers, such as coordinates; this subnetwork could use the mean squared error cost C_1.

    - The second subnetwork will take the same embedding vector as input and predict the probabilities for each tag. It can have a softmax as the last layer, which is appropriate for the probabilistic output, and use the averaged negative log-likelihood cost C_2 (also called cross-entropy cost).
  - Obviously, you are interested in both accurately predicted coordinates and the tags. However, it is impossible to optimize the two cost functions at the same time. By trying to optimize one, you risk hurting the second one and the other way around. What you can do is add another hyperparameter γ in the range $(0, 1)$ and define the combined cost function as $\gamma C_1 + (1 - \gamma)C_2$. Then you tune the value for γ on the validation data just like any other hyperparameter. .

End-to-end learning

- **Sentiment classification**
  - The problem of recognizing positive vs. negative opinions. For example, examining online product reviews if the writer liked or disliked a product.
  - One of the problems of sentiment classification is, that there is usually not enough labeled data. But with word embeddings, far less data are needed.
  - There are multiple algorithms using word embeddings:
    - From a given sentiment, get a number (position in dictionary) for each word and create a one-hot representation of them, and for each - multiply this one-hot vector by E (which you can learn from much bigger corpus), and then you have embedding vector (for each word). Then, the algorithm averages all embedding vectors (from a given sentiment) which result is given to a softmax classifier (over 5 possible outcomes, if a given sentiment can be rated by 1 to 5 stars for instance). However, this algorithm completely lacks an order of the words in a given sentiment.
    - So, instead of using some averaging (previous example/algorithm), RNN can be used with many-to-one architecture. "Many" are basically all embedding vectors from the input sentiment, "one" is softmax.
- To build a system for sentiment classification, it is needed to build a **pipeline of 2 components:**
  - **Parser**. A system that annotates the text with information identifying the most important words. For example, the parser might used to label all the adjectives and nouns.
  - **Sentiment classifier**. A learning algorithm that takes as input the annotated text and predicts the overall sentiment. The parser's annotation could help this learning algorithm greatly: By giving adjectives a higher weight, your algorithm will be able to quickly hone in on the important words such as "great" and ignore less important words such as "this".

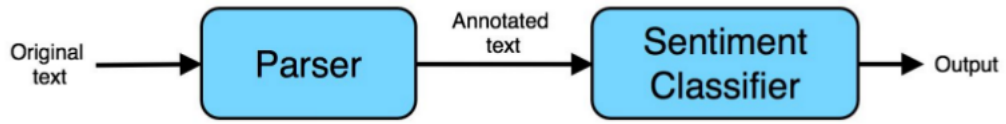


Figure 1.15: For sentiment classification system it is needed to have pipeline of 2 tasks, as it is displayed on this figure.

- Or another example, speech recognition system - pipeline may have 3 components: 1) **extract hand-designed features** (MFCC), phoneme recognizer, and then 2) **final recognizer.** This and the previous **pipelines were linear** - the input is sequentially passed from one staged to the next. However pipelines can be more complex, like it is depicted on the figure below.

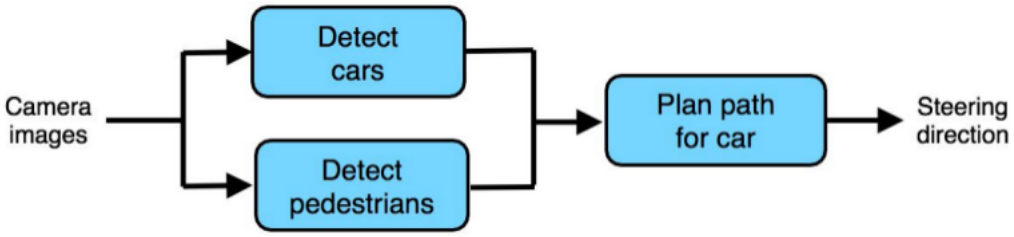


Figure 1.16: An example of architecture of system for an autonomous cat detection. A final component plans a path for a new car that avoids other cars and pedestrians. It should be noted, that not every component in a pipeline has to be learned. Many algorithms do not involve learning. Additionally, each of these 3 components is a relatively simple function - and can thus be learned with less data than the purely end-to-end approach that requires a lot of data.

- However, recent trend is toward replacing such pipeline systems with a single learning algorithm - an **end-to-end learning algorithm**[24]. The algorithm would simply take as input the raw original text, and try to directly recognize the sentiment. ANN are commonly used in end-to-end learning systems.
- **Feature engineering simplifies the input and throws some information away,** which can limit the overall system's performance. End-to-end learning is

[24]The term "end-to-end" refers to the fact that we are asking the learning algorithm to go directly from the input to the desired output.

usable if data is abundant. If the learning algorithm is a large-enough neural network and if it is trained with enough training data, it has the potential to do very well, and perhaps even approach the optimal error rate.

- **However, end-to-end learning is not always a good choice.** Allowing **hand-engineered components also has some advantages. They help simplify the problem for the learning algorithm. It is a way how to manually inject knowledge into the system.**
  - For example, in speech recognition system, to recognize phonemes with a separate module in the pipeline can also help the learning algorithm understand basic sound components (because phonemes are a reasonable representation of speech) and therefore improve the system's performance.
  - Having more hand-engineered components generally allows a speech system to learn with less data. When we don't have much data, this knowledge is useful. If you are working on a machine learning problem where the training set is very small, most of your algorithm's knowledge will have to come from your human insight (i.e., from your "hand engineering" components).
- When to choose - "pipelines" approach vs end-to-end learning? It's mostly about data.
  - For example, for end-to-end learning system of driving autonomous car, it is needed to have a fleet of specially-instrumented cars (for having steering directions captured), and a huge amount of driving to cover a wide range of possible scenarios. On the other hand, data with pedestrians and cards as images are easier to obtain.
  - In summary, when deciding what should be the components of a pipeline, try to build a pipeline where each component is a relatively "simple" function that can therefore be learned from only a modest amount of data.
- New trend in end-to-end learning, is not just to predict some real number or an integer (or class), but it is letting us directly **learn outputs which are much more complex than a number.**
  - For example, for an image where x is a yellow bus with trees, a learning algorithm would produce y = "A yellow bus driving down a road with green trees and green grass in the background."

Bleu Score

- Bilingual evaluation understudy (from 2002) - given a machine generated translation, it allows you to automatically compute a score that measures **how good is a given machine translation** (if you have 2 good translations and want to decide what would person prefer - these 2 are equally good answer).

- Central idea - quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is". **It is a single number evaluation metric.** That is why Bleu score was revolutionary in machine translation. But it is also used in image captioning system. It is not used in speech recognition as there we have an exact representation of the ground truth.
- Output is always a number between 0 and 1. This value indicates how similar the candidate text is to the reference texts, with values closer to 1 representing more similar texts.
- We will give each word credit only up to the maximum number of times it appears in the all reference sentences. **Bleu score is a modified precision defined on n-grams** (not just separate words, but maybe pairs - bigrams, or triplets, and so on), ($\hat{y}$is output from machine translation):

$$p_n = \frac{\sum_{n-grams \in \hat{y}} Count_{clip}(n-gram)}{\sum_{n-grams \in y} Count\,(n-gram)} \tag{1.11}$$

- If $p_n = 1$ then you are exactly equal to one of the human (translation) references.
- By convention, you compute p_1, p_2, p_3, and p_4, and then you compute **the average of the results** and then **final Bleu score** is defined as $BP * exp^{avg}$. BP is a parameter and stands for so called **brevity penalty**. This is due to observation, that shorter translations have bigger p_n score, and we don't want to always output very short translations. BP penalizes translations which are very short. BP = 1 if output of machine translation is bigger than human reference output. Otherwise, it is equal to:

$$exp(1 - \frac{machine\,translation\,output\,length}{human\,reference\,output\,length}) \tag{1.12}$$

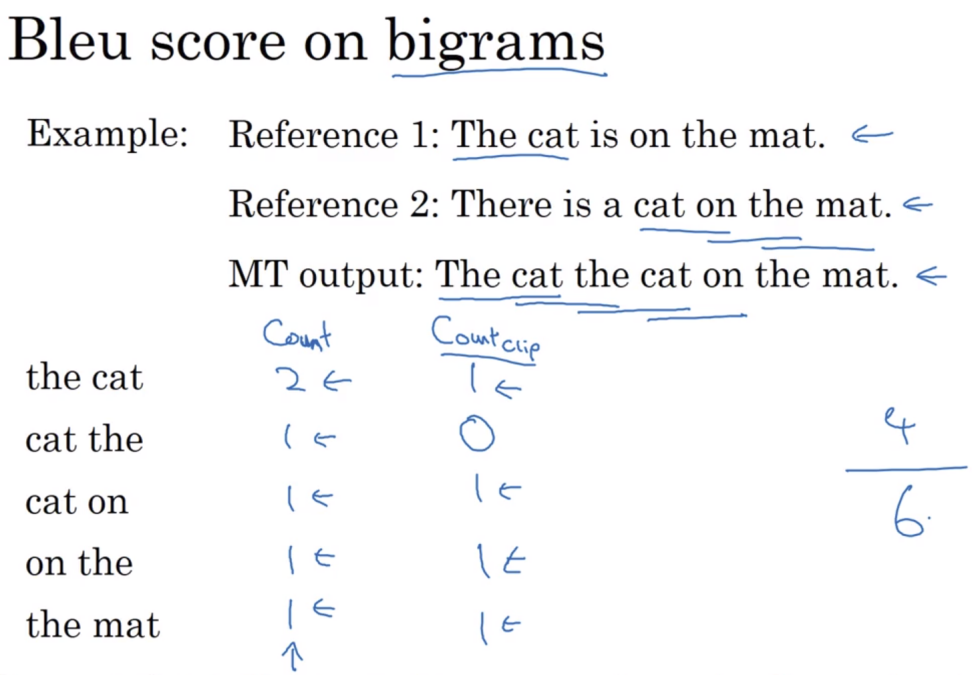


Figure 1.17: An example of Bleu score on bigrams. Resulting score is 4/6. **Count** means number of bigrams from machine translation. **Count clip** means the number of maximum bigrams appearing in 1 from all references.

Model error: bias + variance.

- **Bias** is a learner's tendency to consistently learn the same wrong thing. It is a systematic error; the **average error** for **different training data**. Can be calculated.
- **Variance** if the tendency to learn random things irrespective of the real signal. It indicates how **sensitive** is model **to variously changing training subsets of data from the whole dataset**. Can be calculated.

The noise: a property of data itself. Can be calculated.

- k-NN is generally considered intolerant of noise; its similarity measures can be easily distorted by errors in attribute values, thus leading it to mis-classify a new instance on the basis of the wrong nearest neighbors.
- Contrary to k-NN, rule learners and most decision trees are considered resistant to noise because their pruning strategies avoid overfitting the data in general and noisy data in particular.

Loss function: usually a function defined on a data point, prediction and label, and measures the penalty.

Cost function: measures the accuracy of our hypothesis function.

- It is usually more general. It might be a sum of loss functions over your training set plus some model complexity penalty (regularization).
- We want to minimize difference between predicted and true value in the whole dataset (each data point), so **we want to minimize a cost function.**
- For example - **Squared error (cost) function** (one of the best candidate for cost functions in regression problems, see Definition 2.2 for Linear regression with 1 variable).
  - However, **Squared error function with a logistic unit has also some drawbacks.** If the desire output is 1, and actual output is 0.000000001, then there is almost no gradient for logistic unit to fix up the error. Slope is almost exactly horizontal. So it will take a very very long time to change the weights (regarding ANN).

    $E = \frac{1}{2}(y-t)^2$, where $y = \sigma(z) = \frac{1}{1+exp(-z)}$ derivatives tend to "plateau-out" when y is close to 0 or 1: $\frac{dz}{dE} = (y-t) * y * (1-y)$.
  - Also, if we are trying to assign probabilities to mutually exclusive class labels, we know that the outputs should sum to 1, but we are depriving the network of this knowledge (for example, if we know that the probability of this is A is 3/4, and the probability that it's a B is also 3/4 is just crazy answer).
  - For classification (not for regression), there is an alternative - force the outputs to represent a probability distribution across discrete alternatives (classes).
- Cost functions can be visualized using a contour plots[25]. We can easily see on these plots a global minimum (minimal cost function).

Objective function: is the most general term for any function that you optimize during training - it is basically a mathematical expression we are trying to minimize or maximize. **A loss function is a part of a cost function which is a type of an objective function. All model-based learning algorithms have a loss function and what we do to find the best model is we try to minimize the objective known as cost function.**

Decision boundary: a boundary between 2 groups, it is some function. It is an ability of hypothesis and not a dataset!

Learning algorithm: again, our goal is to **find or approximate the target function**, and the learning algorithm is a set of instructions that tries to model the target

[25] A contour plot is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. In other words, cost function for different parameters has the same results for these parameters even they are different, and we can see them on the same line in a countour plot.

function using our training dataset. A learning algorithm comes with a hypothesis space, the set of possible hypotheses it can come up with in order to model the unknown target function by formulating the final hypothesis.

	Decision Trees	Neural Networks	Naïve Bayes	kNN	SVM	Rule-learners
Accuracy in general	**	***	*	**	****	**
Speed of learning with respect to number of attributes and the number of instances	***	*	****	****	*	**
Speed of classification	****	****	****	*	****	****
Tolerance to missing values	***	*	****	*	**	**
Tolerance to irrelevant attributes	***	*	**	**	****	**
Tolerance to redundant attributes	**	**	*	**	***	**
Tolerance to highly interdependent attributes (e.g. parity problems)	**	***	*	*	***	**
Dealing with discrete/binary/continuous attributes	****	***(not discrete)	***(not continuous)	***(not directly discrete)	**(not discrete)	***(not directly continuous)
Tolerance to noise	**	**	***	*	**	*
Dealing with danger of overfitting	**	*	***	***	**	**
Attempts for incremental learning	**	***	****	****	**	*
Explanation ability/transparency of knowledge/classifications	****	*	****	**	*	****
Model parameter handling	***	*	****	***	*	***

Figure 1.18: Comparison of learning algorithms (**** stars represent the best and * star the worst performance). From [7].

Genetic Algorithms

- They "sort of" belong to machine learning, because these are numerical optimization technique, that are used to optimize non-differentiable optimization objective functions. They use concepts from evolutionary biology to search for a global optimum (minimum or maximum) of an optimization problem, by mimicking evolutionary biological processes.
- Genetic algorithms work by starting with an initial generation of candidate solutions. If we look for optimal values of the parameters of our model, we first randomly generate multiple combinations of parameter values. We then test each combination of parameter values against the objective function. Imagine each combination of parameter values as a point in a multi-dimensional space. We then

generate a subsequent generation of points from the previous generation by applying such concepts as **selection**, **crossover**, and **mutation**. In a nutshell, that results in each new generation keeping more points similar to those points from the previous generation that performed the best against the objective. In the new generation, the points that performed the worst in the previous generation are replaced by "mutations" and "crossovers" of the points that performed the best.

- A **mutation** of a point is obtained by a random distortion of some attributes of the original point.
- A **crossover** is a certain combination of several points (for example, an average).
- Genetic algorithms allow finding solutions to any measurable optimization criteria. For example, GA can be used to optimize the hyperparameters of a learning algorithm. They are typically much slower than gradient-based optimization techniques or some other methods and they are not often used as components of machine learning algorithms - but there exists algorithms that incorporates them into neural networks for instance.

Multi-Label Classification

- In some situation, more than one label is appropriate to describe an example from the dataset. For example, if we want to describe an image, we could assign several labels to it: "road", "mountain", "car" at the same time.
- We can transform each labeled example into several labeled examples, one per label. These new examples all have the same feature vector and only one label. That becomes a multi-class classification problem. The only difference with the usual multi-class problem is that now we have a new hyperparameter - threshold. If the prediction score for some label is above the threshold, this label is predicted for the input feature vector. In this scenario, multiple labels can be predicted for one feature vector.
- Neural networks can naturally train multi-label classification models by using binary cross-entropy cost function.

Learning consists of the following 3 components:

- **Representation** - a classifier must be represented in some formal language that the computer can handle. Representable != learnable (eg. many local optimas in hypothesis space, learner may not find the true function even if it's representable).

  Examples:

  – Instances (k-nearest neighbor, SVM).

  - Hyperplanes (Naive Bayes, Logistic regression) form a linear combination of the features per class and predict the class with the highest-valued combination.
  - Decision trees (CART) test 1 feature at each internal node, with 1 branch for each feature value, and have class predictions at the leaves.
  - Set of rules (Propositional rules, Logic programs).
  - Neural networks.
  - Graphical models (Bayesian networks, Conditional random fields).
- **Evaluation** - an evaluation function (also called objective or scoring function) is needed to distinguish good classifiers from bad ones. The evaluation function used internally by the algorithm may differ from the external one, that we want the classifier to optimize.

  Examples:
  - Accuracy, error rate, and Squared error.
  - Precision and recall.
  - Likelihood.
  - Posterior probability.
  - Information gain.
  - K-L divergence.
  - Cost/utility.
  - Margin.
- **Optimization** - a methods to search among the classifiers in the language for the highest scoring one. The choice of optimization technique is key to the efficiency of the learner, and also helps determine the classifier produced if he evaluation function has more than 1 optimum.

  Examples:
  - Combinatorial optimization (Greedy search, Beam search, Branch-and-bound).
  - Continuous optimization, which can be constrained (Linear or Quadratic programming), or unconstrained (Gradient descent, Conjugate gradient, Quasi-Newton methods).

Building blocks of a learning algorithm

- **Loss function.**
- **Cost function** - an optimization criterion based on the loss function.

- **Optimization routine** leveraging training data **to find a solution to the optimization criterion** (e.g. gradient descent is the most frequently used optimization algorithm in cases where the optimization criterion is differentiable).

Covariate shift: if you train your model with data with some distribution, and then you will try the model with data from different distribution. Maybe both data could have the same decision boundary and there could exist a model that would perform very well on both, but a learning algorithm cannot find that decision boundary from a training data.

Edge deployment - you put a processor right where the data is collected so that you can process the data and make a decision very quickly without needing to transmit the data over the Internet to be processed somewhere else (for example, self-driving cars). Other types of deployments of AI projects are on **cloud** (if you rent compute servers such as from Amazon's AWS, or Microsoft's Azure, or Google's GCP in order to use someone else's service to do your computation), or **on-premises** (buying your own compute servers and running the service locally in your own company).

Corpus: terminology from NLP, which means basically a set of large body or a very large set of text of human-language (mostly English?) sentences. For NLP problems, first it is needed to tokenize the sentences in corpus (using vocabulary), and then map each word to one-hot vector. Also, a common thing is to model when a given sentence ends (adding extra token <EOS> - end of sentence). Words outside of your dictionary will be replaced by token <UNK> = unknown word.

Kolmogorov Complexity: concept in the field of Information Theory. It says, that the **complexity of a learned function is the length of the shortest computer program that can produce that function**. This theoretical concept has found few practical applications in AI as well.

Hyperparameters are properties of a learning algorithm, usually (but not always) having a numerical value. A value of a hyperparameter influences the way the algorithm works. Hyperparameters aren't learned by the algorithm itself from data. They have to be set by the data analyst before running the algorithm.

Parameters are variables that define the model learned by the learning algorithm. Parameters are directly modified by the learning algorithm based on the training data. The goal of learning is to find such values of parameters that make the model optimal in a certain sense.

Probability

- **Random variable** (usually written as italic capital letter, like X), is a variable whose possible values are numerical outcomes of a random phenomenon. There are 2 types of random variables - **discrete** (e.g. toss a coin - it takes only a countable

number of distinct values), and **continuous** (e.g. height of the first stranger you meet outside).

- **Probability distribution** of
  - A discrete random variable is described by a list of probabilities associated with each of its possible values. This list of probabilities is called a **probability mass function**. For example, $P(X = red) = 0.3$, $P(X = yellow) = 0.45$, $P(X = blue) = 0.25$. Each probability in a probability mass function is a value greater than or equal to 0. The sum of probabilities equals 1.
  - A continuous random variable takes an infinite number of possible values in some interval. Because the number of values of a continuous random variable X is infinite, the probability $P(X = c)$ for any c is 0. Therefore, instead of the list of probabilities, the probability distribution of a continuous random variable is described by a **probability density function**. It is a function whose co-domain[26] is non-negative and the area under the curve is equal to 1.

- **Expectation of a discrete random variable** X, that have k possible values $\{x_i\}_{i=1}^{k}$ is denoted as $E[X]$ and is given by:

$$E[X] = \sum_{i=1}^{k}[x_i.P(X = x_i)] = x_1.P(X = x_1) + x_2.P(X = x_2) + ... + x_k.P(X = x_k) \tag{1.13}$$

  where $P(X = x_i)$is the probability that X has the value x_i according to the probability mass function. The expectation of a random variable is also called the **mean**, **average**, or **expected value**, and is frequently denoted with the letter μ. It is one of the most important statistics of a random variable.

- **Expectation of a continuous random variable** X is given by

$$E[X] = \int_R x f_X(x) dx \tag{1.14}$$

  where f_x is the probability density function of the variable X and $\int_R$is the integral of function xf_X. The property of the probability density function that the area under its curve is 1 means that $\int_R f_X(x)dx = 1$. Most of the time we don't know f_X, but we can observe some values of X (these are called examples, and its collection is called dataset).

[26] Funtion is a relation that associates each element x of a set X, the domain of the function, to a single element y of another set Y, the codomain of the function.

- **Standard deviation** is defined as $\sigma = \sqrt{E[(X-\mu)^2]}$ and **variance** is $\sigma^2 = E[(X-\mu)^2]$.
  - It is a measure how (un)certain your decision is. It lies between 0 and 1, where 0 means impossible and 1 means certain.[27]

- **Unbiased estimators**:
  - If probability density function of the variable X is unknown (which usually is), but we have a dataset $S_X = \{x_i\}_{i=1}^N$, we are often not satisfied with true values of statistics of the probability distribution, such as expectation, but with their unbiased estimators. We say that $\widehat{\theta}(S_X)$ is an unbiased estimator of some statistic θ calculated using a dataset $S_X$drawn from an unknown probability distribution, if $\widehat{\theta}(S_X)$ has the following property: $E[\widehat{\theta}(S_X)] = \theta$, where $\widehat{\theta}$ is a sample (dataset) statistic, obtained using a dataset $S_X$and not the real statistic θ that can be obtained only knowing X; the expectation is taken over all possible samples drawn from X. This means that if you have an unlimited number of such samples as S_X, and you compute some unbiased estimator, such as $\widehat{\mu}$ using each sample, then the average of all these $\hat{\mu}$ equals to the real statistic μ that you would get computed on X. Unbiased estimator of an unknown $E[X]$ is given by $\frac{1}{N}\sum_{i=1}^N x_i$ which is called sample mean.
  - In statistics, the bias (or bias function) of an estimator is the **difference between this estimator's expected value and the true value of the parameter being estimated. An estimator with zero bias is called unbiased.** Otherwise the estimator is said to be biased.

- **Marginal probability**: probability of event A regardless whether event B occurred or not.

- **Conditional probability** is a probability of one random variable given an another random variable has a particular value. Formally, we say "the conditional probability of X given Y is the probability of event X when event Y is known".

$$p(X|Y=y) = \frac{p(X, Y=y)}{p(Y=y)} \tag{1.15}$$

  - where $p(X, Y)$ is a probability, that event X and Y occur at once.

- **Posterior probability** is just the conditional probability that is outputted by the Bayes theorem.[28] So it is assigned after the relevant evidence or background is taken into account.

- **Normalization constraint** is the probability theory's law that all probability values of any given random variable must add up to 1.

[27] https://medium.com/@laumannfelix/statistics-probability-fundamentals-1-1325ef72f3f
[28] https://stats.stackexchange.com/questions/347526/posterior-vs-conditional-probability

- **Probability distribution** is a function which gives the probability for every possible value of a random variable. Can be discrete or continuous (depends on values).
- **Probability mass function** is a function of a **discrete random variable**, whose **sum across an interval** gives the probability that the value of the variable lies within the same interval. It gives the probability that a discrete random variable is exactly equal to some value. We must define it for each possible value.
- **Probability density function** (pdf) a function of a **continuous random variable**, whose **integral across an interval** gives the probability that the value of the variable lies within the same interval. For modeling the pdf of unknown probability distribution from which some dataset has been drawn, we can use density estimation (see Section **??**).
- **Expected value** is the long-run average value of repetitions of the experiment it represents. We weigh each observation x with its probability p:

$$E[X] = x_1p_1 + x_2p_2 + ... + x_kp_k \tag{1.16}$$

  If you don't know the probability of each observation, calculate the arithmetic mean and you'll see that they are exactly the same.
- **Independence**: random variables are independent if knowing about X tells us nothing about Y. They are independent if and only if: a) $p(Y|X) = p(Y)$, and $p(X,Y) = p(X).p(Y)$. To prove independence, there is an algorithm called **D-Separation**.
- **Marginal independence:** Random variable X is conditionally independent from random variable Y if for all $x_i \in dom(X)$, $y_j \in dom(Y)$, $y_k \in dom(Y)$, then $P(X = x_i|Y = y_j) = P(X = x_i|Y = y_k) = P(X = x_i)$. So, knowledge of Y value doesn't affect your belief in the value of X.[29]
- **Conditional independence**: a random variable X is conditionally independent from Y given Z: $X \coprod Y|Z = p(X,Y|Z) = p(X|Z).p(Y|Z)$, which can be seen in the following diagram:

[29] https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202006-7/Lectures/lect25.pdf

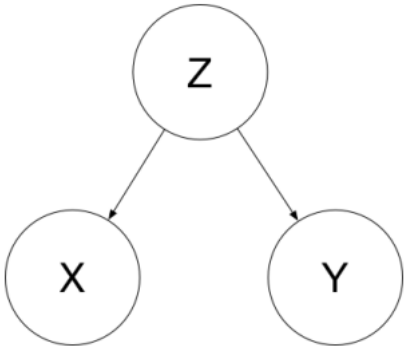


Figure 1.19: Conditional independence example. As soon as we know something about Z, we don't need any information about Y to know something about X, and we don't need any information about X to know something about Y.

  - Another definition of conditional independence: Random variable X is conditionally independent from random variable Y given random variable Z if for all $x_i \in dom(X)$, $y_j \in dom(Y)$, $y_k \in dom(Y)$, and $z_m \in dom(Z)$, then $P(X = x_i|Y = y_j \wedge Z = z_m) = P(X = x_i|Y = y_k \wedge Z = z_m) = P(X = x_i|Z = z_m)$. So, knowledge of Y value doesn't affect your belief in the value of X given a value of Z. Sometimes, two random variables might not be marginally independent. However, they can become independent after we observe some third variable.
  - Conditional independence means for graph (A->B->C) that observing A would not be influential at all for C, if we observed B. To express this more sophisticated: C is conditional independent of A given B.

- **Maximum Likelihood Estimation** (MLE) is an algorithm often used for learning parameters of statistical model. So it is a technique used for estimating the parameters of a given distribution, using some observed data. **Expectation-Maximization** (EM) is algorithm for finding out MLE parameters of stochastic models, where a model is depending on hidden relationships between variables, that contain hidden variables. EM is trying to optimize likelihood. Likelihood is a probability of data in statistical model given by parameters. Likelihood is defined as a set of parameter values of a model for input variables, which equals to probability of occurrence of input variables given parameters of model. **Likelihood function** is a measure how well the data summarizes the parameters of our model, i.e. our probability distribution. Formal definition of MLE:[30]
  - Let $x_1, x_2, ..., x_n$ be observations from n independent and identically distributed random variables drawn from a probability distribution f_0, where $f_0$is known to be from a family of distributions f that depend on some parameters θ.

[30] https://brilliant.org/wiki/maximum-likelihood-estimation-mle

  - For example, $f_0$could be known to be from the family of normal distributions f, which depend on parameters σ (standard deviation) and μ (mean), and $x_1, x_2, ..., x_n$ would be observations from f_0.
  - The goal of MLE is to maximize the likelihood function:
    $L = f(x1, x2, ..., xn|\theta) = f(x1|\theta) \times f(x2|\theta) \times ... \times f(xn|\theta)$
  - Often, the average log-likelihood function is easier to work with:
    $\hat{l} = \frac{1}{n} log(L) = \frac{1}{n} \sum_{i=1}^{n} log(f(x_i|\theta))$

- There are several ways that MLE could end up working: it could discover parameters θ in terms of the given observations, it could discover multiple parameters that maximize the likelihood function, it could discover that there is no maximum, or it could even discover that there is no closed form to the maximum and numerical analysis is necessary to find an MLE.

- Though MLEs are not necessarily optimal (in the sense that there are other estimation algorithms that can achieve better results), it has several attractive properties, the most important of which is consistency: a sequence of MLEs (on an increasing number of observations) will converge to the true value of the parameters.

- **Inference**: the process of computing probability distributions over certain specified random variables, usually after observing the value of some other variables in the model.
  - Example: you got model A->B->C, we observe B, but don't know A and C. A does not matter at all for calculating C, if we observe B. How we can calculate probability of C? Exactly with $P(C|B)$ and this is inference.

Gaussian Process Models: these models (GP models) assume that similar inputs give similar outputs. This is very weak but very sensible prior for the effects of hyper-parameters. For each input dimension, they learn the appropriate scale for measuring similarity. These models predict a Gaussian distribution of values. If you have the resources to run a lot of experiments, Bayesian optimization is much better than a person at finding good combination of hyper-parameters.

Random variable: a named quantity (=variable) whose value is uncertain. Consequently, we can only give a probability that this quantity has a given value.

Dummy variable: (also known as an indicator variable, design variable, one-hot encoding, Boolean indicator, binary variable, or qualitative variable) are used as devices to sort data into mutually exclusive categories (such as smoker/non-smoker, etc.). Dummy variable takes the value 0 or 1 to indicate the absence or presence of some categorical effect.

- One-Hot Encoding is especially useful when we want, for some reason, to transform categorical values into several binary ones (the reason can be, that some learning

algorithms only work with numerical feature vectors, and we would have vector of binary numerical values as a result). So if we have for example a feature that has 3 possible values: red, yellow, and green, we could transform them into $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$, but not to 1, 2, and 3, because that would mean that there is an order among the values, and this specific order is important for the decision making (yes, we would avoid the dimensionality, but we cannot do that). If the order of a features' values is not important, using ordered numbers as values is likely to confuse a learning algorithm, because the algorithm will try to find a regularity where there's no one, which may potentially lead to overfitting.

- On the other hand, when the ordering of values of some categorical variable matters, we can replace those values by numbers by keeping only one variable. For example if we have feature with 3 values: poor, decent, excellent, when we can replace them by 1, 2, and 3.

Median: the value separating the higher half of a data sample / population / probability distribution, from the lower half.

Mode: the value that appears the most often.

Variance: a measure how much the values vary around the mean.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - m)^2 \tag{1.17}$$

Standard deviation: the square root of the variance, abbreviated as σ.

Likelihood: it works with discrete values, SUM of all likelihoods of some observations is INF. "L likelihood that this comes from this category". It is conditional probability. The unknown (variable) is an assumption. It has inductive character.

Entropy:

- It is average rate at which information is produced by a stochastic source of data.
- In other words, entropy (Shannon entropy) measures the amount (or ratio) of information in data. The higher value of entropy we have, the more ambiguity is in our data and information value is lower. So, lower entropy means that we can better predict type of record in our data.

$$H(X) = E[I(X)] = -\sum_{\forall x \in X} p(x)I(x) = -\sum_{\forall x \in X} p(x) log\, p(x) \tag{1.18}$$

- **Example**: if all records in our dataset are yellow cards, then entropy is 0 (if our classification attribute is color of car), since we can predict color of a car.

Occams Razor:

This tells us, that simpler models are more probable than complex ones. ANN can overfits, because a model is more complex. So, by computation of the most probable model we get the simplest possible model, and this model is the lowest prone to be overfitting.

- An example: we have a sequence of numbers: $-1, 3, 7, 11$. What is the next number?
  - Let's state the first hypothesis $H_1 = x_{i+1} = x_i + 4$. This makes sense, the next number should b e 15.
  - Let's state the second hypothesis: $H_2 = x_{i+1} = \frac{-1}{11} * x_i^3 + \frac{9}{11} * x_i^2 + \frac{23}{11}$. This is more complicated case, but still correct and the next number should be -219/11.
  - What is more probable hypothesis? Occams Razor will prefer the first one, because is more simple. For computation, Occams Razor uses (for hypothesis comparison) the amount of data, that can occur by the hypotheses. For estimating probabilities of our hypotheses, we use Bayesian Theorem: $\frac{P(H_1|D)}{P(H_2|D)} = \frac{P(H_1)}{P(H_2)} \cdot \frac{P(D|H_1)}{P(D|H_2)}$
  - Let's consider, that we don't have any initial thoughts and clues about probabilities of each hypotheses. That means, that the first part of the equation is the following: $\frac{P(H_1)}{P(H_2)} = 1$. For avoiding infinity and so on, let's consider interval between $[-50, 50]$.
  - For the first case, $P(D|H_1)$: $H_1$is a linear function. The first number of sequence as well as the constant (+4 in our equation) can both have 101 possibilities (see interval). This means, that they both can have 101 possibilities: $P(D|H_1) = \frac{1}{101} \cdot \frac{1}{101} = 10^{-4}$.
  - For the second case, $P(D|H_2)$: H_2 can be calculated as the first value in the sequence (1/101) and with 3 fractions. The first number -1/11 can be expressed by 4 possibilities: $-1/11 \bigvee -2/22 \bigvee -3/33 \bigvee -4/44$. Similarly for number 9/11, we can have 4 options. For number 23/11, with only 2 options (because we have restriction to 50, see our interval): $23/11 \bigvee 46/22$. As a result, $P(D|H_2) = \frac{1}{101} \cdot (\frac{4}{101} \cdot \frac{1}{50}) \cdot (\frac{4}{101} \cdot \frac{1}{50}) \cdot (\frac{2}{101} \cdot \frac{1}{50}) \cdot (\frac{2}{101} \cdot \frac{1}{50}) = 2.5.10^{-12}$.
  - As a result, $\frac{P(H_1|D)}{P(H_2|D)} = \frac{10^{-4}}{2.5.10^{-12}} = 4.10^7$and therefore the first hypothesis is preferred in ratio by almost 40,000,000:1. And if we did not restrict possible values into interval of 101 values, this win would be even bigger.

Exponentially weighted averages

- Averaging over the last n samples (this is just an approximation, where $n \approx \frac{1}{1-\beta}$ - so the bigger β is, the more previous data are taken into account; for example, if $\beta = 0.5$, then only 2 samples are taken into account, if $\beta = 0.98$, then 50 samples;

and thus β is another hyperparameter to optimize). The formula is following, where $\theta_t$is an input feature (for example):

$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t \tag{1.19}$$

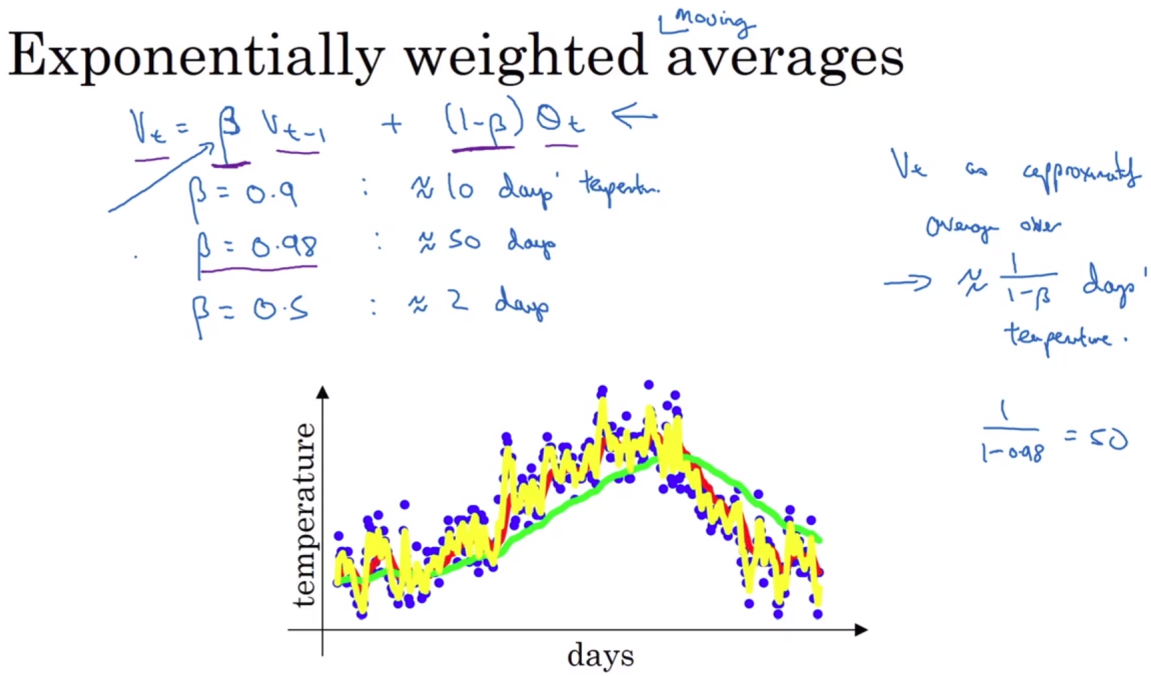


Figure 1.20: An example of exponentially weighted averages on temperature data for a year. Yellow curve is when $\beta = 0.5$ (averaging cca through 2 days, so it adapts quickly), red is when $\beta = 0.9$ (averaging cca over the last 10 days), and green line is $\beta = 0.98$ (equal to cca 50 previous days). β is a hyperparameter to estimate. It would be good also to perform bias correction (final value is divided by $1 - \beta^{iteration}$ where *iteration* is a current iteration number), but people don't often bother. This is a problem when the algorithm starts (curve starts off really low) and is still warming (because we take data (gradients for example) from the previous iterations, but at the beginning there are no values. Increasing β will shift the line slightly to the right, and decreasing it will create more oscillation within the line.

- However, it is needed to perform bias correction. The problem arises from initializing $V_0 = 0$, which causes $V_1 = 0.98V_0 + (1 - 0.98)\theta_1 = 0 + 0.02\theta_1$, which is only 2% of θ_1 - a very poor estimate of the first data sample. And if you would compute V_2, then it would be much less then either θ_1 or θ_2 and it is not a good estimate for the first two samples.

- A better approach is to use:

$$V_t = \frac{V_t}{1 - \beta^t} \tag{1.20}$$

Especially for small values of t - when this algorithm is still "warming up". Not everyone uses this approach, but it is useful. As t increases, so $1 - \beta^t \to 1$, then the bias correction effect is no longer making a difference on the exponentially weighted average output.

Covariance: how are 2 quantities (or parameters) **similar to each other** regarding their **average values** (or means) = joint variability of 2 random variables.

$$s_{xy} = \frac{\sum(x - \bar{x})(y - \bar{y})}{n - 1} \tag{1.21}$$

Correlation: standardize (normalize) covariance (so divided by multiplication of both standard deviations), **the rate of linear dependency**. It ranges from -1 to 1. You should normalize your data before you compute the correlation between 2 random variables.

$$r = \frac{s_{xy}}{s_x s_y} = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2} * \sqrt{(y - \bar{y})^2}} \tag{1.22}$$

Causation: causation exists between 2 random variables, if changes in one is the reason of changes in the other. "Correlation Does Not Imply Causation".[31]

Example: imagine it is 23th May in Copenhagen, around 22 degrees of Celsium. Many people are eating ice-cream and you ask if that's because it is hot, or local people just like ice-cream. After a few more days in city, temperature ranging from 10-25 Celsiums, you guess that it is because of the high temperature. So you drawn a causal relation between 2 random variables "temperature" and "eating ice-cream". But you could also observe that people go shopping a lot in the warm days, more than on the colder days you spent in the city. Does this also imply a causal relation? You cannot be certain, because there could simply has been public holidays on the cold days, and most stores were closed, so less people were attracted to go shopping.

Regression: also standardize covariance, but it is calculated differently via influences of variables, not standard deviations.

$$r_x = \frac{s_{xy}}{s_x s_x} = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2} \tag{1.23}$$

[31] https://medium.com/@laumannfelix/statistics-probability-fundamentals-2-cbb1239f9605

Regularization: is a process of introducing additional information in order to prevent overfitting.

- These algorithms are basically an extension made to another method (typically regression methods) that penalizes models based on their complexity, favoring simpler models that are also better at generalizing.
- It makes hypothesis "simpler". We usually add regularization to cost function (so it results in adding regularization in gradient descent; however it is possible to use regularization also in normal equation).
- Keep all the features, but reduce magnitude/values of parameters .
- Works well when we have a lot of features, each of which contributes a bit to predicting .
- Penalization of some thetas, so they are very small (close to 0).
- **Too much regularization** - we can **under-fit** the training set and this can lead to worse performance even for examples not in the training set.
- But we do not know, what thetas we should shrink, so we use regularization on all, we practically use regularization in cost function itself (inside) with
  - regularization term (sum over all thetas)
  - regularization parameter λ. This controls trade-off between a) fit the training data well b) keep the parameters small. If λ is too big, then we end up penalizing all thetas very highly, so our hypothesis is causing underfitting. It can be automatically chosen by multi-selection algorithms
    - Explanation: if we chose very big λ, then to get cost function close to 0, thetas have to be very small (if they are big, we have very big output of cost function - big penalization).
- **Weight decay** - a regularization technique (also known as L2 regularization) that results in gradient descent shrinking the weights on every iteration.[32]

  See L2 regularization, update rule for weight on some layer: $w = w - \alpha(\frac{\partial J}{\partial w} + \frac{\lambda}{m}w - \alpha\frac{\partial J}{\partial w}) = w - \alpha\frac{\lambda}{m}w - \alpha\frac{\partial J}{\partial w} = w(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{\partial J}{\partial w}$, where term $1 - \alpha\frac{\lambda}{m}$ is weight decay - because on every update of a given weight, we decrease a given weight by $1 - \alpha\frac{\lambda}{m} < 1$ regardless of $\frac{\partial J}{\partial w}$.
- **L1 regularization** - similar to L2 regularization, penalization of large weights and tending to make the network prefer small weights. Of course, it is not the

[32] https://stats.stackexchange.com/questions/29130/difference-between-neural-net-weight-decay-and-learning-rate

same. L1 regularization is also known as **lasso**.

$$C = C_0 + \frac{\lambda}{n} \sum_w |w| \tag{1.24}$$

- Partial derivatives of the cost function is $\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} sign(w)$, where $sign(w)$ is the sign of w, that is ± 1. Update rule for L1 regularized network is $w \rightarrow w^{'} = w - \frac{\eta\lambda}{n} sign(w) - \eta \frac{\partial C_0}{\partial w}$ and we can estimate also this using mini-batch average (just a summed average over m training samples in mini-batch).
- If we compare this with resulting update rule for L2: $w \rightarrow w^{'} = w(1 - \frac{\eta\lambda}{n}) - \eta \frac{\partial C_0}{\partial w}$, we can see, that in both expressions the effect of regularization is to shrink the weights. In L1, however, the weights are shrinked by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. So, when a particular weight has a large magnitude, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when $|w|$ is small, L1 regularization shrinks the weight much more than L2 regularization. So, L1 regularization tends to concentrate he weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.
- By the way, for L1 regularization, $\partial C \,/\, \partial w$ is not defined when $w = 0$. The reason is that function $|w|$ has a sharp corner at $w = 0$, and so it isn't differentiable at that point. But it's okay. When $w = 0$, we will apply just usual unregularized rule for stochastic gradient descent. So intuitively, the effect of regularization is to shrink weights, and obviously it can't shrink a weight which is already 0. By the way, L2 regularization is fully differentiable.
- L1 regularization results in a **sparse** (a lot of zeros) weight vectors, which some people argue that it helps to compress the model because for the parameters that are zero we need less memory to store the model.

- **L2 regularization** (see ANN chapter) also known as **ridge regularization**.
- **Frobenius norm** is similar to L2, but often used more in practice. It can be calculated as:

$$||w^{[layer]}||_F^2 = \sum_i^{n^{[layer-1]}} \sum_j^{n^{[layer]}} (w_{ij}^{[layer]})^2 \tag{1.25}$$

where the indices of the summation are determined by dimensions of number of hidden units in layers $layer$ and $layer - 1$.

- **Dropout**

  - Widely used regularization technique that is specific to deep learning. It randomly shuts down some neurons in each iteration.
  - This is radically different technique for regularization. It does not rely on modifying the cost function, but the network itself. Input and output neurons are untouched, but neurons in all hidden layers are (with some chance) temporarily deleted (for instance, let's say half of them). Input is then forward propagated through such modified network, and then the result is backpropagated, also through the modified network. This can be done on the whole mini-batch of examples. Then, we can repeat the process, firstly restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.
  - Different networks may over-fit in different ways, and averaging of multiple different nets may help to eliminate overfitting. And basically, similar thing is going on with dropout. Dropout eliminates different sets of neurons, so it's like training different neural nets (so, it's something like ensemble learning). Dropout is like averaging the effects of a very large number of different networks. The idea was that different networks will over-fit in different ways, and so hopefully, the net effect of dropout will be to reduce overfitting.
  - Cost function J is no longer well defined when using dropout, because on any given iteration some units are zeroed out. So checking that J is going down with every iteration is impossible and thus we lost this debugging tool. As a workaround, if you turn off dropout by setting $keep\,prob = 1$ and check if J is decreasing monotonously. If yes, then turn on dropout and hope no bugs were introduced.

- **Weight sharing** - it prevents overfitting as well. It basically **makes a model** (ANN) **simpler** by insisting that many weights have exactly the same value as each other. You don't know what the value is, and you are going to learn it, but it has to be exactly the same for many of the weights.

- **Artificially expanding the training data** - another technique that can help to reduce overfitting. For example, if we have a pictures of handwritten digits (MNIST for example), we can rotate each image multiple times, always under different angle. And so on. In general, the idea is to expand training data by applying operations that reflect real-world variation.

Feature selection: family of algorithms that improves estimators' accuracy scores or boosts their performance on very high-dimensional datasets.

- On the other hand, **adding many new features** gives us more expressive models which are able to **better fit our training set**. If too many new features are added, this can lead to **overfitting of the training set**.

- Feature selection to decrease number/type of input features: This technique might help with variance problems, but it might also increase bias. Reducing the number of features slightly (say going from 1,000 features to 900) is unlikely to have a huge effect on bias. Reducing it significantly (say going from 1,000 features to 100 - a 10x reduction) is more likely to have a significant effect, so long as you are not excluding too many useful features. In modern deep learning, when data is plentiful, there has been a shift away from feature selection, and we are now more likely to give all the features we have to the algorithm and let the algorithm sort out which ones to use based on the data. But when your training set is small, feature selection can be very useful.

Feature scaling: sometimes necessary, sometimes good to have because models can converge faster (for example, see contour graphs on 2 variables with totally different ranges - graph is skewed and gradient descent must perform many steps, in comparison to more circular graph with much less steps[33]). Normalize the whole dataset at once! Do not normalize train and test data separately - use the same mean, standard deviation and so on. It is needed that all data goes through the same transformation. There exist different techniques:

- **Rescaling** (also known as Min-Max normalization) is the simplest method, and it puts values into the range of [0;1] or [-1;1]

$$x_i^{'} = \frac{x_i - min}{max - min} \tag{1.26}$$

where the denominator is basically the range of possible values.

- **Mean normalization**

$$x_i^{'} = \frac{x_i - \overline{x_i}}{max - min} \tag{1.27}$$

where $\overline{x}$ is mean and the denominator is basically the range of possible values.

- **Standardization** - widely used (SVM, Logistic regression, ANN)

$$x_i^{'} = \frac{x_i - \bar{x_i}}{\sigma_i} \tag{1.28}$$

where $\overline{x}$ is mean (also can be noted as μ in some literature) and σ is standard deviation. Standardization (z-score normalization) is a procedure during which

[33] Gradient Descent in Practice I - Feature Scaling
https://www.coursera.org/learn/machine-learning/lecture/xx3Da/gradient-descent-in-practice-i-feature-scaling

the feature values are rescaled so that they have properties of a standard normal distribution with $\mu = 0$ and $\sigma = 1$. Unsupervised learning algorithms usually benefit more from standardization than from normalization. It is also preferred for a feature if the values this feature takes, are distributed close to a normal distribution. Or, for a feature which may have sometimes very high or low values (outliers), because this will squeeze the normal values into a very small range. Usually, in other cases, normalization is preferable, but this is not strictly defined, rather one needs to experiment and try multiple approaches (this depends on a dataset size and training time).

Missing features

- Remove examples with missing features from dataset - if your dataset is big enough so that you can sacrifice some training examples.
- Some algorithms can deal with missing values and this depends on the implementation and/or modifications to it.
- Use data imputation technique. You can a lot of possible strategies, for example, replace the missing value of a feature by:
  - an average value of this feature in the dataset.
  - middle value of a given range.
  - with a value outside of typical range, so that the learning algorithm will learn what is best to do when the feature has a value significantly different from regular values.
  - consider it to be a target variable for a regression problem. You can use all remaining features to form a feature vector, build a regression model to predict it. Of course, to build training examples, you only use those examples from the original dataset, in which the value of missing feature is present.
  - add a new feature that will represent a binary flag whether a given potentially missing feature is present or not (and fill the missing feature with 0 or any number of your choice).

Error analysis:

- The process of looking at misclassified examples.
- Usually on dev set.
- Error analysis can often help you figure out how promising different directions are. Don't spend a month of effort on some approach which could lead that your model accuracy will increase just by 0.5%. Investigate what could be the outcome of an approach by error analysis (by manually examining misclassified examples).

- Evaluate multiple ideas in parallel during error analysis - gather all ideas, put it into spreadsheet and go through misclassified dev set samples, consider each idea, or even make notes on each dev sample (100 or a bit more samples are manageable).
- Suppose you have a large dev set of 5,000 examples in which you have a 20% error rate. Thus, your algorithm is misclassifying ~1,000 dev images. It takes a long time to manually examine 1,000 images, so we might decide not to use all of them in the error analysis. In this case, I would explicitly split the dev set into two subsets, one of which you look at (randomly pick 10% of all dev set data - 500 samples, about 20% of them will be misclassified - 100 samples; you manually examining these 500 samples - **eyeball dev set**), and one of which you don't (4500 samples - **blackbox dev set**). You will more rapidly over-fit the portion that you are manually looking at. You can use the portion you are not manually looking at to tune parameters.
  - Eyeball dev set manual examination of errors - use this information to prioritize what types of errors to work on fixing.
  - Why do we explicitly separate the dev set into Eyeball and Blackbox dev sets? Since you will gain intuition about the examples in the Eyeball dev set, you will start to over-fit the Eyeball dev set faster. If you see the performance on the Eyeball dev set improving much more rapidly than the performance on the Blackbox dev set, you have over-fit the Eyeball dev set. In this case, you might need to discard it and find a new Eyeball dev set by moving more examples from the Blackbox dev set into the Eyeball dev set or by acquiring new labeled data.
  - If you are working on a task that even humans cannot do well, then the exercise of examining an Eyeball dev set will not be as helpful because it is harder to figure out why the algorithm didn't classify an example correctly. In this case, you might omit having an Eyeball dev set.
  - If you have a small dev set, then you might not have enough data to split into Eyeball and Blackbox dev sets that are both large enough to serve their purposes. Instead, your entire dev set might have to be used as the Eyeball dev set—i.e., you would manually examine all the dev set data.
  - If you only have an Eyeball dev set, you can perform error analyses, model selection and hyperparameter tuning all on that set. The downside of having only an Eyeball dev set is that the risk of overfitting the dev set is greater.
  - If you have plentiful access to data, then the size of the Eyeball dev set would be determined mainly by how many examples you have time to manually analyze. For example, I've rarely seen anyone manually analyze more than 1,000 errors.
  - The Eyeball dev set should be big enough so that your algorithm mis-classifies enough examples for you to analyze. A Blackbox dev set of 1,000-10,000 examples is sufficient for many applications.

Recommender systems

- These systems seek to predict the "rating" or "preference" a user would give to an item.
- An example of one problem solved with recommender systems is on the following figure.

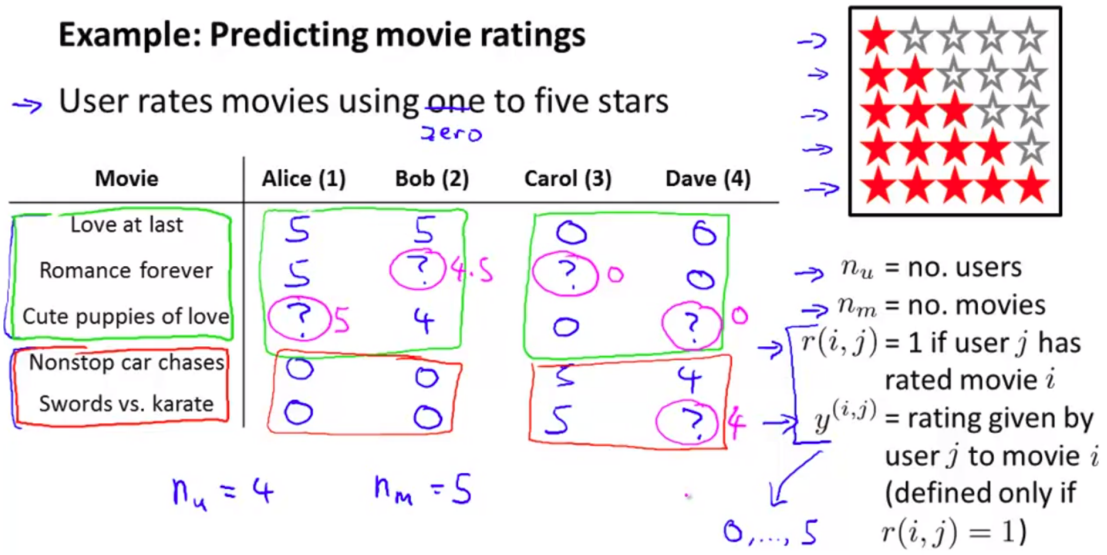


Figure 1.21: Formulation of predicting movie ratings problem, which can be implemented with recommender systems. Some users did not watch all the movies (purple), which is completely fine and realistic. In this example, Alice and Bob clearly likes romantic movies and do not like action movies (in opposite to Carol and Dave).

- Traditionally, two approaches were used to give recommendations:
  - **Content-based filtering** - how users would rate movies that they did not watch/rated yet. In order to make predictions, we could treat predicting ratings of a user as a separate linear regression problem, see figures below.
    - This method (in this case) is called content-based recommendation, because we assume that features for different movies are available to us. These features capture what is the content of these movies = how romantic or action is a given movie. So, for making predictions, we're using features of a movie content.
    - Another approach, that isn't content based and doesn't assume that we have someone else giving us all of these features for all of the movies in our data set, is called **collaborative filtering**.

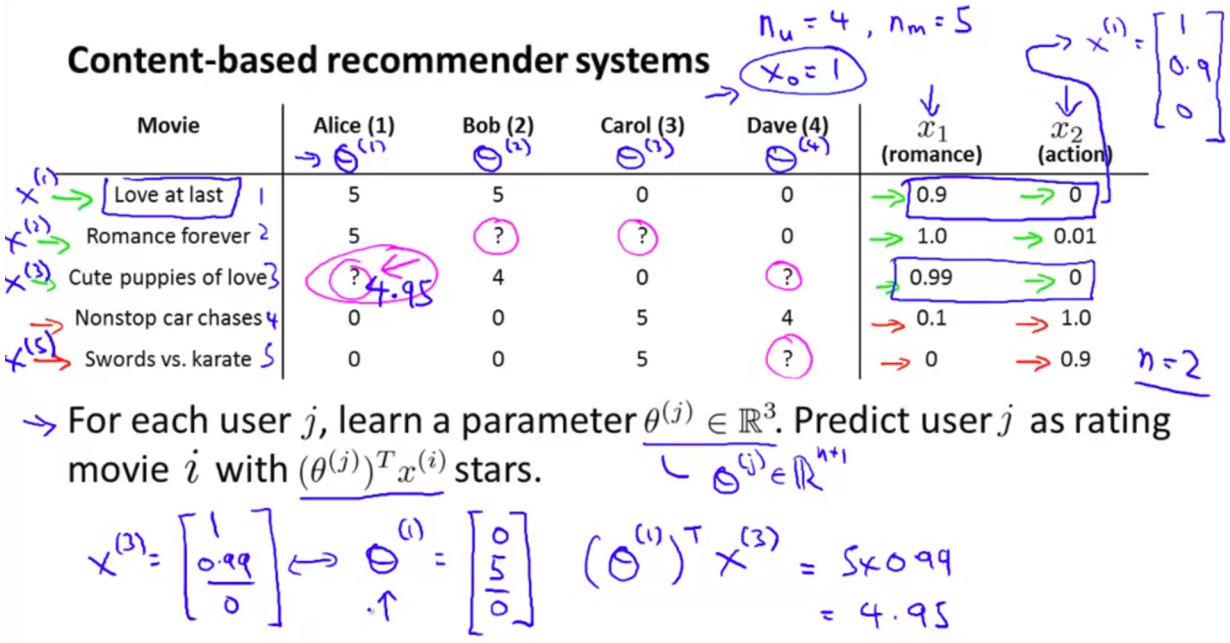


Figure 1.22: Content-based recommender systems, an example of problem with movie rating prediction using linear regression.

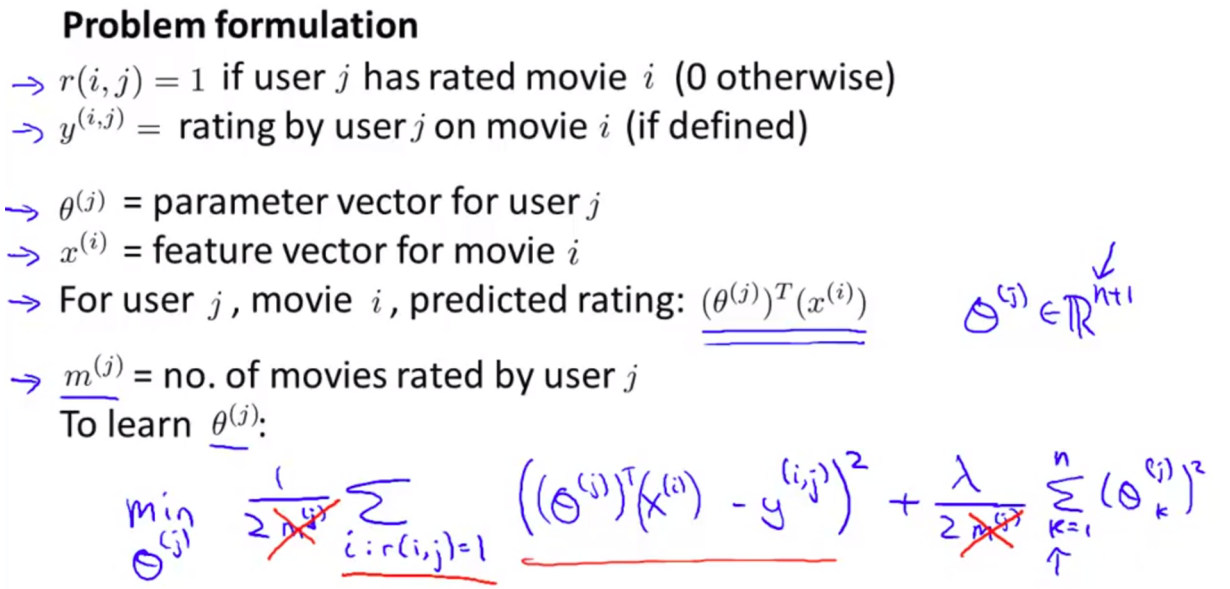


Figure 1.23: Content-based recommender system problem formulation using linear regression.

– **Collaborative filtering** - as mentioned in the previous paragraph, if we don't have any information about feature values (like, how romantic or action is a specific movie), we can compute them from data (for example users' data - users provided how they like romantic or action movies, and rated some movies, see a figure below).
  * So basically, users are collaborating and each user hopes that the algorithm will learn better features because of him, and perhaps the whole system (for example for movie prediction) will perform better.

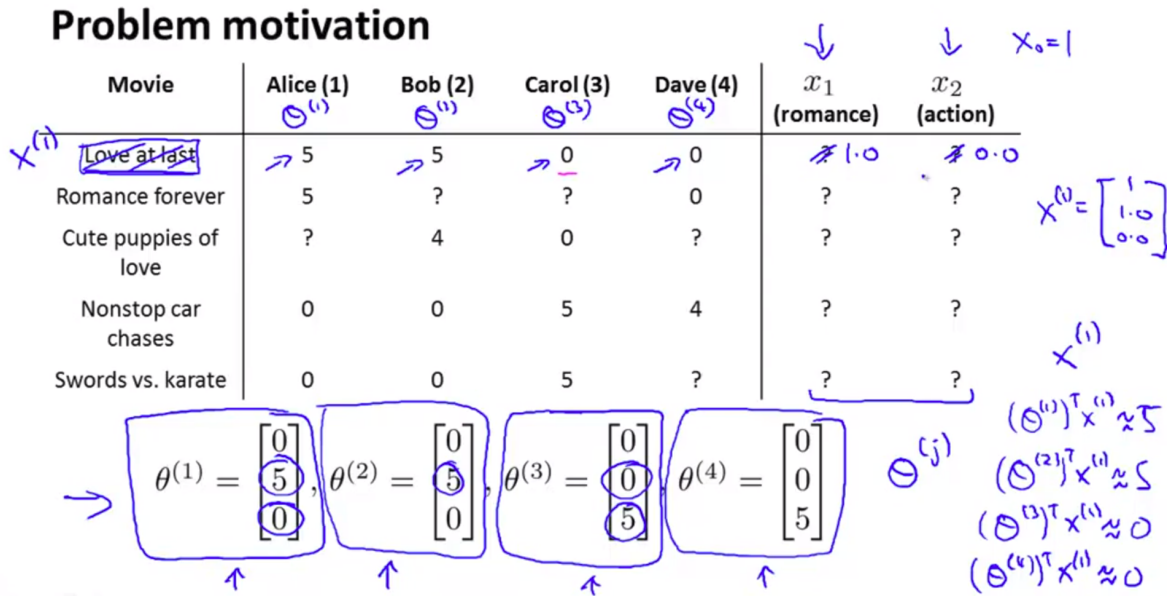


Figure 1.24: Formulation of predicting movie ratings problem, which can be implemented with recommender systems using collaborative filtering.

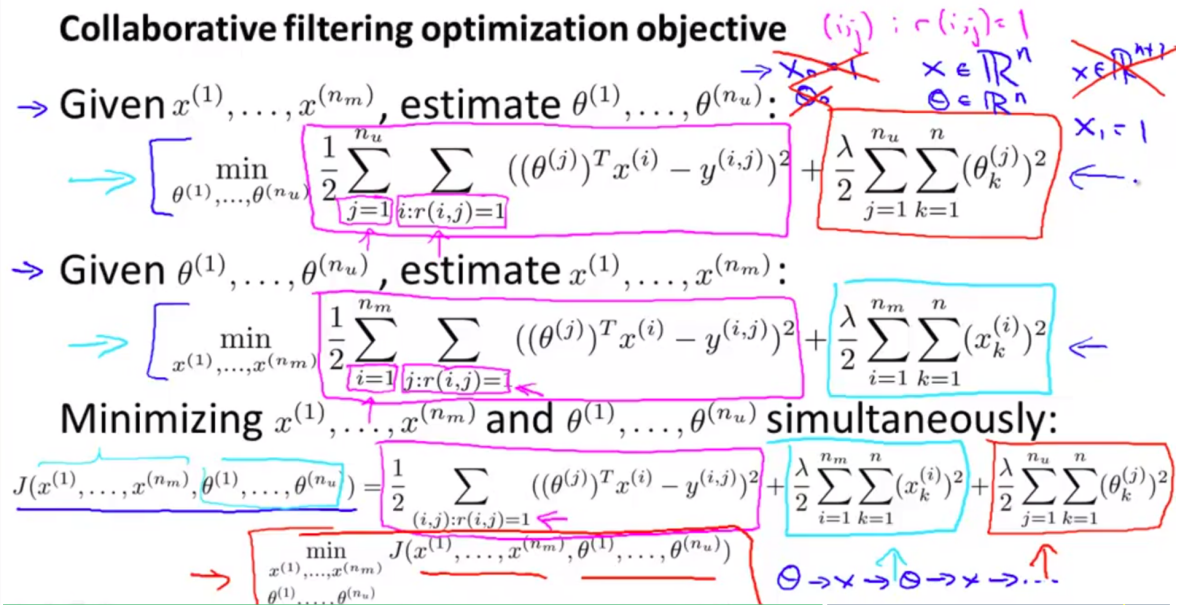


Figure 1.25: Collaborative filtering optimization objective. We want to minimize all x and all θ - so we want to find all parameters θ for our users, and we want to estimate features x. We could go back and forth - solve parameters, features, parameters, features, and so on - but much better, and more efficient approach is to solve them both simultaneously.

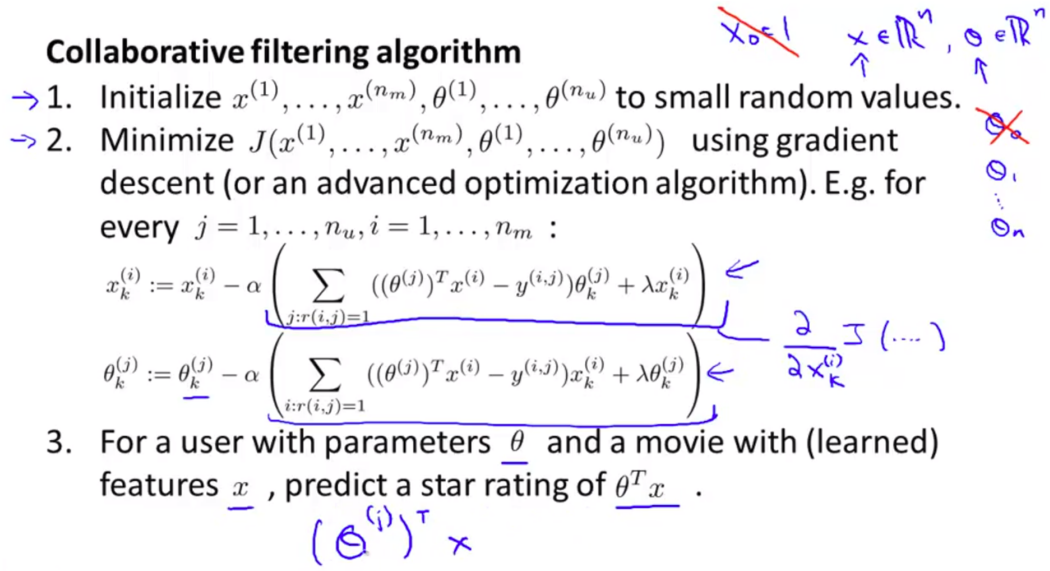


Figure 1.26: Collaborative filtering algorithm summary.

- Most real-world recommender systems use a hybrid approach: they combine recommendations obtained by the content-based and collaborative filtering models. Two effective recommender system learning algorithms are**:**
- **Factorization machines** (FM)
  - Relatively new kind of algorithm, explicitly designed for sparse datasets.
  - Its model is defined as follows:

$$f(x) = b + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D} \sum_{j=i+1}^{D} (v_i v_j) x_i x_j \tag{1.29}$$

    where b and w_i, $i = 1, ..., D$ are scalar parameters similar to those used in linear regression. Vectors v_i are k-dimensional vectors of **factors**. k is a hyperparameter and is usually much smaller than D. So, instead looking for one wide vector of parameters, which can reflect poorly the interactions between features because of sparsity, we complete it by additional parameters that apply to pairwise interactions $x_i x_j$ between features. However, instead of having a parameter $w_{i,j}$ for each interaction, which would add an enormous quantity of new parameters to the model, we factorize $w_{i,j}$ into $v_i v_j$ by adding only $Dk << D(D-1)$ parameters to the model.
  - Depending on the problem, the loss function could be squared error loss (for regression), or hinge loss. For classification with $y \in \{-1, +1\}$, with hinge loss or logistic loss the prediction is made as $y = sign(f(x))$.
  - Gradient descent can be used to optimize the average loss.
- **Denoising autoencoders** (DAE) see Section 195. The fact, that the input is corrupted by a noise, while the output shouldn't be, makes denoising autoencoders an ideal tool to build a recommender model. The idea is very straightforward - new movies that a user could like, are seen as removed from the complete set of preferred movies by some corruption process. The goal of the denoising autoencoder is to reconstruct those removed items.

1.5 Performance

We need to measure the performance of a given model somehow. There exist several metrics.

- For **regression problems**, the situation is quite simple. A well-fitting regression model results in predicted values close to the observed data values. The mean model, which always predicts the average of the labels in the training data, generally would be useful if there were no informative values. The fit of a regressor should be better than the fit of the mean model. So we can compare the performances of the model on the training and test data, for example with MSE (or other type of average loss function that makes sense) for training, and then separately, for test data. If the MSE of the model on the test data is substantially higher than MSE obtained on the training data, this is a sign of overfitting.
- For **classification problems**, we have a lot of options: confusion matrix, accuracy, cost-sensitive accuracy, precision/recall, or area under the ROC curve.

Basic terminology

- condition positive (P) - the number of real positive cases in the data
- condition negative (N) - the number of real negative cases in the data
- true positive (TP) - eqv. with **hit**
- true negative (TN) - eqv. with **correct rejection**
- false positive (FP) - eqv. with **false alarm**, **Type I error**
- false negative (FN) - eqv. with **miss**, **Type II error**

Accuracy - carefully, there can be Accuracy Paradox[34] when working with unbalanced dataset (skewed classes).

$$\frac{TruePositives + TrueNegatives}{TruePositives + TrueNegatives + FalsePositives + FalseNegatives} \tag{1.30}$$

which is basically

$$\frac{CorrectPredictions}{AllPredictions} \tag{1.31}$$

[34] Accuracy paradox - https://en.wikipedia.org/wiki/Accuracy_paradox

Precision - from all patients, ones we predicted to have cancer - how many truly has cancer? (**true positives / predicted positives**, see confusion matrix if you are confused :-)). Basically, how precise is what I recall?[35]

$$\frac{TruePositives}{TruePositives + FalsePositives} \tag{1.32}$$

Recall - (a.k.a. sensitivity, or probability of detection) from all patients for truly have cancer, how many of them we correctly detected that they have cancer? (**true positives / actual positives**, see confusion matrix if you are confused :-)). Measures how good classifier we have. It is always a balance between recall and precision. One single metric, between precision and recall is **F1-score**.

$$\frac{TruePositives}{TruePositives + FalseNegatives} \tag{1.33}$$

Trading off precision and recall

- For example, in logistic regression, if we predict '1' if $h_\theta(x) \geq 0.9$ (0 otherwise), then we tell someone to has cancer only when we are sure on at least 90%. This is higher precision, but lower recall. On other other side, if we predict '1' if $h_\theta(x) \geq 0.3$ (0 otherwise), we claim that someone has cancer even when we are not very certain (at least 30%) - higher recall, but lower precision. Depends on a final system.

[35] https://www.quora.com/What-is-the-best-way-to-understand-the-terms-precision-and-recall

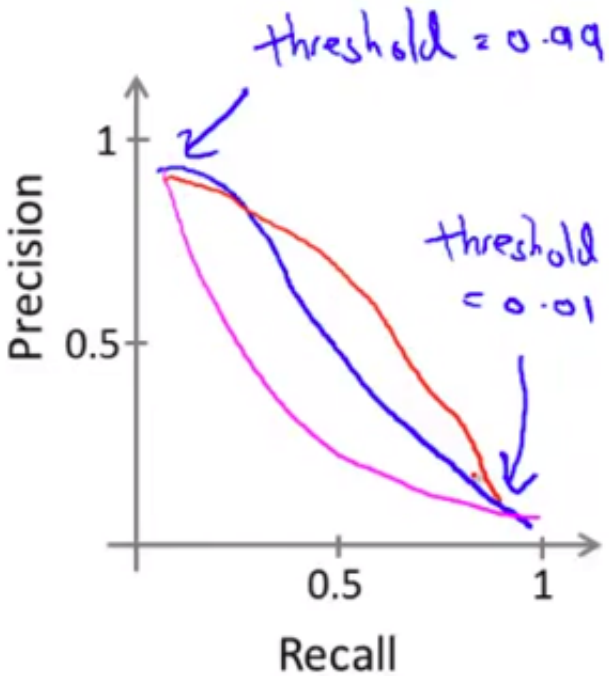


Figure 1.27: An example of plotting a tradeoff between precision a recall. The curve can have very different shapes, like on the graph (different colors).

- To understand the meaning and importance of precision and recall for the model assessment it is often useful to think about the prediction problem as the problem of research of documents in the database using a query. The precision is the proportion of relevant documents in the list of all returned documents. The recall is the ratio of the relevant documents returned by the search engine to the total number of the relevant documents that could have been returned.
- In the case of the spam detection problem, we want to have high precision (we want to avoid making mistakes by detecting that a legitimate message is spam), and we are ready to tolerate lower recall (some spam messages in our box).
- It is usually impossible to have both, so you can achieve either of the two by various approaches:
  - By assigning a higher weighting to the examples of a specific class.
  - By tuning hyperparameters to maximize precision or recall on the validation set.
  - By varying the decision threshold for algorithms that return probabilities of classes. For example, to set that a given prediction will be positive only if the probability returned by the model is higher than 0.9.

True negative rate - (a.k.a. specificity)

$$\frac{TrueNegative}{TrueNegative + FalsePositive} \tag{1.34}$$

F1-score - harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. Not just average, because of skewed classes problem. Why the following formula? Debug it when put $P = 0 \, or \, R = 0$, and then also $P = 1 \, and \, R = 1$ and you will see that it is always between 0 and 1.

$$F_{1=} \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}} = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{1.35}$$

- There exists $F_\beta - score$. It affects a "weight" of either precision to have bigger, or recall. If we want treat them equally (most cases, but it depends on use-case of resulting system), then $\beta = 1$. If
  - β in range (0,1), then Recall is considered to be lower than Precision.
  - β bigger than 1, then Recall is considered to be higher than Precision.

  $$F_\beta = (1 + \beta^2) * \frac{Precision * Recall}{(\beta^2 * Precision) + Recall} \tag{1.36}$$
- Often a resulting metric of some learned model.

Confusion matrix - (also known as error matrix) is a specific table layout that allows visualization of the performance of an algorithm. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (**or vice versa**, but very often like on the figure below). Confusion matrix is used to calculate two other performance metrics: precision and recall.

		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	**True positive**, Power	**False positive**, Type I error
	Predicted condition negative	**False negative**, Type II error	**True negative**

Figure 1.28: Confusion matrix scheme, from Wikipedia

Area under ROC Curve (AUC)

- The ROC curve (receiver operating characteristic, from radar engineering) is also commonly used method for measuring performance of classification models. ROC curve uses a combination of the **true positive rate** (TPR is basically the recall, defined as $\frac{TP}{TP+FN}$) and **false positive rate** (FPR, proportion of negative examples predicted incorrectly, defined as $\frac{FP}{FP+TN}$).
- ROC curves can only be used with classifiers that return some confidence score or a probability of prediction.
- To draw a ROC curve, you first discretize the range of the confidence score. It is it $[0, 1]$, then you can discretize if like this: $[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$. Then, you use each discrete value as the prediction threshold and predict labels of examples in your dataset using the model and this threshold. For example, if you want to compute TPR and FPR for the threshold equal to 0.7, you apply the model to each example, get the score, and if the score is higher than or equal to 0.7, you predict the positive class. Otherwise, you predict the negative class. In case of ROC curve, upper right corner is when the threshold is 0, thus all predictions will be positive, and if the threshold is 1, then no positive prediction will be made and both TPR and FPR will be 0 (lower left corner).
- The higher the area under the ROC curve (AUC), the better the classifier. Usually, if your model be haves well, you obtain a good classifier by selecting the value of the threshold that gives TPR close to 1while keeping FPR near 0.
- ROC curves are popular because they are relatively simple to understand, they capture more than one aspect of the classification (taking both, false positives and false negatives into account), and allow visually and with low effort comparing the performance of different models.

An example - Suppose you are working on a spam classifier, where spam emails are positive examples ($y = 1$) and non-spam emails are negative examples ($y = 0$). You have a training set of emails in which 99% of the emails are non-spam and the other 1% is spam.

- If you always predict non-spam, your classifier will have 99% accuracy on the training set, and it will likely perform similarly on the cross validation set. The classifier achieves 99% accuracy on the training set because of how skewed the classes are. We can expect that the cross-validation set will be skewed in the same fashion, so the classifier will have approximately the same accuracy.
- If you always predict spam, your classifier will have a recall of 100% and precision of 1%. Since every prediction is ($y = 1$), there are no false negatives, so recall is 100%. Furthermore, the precision will be the fraction of examples with are positive, which is 1%.

- f you always predict non-spam, your classifier will have a recall of 0%. Since every prediction is $(y = 0)$, there will be no true positives, so recall is 0%.

Combining multiple evaluation metrics

- Suppose you care about both the accuracy and the running time of a learning algorithm. It seems unnatural to derive a single metric by putting accuracy and running time into a single formula, such as $accuracy - 0.5 * runtime$. Instead, define what is an "acceptable" running time. Lets say anything that runs in 100ms is acceptable. Then, maximize accuracy, subject to your classifier meeting the running time criteria. Here, running time is a "satisfying metric" - your classifier just has to be "good enough" on this metric, in the sense that it should take at most 100ms. Accuracy is the "optimizing metric." **If you have N different criteria, then consider $N-1$ of them as "satisfying" and pick one as "optimizing" one.**

1.6 Evaluating a Learning Algorithm

"Choose dev and test sets to reflect data you expect to get in the future and want to do well on." [8]

There are 3 accuracy estimation methods: holdout, cross-validation, and bootstrap.

- This means, that we train some model, and then perform model selection or calculate prediction accuracy with these methods. CV and bootstrap refit a model from samples formed from the training set, in order to obtain additional information about the fitted model, for example test-set prediction error, standard deviation and bias of our parameter estimates[36].
- Training and test error can be different, as we can see on the figure below:

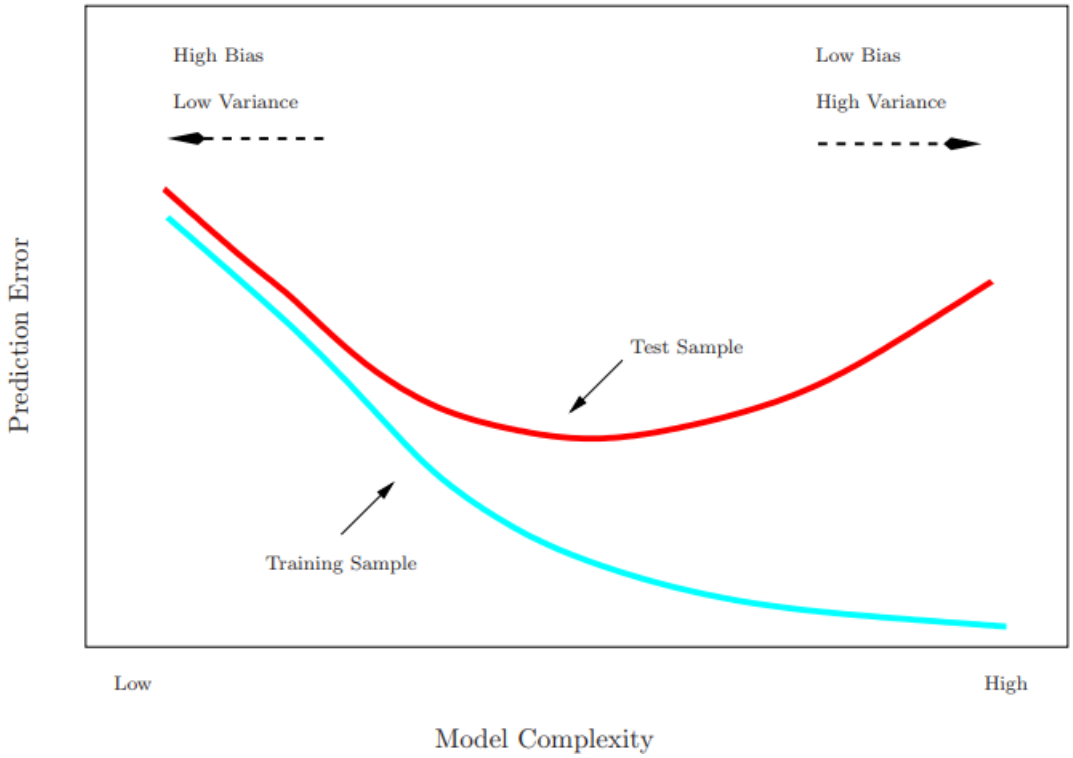


Figure 1.29: An example in which we can see that the training error rate is quite different from the test error rate. Test error rate can dramatically underestimate the training error rate.

[36] https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/cv_boot.pdf

Holdout method

- The data set is randomly separated into 2 mutually exclusive subsets usually called the training set (2/3) and the test set (1/3). This method involves a single run, in comparison to other methods.
- The more data we leave for test set, the higher bias of our estimation. Fewer test set instances however means, that the confidence interval for the accuracy will be wider. Generally, the larger the training data the better the classifier. The larger the test data the more accurate performance measures estimate (e.g. accuracy)[37].
- The function approximator fits a function using the training set only. Then the function approximator is asked to predict the output values for the data in the testing set (it has never seen these output values before). The errors it makes are accumulated as before to give the mean absolute test set error, which is used to evaluate the model.
- In random sub-sampling, the holdout method is repeated k times, and the estimated accuracy is derived by averaging the runs.
- Many sources claim, that holdout is the simplest kind of cross-validation. Many other sources instead classify holdout as a type of simple validation, rather than a simple or degenerate form of cross-validation.
- **Pros**
  - Fully independent data.
  - Only needs to be run once so has lower computational costs.
- **Cons**
  - It makes inefficient use of the data: 1/3 (testing subset) of dataset is not used for training the inducer.
  - The evaluation can have a high variance. It may depend heavily on which data points end up in the training set and which end up in the test set, and thus the evaluation may be significantly different depending on how the division is made.

[37] http://staffwww.itn.liu.se/~aidvi/courses/06/dm/lectures/lec6.pdf

Cross-validation

- **Any particular cross-validation experiment yields only an approximation of the true error rate.**
- Sampling without replacement. The same instance, once selected, can not be selected again for a sample. There are no duplicate records in the training and test sets.
- A dataset can be repeatedly split into a training dataset and a validation dataset: this is known as cross-validation.
- Cross-validation doesn't work in situations where you can't shuffle your data, most notably in time-series.
- It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. So not for model building (training itself).
- The goal of cross-validation is to test the model's ability to predict new data that was not used in estimating it, in order to flag problems like overfitting **(and CV is mostly used for overcoming overfitting!)** and to give an insight on how the model will generalize to an independent dataset[38].
- One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, in most methods multiple rounds of cross-validation are performed using different partitions, and the validation results are combined (e.g. averaged) over the rounds to give an estimate of the model's predictive performance.
- Recent experimental results on artificial data and theoretical results in restricted settings have shown that for selecting a good classier from a set of classifiers (model selection), 10-fold cross-validation may be better than the more expensive leave-one-out cross-validation [6].

[38] https://en.wikipedia.org/wiki/Cross-validation_(statistics)

The whole process

(from[39])

1. **Training set**: **model is initially fit on this subset**, for fitting a model's parameters (e.g. weights of connections between neurons in artificial neural networks).

2. **Validation set**: **this subset is optional**, and it is aimed to **minimizing overfitting** problem. Mostly used for tuning the hyper-parameters. This data is used during training to assess how well a model is currently performing - the performance of the ANN on this data may be used to guide the training in some way (e.g. controlling the learning rate, deciding when to stop training, number of hidden units in ANN, choosing between several trained networks - different architectures for instance), but you are not adjusting weights (parameters) of model (ANN for example) with this set.
   - You're just verifying that any increase in accuracy over the training data set actually yields an increase in accuracy over a data set that has not been shown to a model before, or at least a model hasn't trained on it. If the accuracy over the training data set increases, but the accuracy over then validation data set stays the same or decreases, then you're overfitting your model and you should stop training.[40]
   - **The validation dataset functions as a hybrid: it is training data used by testing, but neither as part of the low-level training nor as part of the final testing.**

3. **Test set**: to assess the performance (i.e. generalization and predictive power). **Ideally, should be used once only, after training is complete.**
   - When the data in the test dataset has never been used in training (for example in cross-validation), the test dataset is also called a holdout dataset.
   - **It is used to asset the performance (i.e. generalization) of a fully specified classifier.**

[39] https://en.wikipedia.org/wiki/Training,_test,_and_validation_sets and https://www.researchgate.net/post/what_is_the_difference_between_validation_set_and_test_set

[40] https://stackoverflow.com/questions/2976452/whats-is-the-difference-between-train-validation-and-test-set-in-neural-netwo

Types of CV

- **Leave-p-out cross-validation** - it involves using p observations as the validation set and the remaining observations as the training set. This is repeated on all ways to cut the original sample on a validation set of p observations and a training set.
- **Leave-one-out cross-validation** - this is a particular case of the previous one with $p = 1$.
  - It is k-fold cross validation taken to its logical extreme, with $k = N$, the number of data points in the set. That means that N separate times, the function approximator is trained on all the data except for one point and a prediction is made for that point. As before the average error is computed and used to evaluate the model.
  - Mainly used for rather small datasets.
  - LOOCV sometimes useful, but typically doesn't shake up the data enough. The estimates from each fold are highly correlated and hence their average can have high variance (unreliable estimates), and the bias is minimized (in fact it is almost unbiased).
  - **Pros** - makes best use of the data for training. Increases the chance of building more accurate classifiers – involves no random sub-sampling.
  - **Cons** - very computationally expensive. Stratification is not possible[41].
- **k-fold cross-validation** - the original sample is randomly partitioned into k mutually exclusive sub-samples, of approximately equal size. The inducer is trained and tested k times (basically k times repeated holdout method).
  - Of the k sub-samples (folds), a single sub-sample is retained as the validation data for testing the model, and the remaining $k - 1$ sub-samples are used as training data (each fold contains equal number of data samples). The cross-validation process is then repeated k times (so, you build k models), with each of the k sub-samples used exactly once as the validation data. The k results can then be averaged to produce a single estimation (averaging results from predictions of k models). All observations are used for both training and validation, and each observation is used for validation exactly once.
  - Results in [6] indicate that stratification is generally a better scheme, both in terms of bias and variance, when compared to a regular cross-validation.
  - For model selection it is recommended to use stratified 10-fold CV [6].
  - We expect an inducer stability to hold more in 20-fold cross-validation than in 10-fold cross-validation and both should be more stable than holdout of 1/3. In fact, the bias of 10 or 20 folds is (according to measurements) good enough [6]. On the contrary, 2 or 3 folds are usually very biased. However, the variance is increased (instability of the training sets themselves).

[41] Ensures that each class is represented with approximately equal proportions in both subsets.

  - **Pros** - it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set $k - 1$ times. The variance of the resulting estimate is reduced as k is increased.
  - **Cons** - a training algorithm has to be rerun from scratch k times, which means it takes k times as much computation to make an evaluation.

- **Repeated random sub-sampling validation** (also known as **Monte Carlo cross-validation**) - we randomly split the dataset into training and validation data. For each such split, the model is fit to the training data, and predictive accuracy is assessed using the validation data. The results are then averaged over the splits.
  - Repeating CV multiple times using different splits into folds.
  - **Pros** - advantage (over k-fold CV) is that the proportion of the training/validation split is not dependent on the number of iterations (folds).
  - **Cons** - some observations may never be selected in the validation sub-sample, whereas others may be selected more than once. In other words, validation subsets may overlap.

Examples of usage

- CV can be used to compare the performances of different predictive modeling procedures. For example, suppose we are interested in optical character recognition, and we are considering using either support vector machines (SVM) or k nearest neighbors (k-NN) to predict the true character from an image of a handwritten character. Using cross-validation, we could objectively compare these two methods in terms of their respective fractions of misclassified characters. If we simply compared the methods based on their in-sample error rates, the k-NN method would likely appear to perform better, since it is more flexible and hence more prone to overfitting compared to the SVM method.
- CV can also be used in variable selection (also known as **feature selection**).
- In the context of linear regression, CV is also useful in that it can be used to select an optimally regularized cost function.
- Consider the following situation[42]: 2 classes, 5000 predictors (input features for example), 50 samples:
  1. Find that 100 predictors have the largest correlation with the class labels.
  2. Then we apply a classifier for example logistic regression using only these 100 predictors.

[42] https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/cv_boot.pdf

Estimation of test set performance of this classifier **cannot** be done by applying CV in 2), forgetting about 1) - because **the procedure already seen the labels of the training data and made use of them**. This is a form of training and must be included in the validation process. **The correct process** is to apply CV in steps 1 and 2.

- NOT TRUE, that training, validation, and test sets, must have the same distributions of data [8].
  - **Example**: Users of your cat pictures app have uploaded 10,000 images, which you have manually labeled as containing cats or not. You also have a larger set of 200,000 images that you downloaded off the internet. How should you define train/dev/test sets? Since the 10,000 user images closely reflect the actual probability distribution of data you want to do well on, you might use that for your dev and test sets. If you are training a data-hungry deep learning algorithm, you might give it the additional 200,000 internet images for training. Thus, your training and dev/test sets come from different probability distributions. How does this affect your work? Instead of partitioning our data into train/dev/test sets, we could take all 210,000 images we have, and randomly shuffle them into train/dev/test sets. In this case, all the data comes from the same distribution. But I recommend against this method, because about 205,000/210,000 $\approx$ 97.6% of your dev/test data would come from internet images, which does not reflect the actual distribution you want to do well on. Most of the academic literature on machine learning assumes that the training set, dev set 11 and test set all come from the same distribution. In the early days of machine learning, data was scarce. We usually only had one dataset drawn from some probability distribution. So we would randomly split that data into train/dev/test sets, and the assumption that all the data was coming from the same source was usually satisfied. But in the era of big data, we now have access to huge training sets, such as cat internet images. Even if the training set comes from a different distribution than the dev/test set, we still want to use it for learning since it can provide a lot of information. Instead of putting all 10,000 user-uploaded images into the dev/test sets, we might instead put 5,000 into the dev/test sets. We can put the remaining 5,000 user-uploaded examples into the training set. This way, your training set of 205,000 examples contains some data that comes from your dev/test distribution along with the 200,000 internet images.
  - **Usually they are all from the same distribution, especially in academic research, but it is important to understand that different training and dev/test distributions offer some special challenges** (and perhaps also some good advantages, like in the previous example).
- When to decide if use all your data?
  - **Example**: Suppose your cat detector's training set includes 10,000 user-

uploaded images. This data comes from the same distribution as a separate dev/test set, and represents the distribution you care about doing well on. You also have an additional 20,000 images downloaded from the Internet. Should you provide all 20,000+10,000=30,000 images to your learning algorithm as its training set, or discard the 20,000 Internet images for fear of it biasing your learning algorithm?

* When using earlier generations of learning algorithms (such as hand-designed computer vision features, followed by a simple linear classifier) there was a real risk that merging both types of data would cause you to perform worse. Thus, some engineers will warn you against including the 20,000 Internet images.
* But in the modern era of powerful, flexible learning algorithms - such as large neural networks - this risk has greatly diminished. If you can afford to build a neural network with a large enough number of hidden units/layers, you can safely add the 20,000 images to your training set. Adding the images is more likely to increase your performance.
* If you think you have data that has no benefit,you should just leave out that data for computational reasons.

Bootstrap

- In comparison to holdout and cross-validation, these methods use uses (uniformly) sampling with replacement to form the training set. This means, that an instance may occur more than once.
- Usage is for estimation of properties of estimator[43] (such as its variance) by measuring those properties when sampling from an approximating distribution.
- Bootstrap has low variance, but extremely large bias on some problems in comparison to cross-validation.
- Primarily used to obtain standard errors of an estimate and confidence intervals. Cross-validation provides a simpler, more attractive approach for estimating prediction error.
- **Probably the best way of estimating performance for very small data sets.**

[43] https://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29

1.7 Derivations

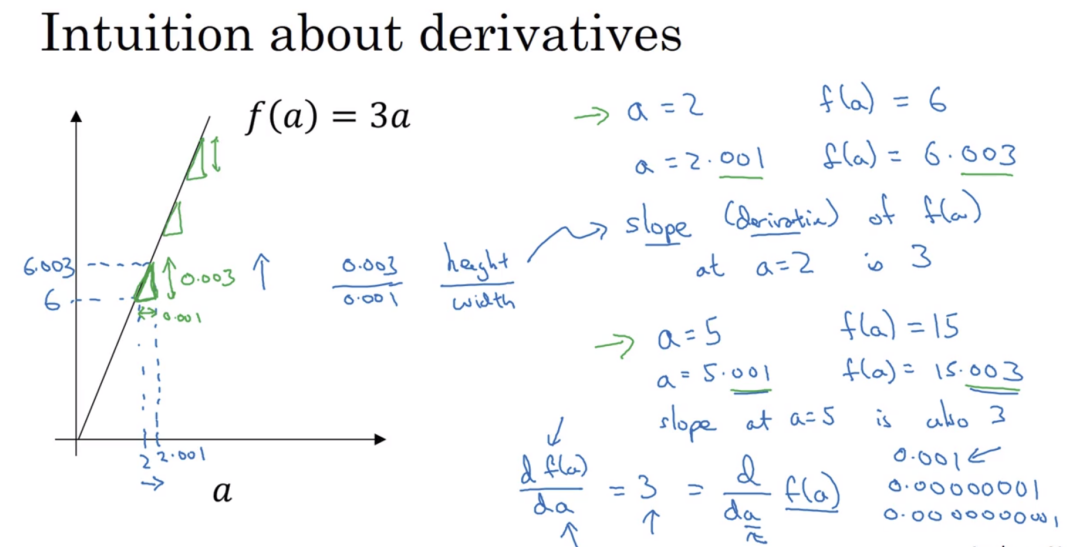


Figure 1.30: An example of derivation of a simple function. We can see that the first derivation of some function is just a slope.

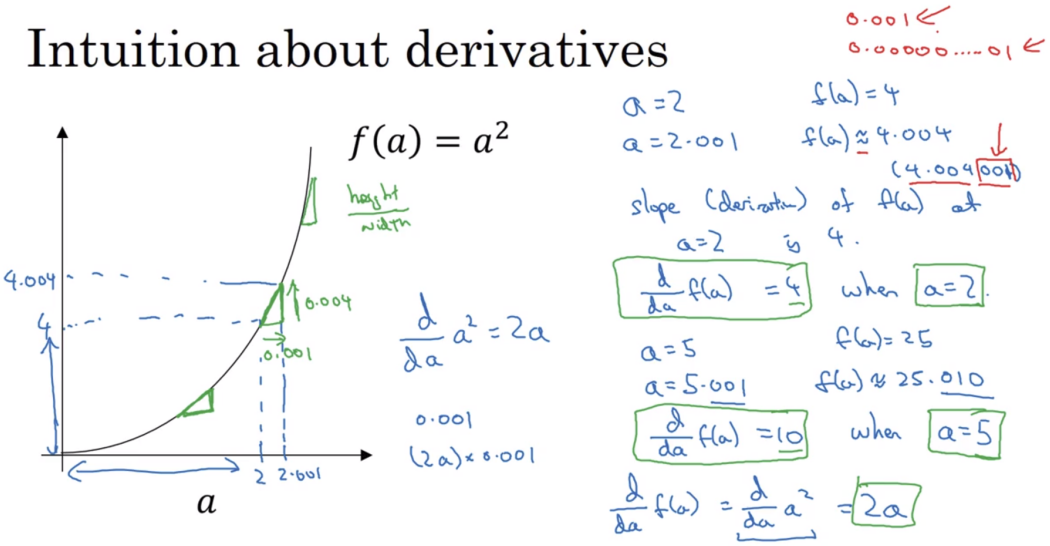


Figure 1.31: Another example of derivation of a simple function.

1.8 Eigenvectors, Eigenvalues, and PCA

- Let's consider a matrix to be a tool for making linear transformation and focus on a mapping of a vector. In other words, a matrix can transform a magnitude and direction of a vector sometimes also into a lower dimension.
- For example, $\begin{pmatrix} 3 & -2 \\ 1 & 0 \end{pmatrix} . \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$, or $\begin{pmatrix} 3 & -2 \\ 1 & 0 \end{pmatrix} . \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$, or $\begin{pmatrix} 3 & -2 \\ 1 & 0 \end{pmatrix} . \begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} -5 \\ -1 \end{pmatrix}$. Draw and see what happended. You have a vector $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and then you have $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$, so its magnitude and direction was changed.
- So basically, if you have matrix $A = \begin{pmatrix} 3 & -2 \\ 1 & 0 \end{pmatrix}$ and $X = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, then linear transformation of X is $\begin{pmatrix} 3 \\ 1 \end{pmatrix}$.
- **Eigenvector** is a vector that when multiplied by a given transformation matrix, is a **scalar multiple** of itself. **Eigenvalue is this scalar multiple. In ML, transforming a vector only by some factoring scalar is very useful.**
- For the example, $A = \begin{pmatrix} 3 & -2 \\ 1 & 0 \end{pmatrix}$ and $X = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ vector X is eigenvector of the matrix A. After linear transformation, we can see that resulting vector is $\begin{pmatrix} 4 \\ 2 \end{pmatrix}$ and the direction was not changed; only magnitude. In this case, 2 is eigenvalue and is usually notes as λ.
- How to find these numbers? The first step is always to look for eigenvalue. For example, let A be transformation matrix, $\bar{X}$ to be eigenvector, and λ to be eigenvalue. Then, $A.\bar{X} = \lambda.\bar{X}$. For finding out eigenvalue, we can use identity matrix I_n. The previous equation will be still the same, $\lambda.\bar{X} = \lambda.I_n.\bar{X}$. So, $A.\bar{X} = \lambda.I_n.\bar{X}$, and then $A.\bar{X} - \lambda.I_n.\bar{X} = 0$, and $(A - \lambda.I_n).\bar{X} = 0$. if $A - \lambda.I_n$ is invertible, we can divide this in both sides and end up with $\bar{X} = \frac{0}{A - \lambda.I_n} => \bar{X} = 0$. But this is not what we want. We want to find λ such that the matrix $A - \lambda.I_n$ is not invertible. This is not invertible if its determinant is equal to 0. This equation is also called characteristic equation of a matrix.
- BTW, **triangular matrix** is such square matrix, where all the elements either above or below the diagonal are zero. Then, its determinant is calculated by simple multiplication of all the elements on its main diagonal and finding out solution to such polynomial function.

- **PCA** can find linear transformation that will reduce dimensions (of data / vectors / matrices).
- Covariance matrix - to find out if components (from PCA) are independent or not, you can calculate this covariance matrix. It is a symmetric matrix, **for example** $\begin{pmatrix} V_a & C_{a,b} \\ C_{a,b} & V_b \end{pmatrix}$ that expresses how each of the variables (in this example, two variables V_a and V_b would be for data that has 2 variables for each data point, such as x and y coordinates) relate to each other. The goal is to find a new axis for our data (in this example each data point has 2 variables) such that we can represent each two dimensional points by using just one dimensional scalar value R, called projection of the datapoint onto the new axis. To achieve this, we consider covariance matrix to be our transformation matrix, and **we have to calculate eigenvalues and eigenvectors of the covariance matrix**. 2x2 covariance matrix has 2 eigenvectors with 2 corresponding eigenvalues. Eigenvectors points to specific directions with relation to the data. The goal is to select one of these eigenvectors as a new axis. We will then take the projection of the original data point onto eigen vector and that we would be our reduced one dimensional data. And we will choose component (eigenvector) with the largest spread - largest variance, because that's where the most of the information is.

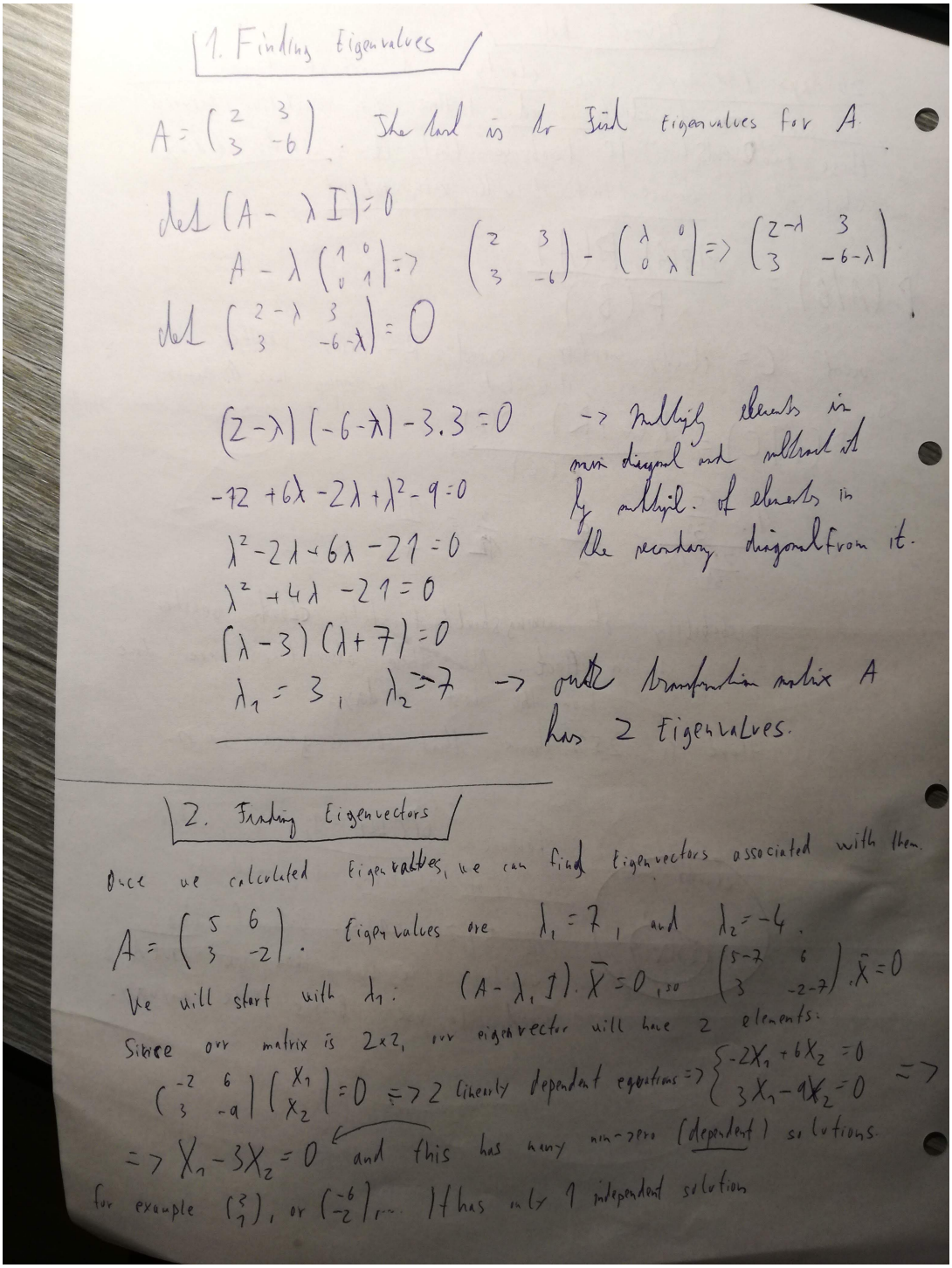


Figure 1.32: A simple example that calculates eigenvalues and eigenvectors.

1.9 Minimizing Cost Function

Newton method

Uses 2 derivations.

Gradient descend

- Iterative optimization algorithm to finding a minimum of a function, in our case a cost function. Results in local minimum (if we want local maximum, we need to use gradient ascend algorithm). Local minimum is when partial derivative of cost function (see below) is 0.
- **Idea**
  1. start at some initial parameters (e.g. Θ_0, Θ_1 for Univariate linear regression).
  2. keep changing parameters (Θ_0, Θ_1) to reduce cost function until we hopefully end up at a minimum.
- **Practical tips and notes**
  - Gradient descend can be much more faster when using feature scaling (see the next section). This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.
  - To make sure that gradient descend is working correctly - plot on x-axis number of iterations, and on y-axis cost function $J(\theta)$ as gradient descent runs. **Cost function** in the graph **must decrease after every iteration**. If otherwise, learning rate α should be probably decreased (see a shape of a given cost function, maybe more scenarios can occur e.g. a bug in code). If α is small enough, it must decrease in every iteration, but the algorithm is probably very slow.
  - How many iterations gradient descend needs to converge depends on the application. It can be for example from 3,000 to 3,000,000 ... There are also automatic converge test - if cost function decreased by less than some ε, for example $\varepsilon = 10^{-3}$ in one iteration. But it is also good enough to use only graph mentioned before (since it is also problem to estimate ε).
  - Works pretty well even with a great number of features.
  - Complexity $O(kn^2)$ where n is a number of features and k is a number of iterations.
  - **Initialization** - the best is to use random initialization! For thetas in gradient descent and advanced optimization method.
  - We can slowly decrease learning rate α over time if we want ϑ to converge (e.g. $\alpha = \frac{const1}{iterationnumber+const2}$, but some people are not doing this because you need to tune these 2 constants in the equation).

- **Types**

  Batch:

  - Classical, basic one, where we iterate through all training samples and then perform update, so all training samples are used in each step.

  Stochastic (also known as online learning or incremental learning):

  1. randomly shuffle dataset
  2. iterate through all training samples and perform update of parameters

  - So stochastic gradient descent works with first sample, performs one step with gradient descent with cost function. In comparison to classic (batch) gradient descent, it does not wait to iterate through all training samples and calculates SUM, but the improvement starts from the beginning. OR it can work with just a subset (100 samples for instance) of training data, and then perform update, so not with all the samples.
  - It is basically a heuristics, where a convergence to global minimum is slower and consequent, but sometimes we can go in wrong direction; however the algorithm is around global optima.
  - So this method is a little less precise, but faster. However, **it is inefficient**, because we are working with just 1 sample and cannot make a benefit out of **vectorization**.
  - To debug if it is working and converging - average the last 1,000 costs (iterations) that we computed and plot it (in BGD we plot $J_{train}(\theta)$ as a function of number of iterations of gradient descent, so this is similar but not with all training samples and also not with just 1 sample). If not 1,000, but 5,000 example for instance, maybe we can get a smoother curve (again, number of iterations vs cost). Small number of examples can cause that the plot is very noisy (almost like 'oscillating') which is normal.

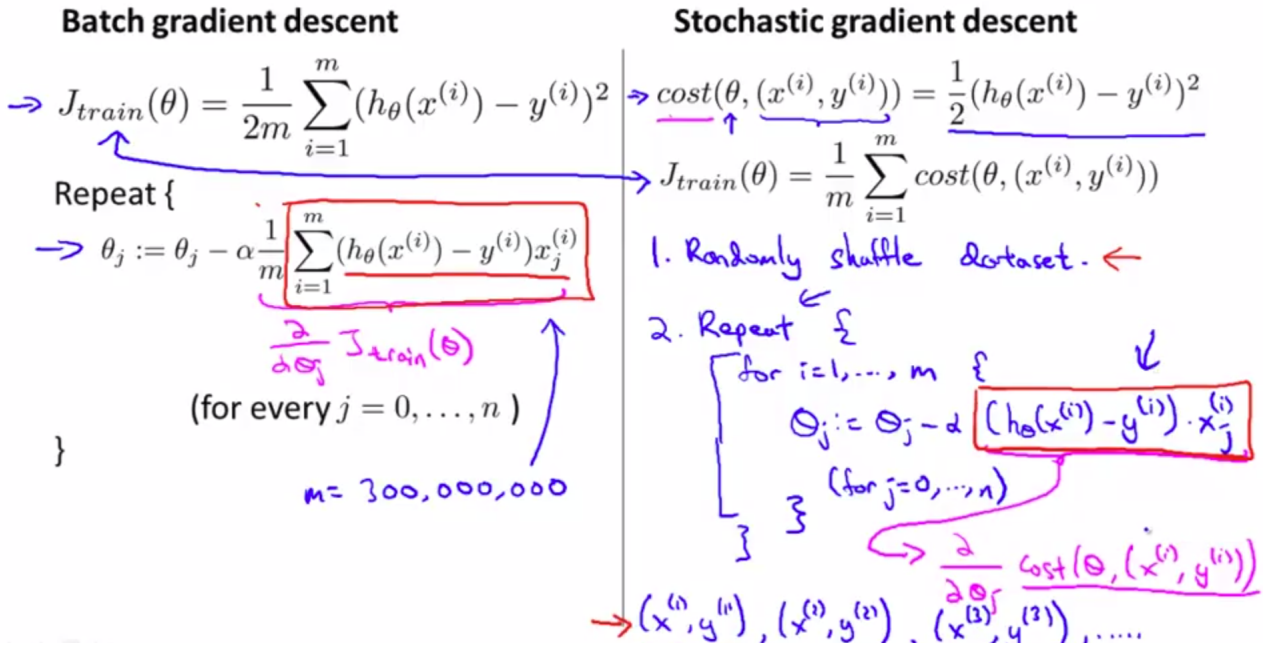


Figure 1.33: A difference between batch and stochastic gradient descent used in linear regression. In SGD, its cost function measures how well is a hypothesis doing on a single training example. Unlike Batch gra- dient descent, SGD does not end up in global minimum. Instead, the algorithm will end up in some close region of minima (this is for algorithms such as linear regression). But in practice this is not a problem, we are satisfied with such result. In SGD, the second step can be performed multiple times - if you need to still improve accurac even if you run a learning algorithm against al training samples. But you may end up in good accuracy even after the first pass, which can be true if you have a lot of data. If you have less data, maybe it is needed to run it more (like 1-10x) times.

Mini-batch:

- Something between batch and stochastic versions. Mini-batch gradient descent uses only a small amount of samples, for example 10 or 100. Why? Because **derivations are just an approximation of direction, no need to compute it on all dataset in each iteration**. Most optimization algorithms work with mini-batches (and with the first derivations).
- In practice, the most used variant. You use vectorization and a progress is made without a need to wait for processing the whole training set.
- **One pass through our training set (multiple batches of samples) is called 1 epoch.** Let's say that we have 100 samples in a batch and 50,000 samples in our training set. So in mini-batch gradient descent, a single pass

though the training set allows us to make 500 gradient descent steps. In comparison, in batch gradient descent, a single pass through training set allows us to make just 1 gradient descent step.

– Full batch gradient descent is **not guaranteed to find a better local minimum** than mini-batch gradient descent. Mini-batch gradient descent can search through weight space due to noise and potentially escape a bad local minima.

Figure 1.34: A comparison of different mini-batch sizes. Basically this is a comparison of extremes as well, so stochastic, mini-batch and batch versions. If you have a small training set, use batch gradient descent. Otherwise, typicall size is a number from 64 to 128 samples. Sometimes it runs faster if it is a power of 2, and also using 256 or even 512 samples is still common. What number, it depends also on CPU/GPU memory, if it fits - this depends on a data. Choosing a good mini-batch size is basically a hyperparameter. Andrew Ng recommends to try several values and then pick one that works the best. Choosing the best mini-batch size is about making a compromise. Too small, and you don't get to take full advantage of the benefits of good matrix libraries optimized for fast hardware. Too large and you're simply not updating your weights often enough. What you need is to choose a compromise value which maximizes the speed of learning. Fortunately, the choice of mini-batch size at which the speed is maximized is relatively independent of the other hyper-parameters (apart from the overall architecture), so you don't need to have optimized those hyper-parameters in order to find a good mini-batch size. The way to go is therefore to use some acceptable (but not necessarily optimal) values for the other hyper-parameters, and then trial a number of different mini-batch sizes, and scaling learning rate. Plot the validation accuracy versus time (as in, real elapsed time, not epoch!), and choose whichever mini-batch size gives you the most rapid improvement in performance. With the mini-batch size chosen you can then proceed to optimize the other hyper-parameters.

- If it is decreasing, it is converging.
- If it is increasing, try to use smaller learning rate α.
- If it is not doing anything, then maybe try to use different model because it is for some reason not learning much.

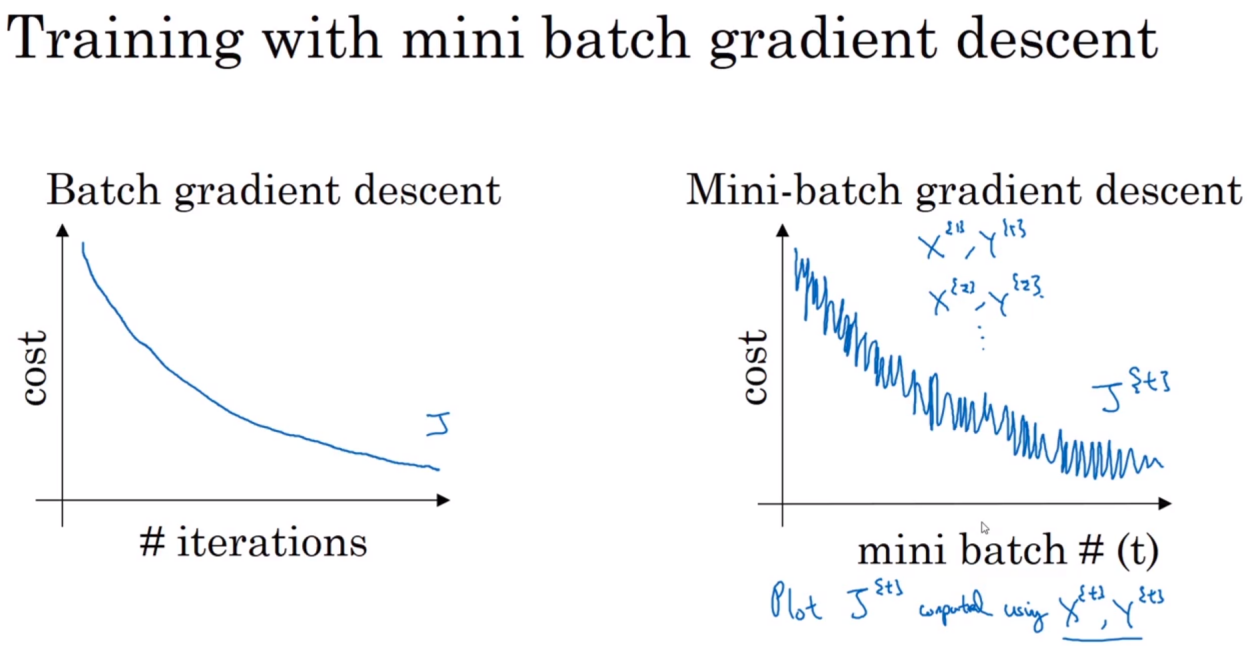


Figure 1.35: In mini-batch gradient descent, learning error (cost function) may not decrease on every iteration as it should in batch gradient descent. A plot is showing learning with multiple epochs. The reason is that some mini-batches can be relatively easy and some may be harder.

- **Adagrad**
  - Adaptive Subgradient Methods for Online Learning and Stochastic Optimization (from 2011).
  - Version of SGD, that scales learning rate α for each parameter according to the history of gradients. As a result, α is reduced for very large gradients and vice-versa.
- **Gradient descent with Momentum**
  - Almost always works faster than the standard gradient descent algorithm. The basic idea is to calculate an **exponentially weighted average** (see Definition 1.19) **of your gradients**, and then use that gradient to update your weights (and biases) instead. Considering exponentially weighted average and its bias correction, in practice not many people are applying bias

correction. The reason is that after 10 iteration, the moving average is already "warmed-up" and is no longer biased.

- Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations. Momentum takes into account the past gradients to smooth out the update.
- This result in smoothing the gradient steps (because we are using averaging). Steps are more direct and there is less oscillation. And the algorithm can take a more straightforward path into minimum.
- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much. Common values for β range from 0.8 to 0.999. Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.
- It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.

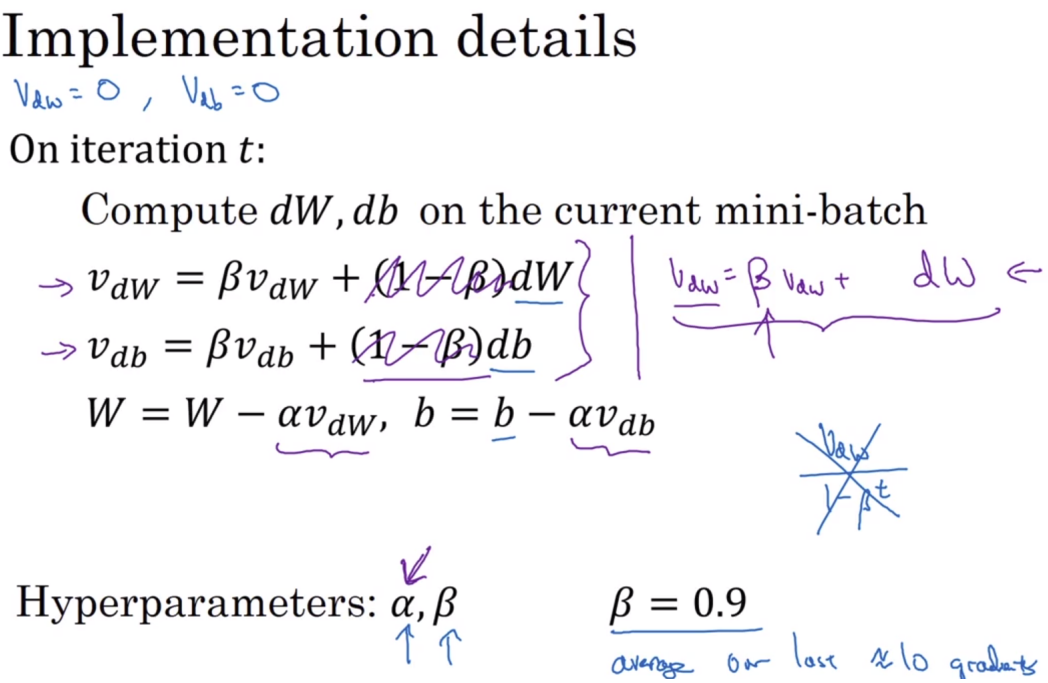
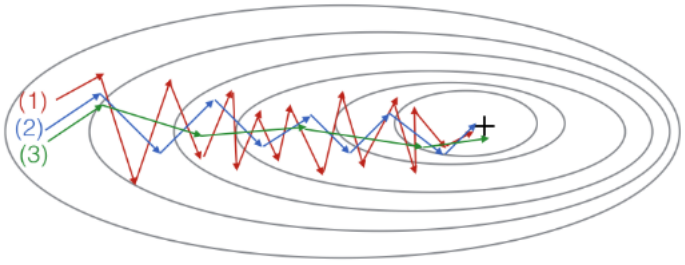


Figure 1.36: Implementation of gradient descent with momentum. The crossed formula in the figure is bias correction, but it is not being used in practice because after for instance 10 iterations, the moving average is warmed up and there is no longer need for a bias estimate. $\beta = 0.9$ is the most common value, but it can be considered as hyperparameter and tuned out. Purple formula on the right side is an alternative that can be seen in literature.



These plots were generated with gradient descent; with gradient descent with momentum (β = 0.5) and gradient descent with momentum (β = 0.9). Which curve corresponds to which algorithm?

Figure 1.37: An example of gradient descent with momentum learning. The correct answer is that (1) is gradient descent. (2) is gradient descent with momentum (*small* β). (3) is gradient descent with momentum (*large* β).

- So, basically we replace gradient descent update rule $w \to w' = w - \eta \nabla C$, where η is learning rate and ∇C is gradient vector, by:

$$v \to v' = \mu v - \eta \nabla C \tag{1.37}$$

$$w \to w' = w + v' \tag{1.38}$$

  where μ is a hyper-parameter controlling the amount of damping or friction in the system. $\mu = 1$ means no friction. So, the force of ∇C is modifying the velocity v, and the velocity is controlling the rate of change of w. Intuitively, we build up the velocity by repeatedly adding gradient terms to it. That means that if the gradient is in (roughly) the same direction through several rounds of learning, we can build up quite a bit of steam moving in that direction. So, if for example with each step the velocity gets larger down the slope, we move more and more quickly to the (local) minima. This can enable the momentum technique to work much faster than standard gradient descent. The problem is, that if gradient should change rapidly, then we could be moving in the wrong direction. For this reason, there is a hyper-parameter μ, in a range between 0 and 1.
- Momentum was probably inspired by **Hessian technique**, which is not used in practice. The reason is that it uses hessian matrix, which is extremely non-optimal (it has a lot of entries; for example, if a neural net has 10^7biases and weights, then this matrix would have 10^{14}entries, and having computation with such matrix is difficult in practice). But both these techniques uses the same idea - using information about how the gradient is changing.

- **RMSprop**
  - Root mean square prop, is a method how to speed up gradient descent. So, it is similar to momentum, and the aim is to reduce oscillations in gradient descent.
  - You can use a bigger learning rate and have faster learning without diverging in the vertical direction (see figure below). It is like Gradient descent with momentum, but we are squaring the derivatives, and then we take a square root at the end.
  - Combination of RMSProp with momentum is called **Adam**.
  - Fun fact is, that RMS Prop was first proposed by Geoffrey Hinton in a course on Coursera.

- **Adam**

  - Adaptive moment estimation (from 2015).
  - It is a combination of momentum and RMSprop algorithms. Most optimization algorithms that have been proposed don't generalize well to the wide range of deep neural networks. RMS prop and Adam do generalize well however.
  - There are following hyperparameters (the last two values were recommended by the authors of Adam paper):
    * α: needs to be tuned
    * β_1: 0.9 (dw Momentum) - also know as The first moment
    * β_2: 0.999 (dw^2RMS prop) - also know as The second moment
    * ϵ: 10^{-8}

- **Some other very advanced alternatives**
  - BFGS
  - L-BFGS
  - Conjugate gradient

  They are often faster than gradient descent, and there is no need to manually pick the learning rate α. These algorithms automatically try and pick learning rate α, which can be even different on every iteration. Andrew NG recommends, he uses them. They are very complex, even he did not know the details for over a decade.

- **Gradient checking** - recommended by Andrew Ng - if you have even a bit more complex model which uses backpropagation or similar gradient descent algorithm, ALWAYS perform gradient checking, which checks a presence of almost any bug in the gradient descent implementation. This also solves problems with buggy implementation of backpropagation algorithms in ANN (see algorithm on Figure 4.11).
  - an approximation of the derivative of cost function.
  - verification that a computation of derivative of cost function is truly correct.
  - however, after we use it and find out that our implementation of for example backpropagation is correct, we have to turn off gradient checking (for learning), since it is not optimal algorithm - forward propagation must be performed twice for every parameter in the network. Backprop is much faster.
- **Learning rate decay**
  - Another way to speed up learning algorithm.
  - Slowly decrease learning rate over time.
  - For example, the following implementation could be used:

$$\alpha = \frac{1}{1 + decay\,rate * epoch} * \alpha_0 \tag{1.39}$$

    where $decay\,rate$ and initial learning rate α_0 are hyperparameters and $epoch$ is a currently processed epoch of data.
  - Another approach is **exponential decay**. This will exponentially quickly degrade the learning rate.

$$\alpha = 0.95^{epoch} * \alpha_0 \tag{1.40}$$

  - But learning rate decay can be achieved also by manually changing learning rate during training, or automatically after a certain number of steps.

Normal Equation

- Another method for finding local optima of cost function. **This method solves θ analytically**.
- TL;DR much faster alternative than gradient descent, not usable with all machine learning algorithms and not suitable every time. Gradient descent is more universal.
- In comparison to gradient descent, there is no need to perform multiple steps and to choose learning rate α. But gradient descent scale better than normal equations method. Normal equations method is slow when there is a big amount (more than 10^3or 10^4) of features (invertion of a huge amount of matrices).
- Complexity is $O(n^3)$ (matrix inversion), where n is a number of features.
- We will minimize J by explicitly taking its derivatives with respect to the $\theta J's$, and setting them to zero.
- **No need** to do a **feature scaling**!
- Calculating a value of θ that minimizes a cost function:

$$\theta = (X^T * X)^{-1} * X^T * y \tag{1.41}$$

  where:
  - m is # of examples.
  - n is # of features.
  - X is a matrix of features of dimension $m * (n + 1)$ (+1 because of $x_0 = 1$ and it is another dimension for calculation purposes).
  - y is vector of output values.
- Or normal equation with regularization:

$$\theta = (X^T * X + \lambda * L)^{-1} * X^T * y$$

  where:
  - λ is a regularization parameter.
  - L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). So this is the identity matrix (though we are not including x_0), multiplied with a single real number λ.

- By the way, what if $X^T X$ matrix is non-invertible (a.k.a. singular / degenerate matrix)? This should happen only rarely. Then we probably have:
  - redundant features (linearly dependent) = delete some features
  - too many features ($m \leq n$) where m is a number of samples and n is number of features = delete some features or use regularization)
  - however, when we add the regularization term $\lambda * L$, then $X^T * X + \lambda * L$ becomes invertible

Dimension hopping

- This occurs when one can take the information contained in the dimensions of some input, and move this between dimensions while not changing the target.
- The canonical example is taking an image of a handwritten digit and translating it within the image. The dimensions that contain "ink" are now different (they have been moved to other dimensions), however the label we assign to the digit has not changed.

  Note that this is not something that happens consistently across the dataset, that is we may have a dataset containing two handwritten digits where one is a translated version of the other, however this still does not change the corresponding label of the digits.
- Another example - determining whether a given image shows a bike or a car. The bike or car might appear anywhere in the image. The input is the whole set of pixels for the image.

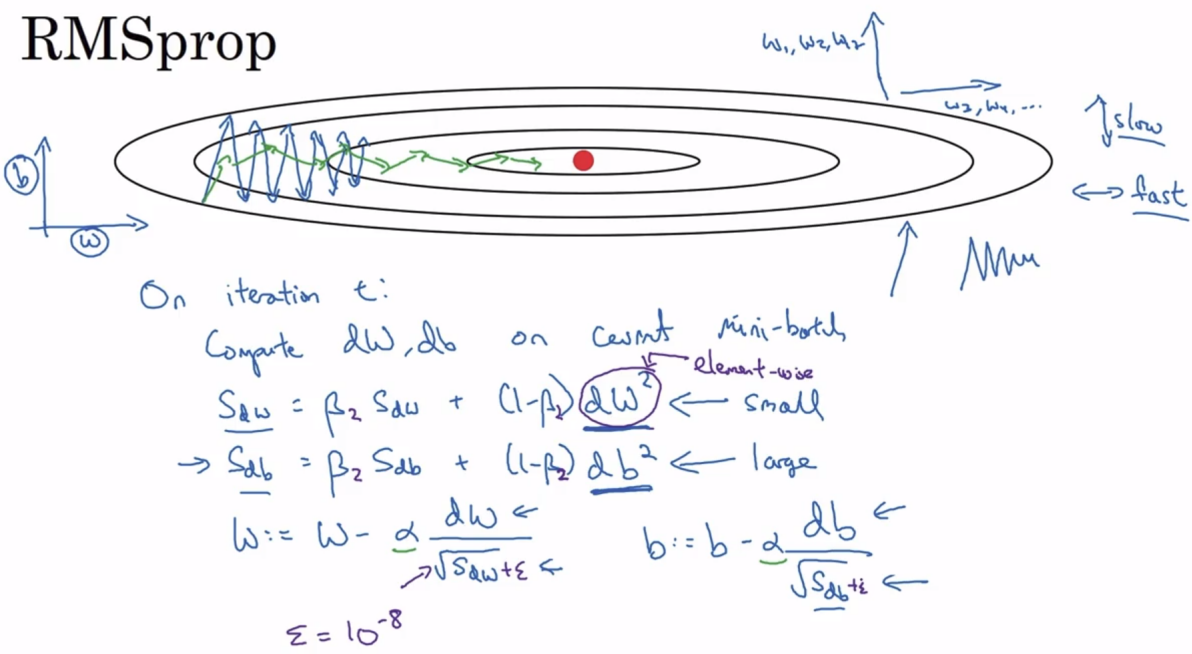


Figure 1.38: RMSprop intuition and formulas. In practice, to avoid a division by zero, we add a small number (here it is defined as $\varepsilon = 10^{-8}$) in denominator for such numberical stability. In traditional gradient descent, you can have a big oscillation in for example vertical direction. To mitigate oscillation on vertical direction and still have progress in horizontal direction, RMSprop is helpful. In this example, there is b on vertical axis and w on horizontal axes. In real-case scenario, there would be much bigger number of dimensions of parameters, for example on horizontal axis $w_3, w_4, ...$ but for the sake of intuition, this example is more simple. If db is large, it will be even a bigger number and eventually, the final b will have in its denominator large number (division by a big number is a small number). On the other hand, updates in w (horizontal direction) will keep going (because we are still dividing by small number). So, the most importantly, in dimensions you got oscillations in your gradient descent, you end up computing a larger sum (these $S_{d\,parameter}$) of weighted averages of squares and derivatives, and so you end up dumping out directions in which there are these oscillations. So, there is no need to know a direction where the oscillation occurs.

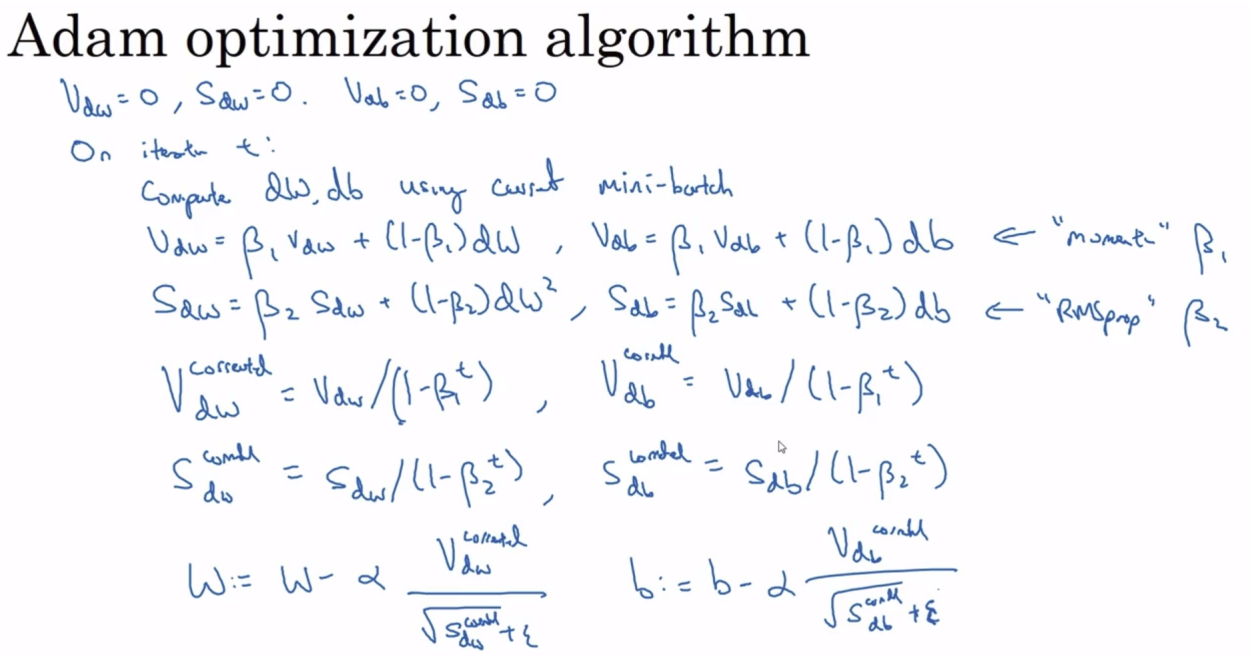


Figure 1.39: Adam optimization algorithm for gradient descent. In Adam, we also compute biases corrections (because of weighted average means). The algorithm has a number of hyperparameters: learning rate α, choice of moving weighted average β_1(called the first moment, default to 0.9) and β_2(called the second moment, for squared derivatives, default to 0.999), correction constant ε (does not matter very much, but the authors proposed to be equal to 10^{-8}), but in reality mostly learning rate is tuned.

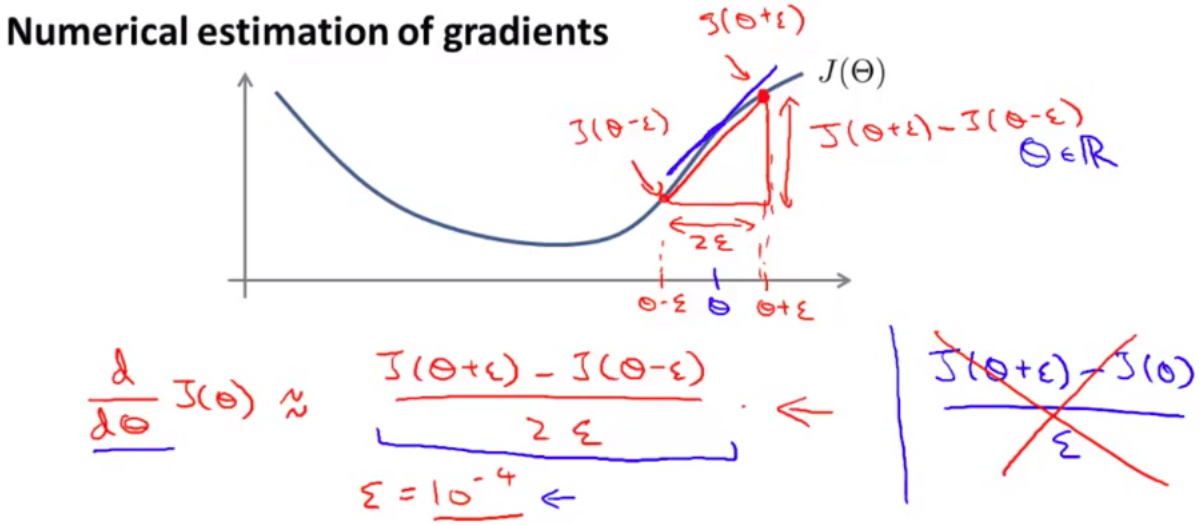


Figure 1.40: A visual representation of Gradient Checking (numerical estimation of the derivative). On the graph, a slope of the line between those 2 red points is an approximation to the derivative. Crossed formula (right side) is one-sided difference estimate and it is an alternative, but Andrew Ng uses two-sided difference estimate (on the left) since it is slightly more accurate. He also uses usually $\varepsilon = 10^{-4}$.

2 Linear Models

Consider the usage of these models on the following scenarios:

- **YES** = problem which is **linearly separable**.
- **YES** = **non-linear** problem with **few features**. We can use polynomial regression "trick" on some linear model, so that we have more features, but can find solution anyway.
- **NO** = once the problem is non-linearly separable and we have many (dozens or hundreds) of features, application of polynomial regression on some of these models is not reasonable. The reason is, that we would end up with extremely great amount of features, and for this there are other models, like SVM or ANN.
  - Polynomial regression - from 100 features it would result in ~5000 features, it grows ~$O(n^2)$,where n is the number of original features (closed to $\frac{n^2}{2}$, when we include all the quadratic terms - for example, if we have 10.000 features, we would end up with $5 * 10^7$ features).
  - So many features can end up with overfitting the training set and a requirement of a great computational power.
  - If we would keep only certain, for example only quadratic features, it would be not enough to fit even training set probably.

2.1 Linear Regression

- Linear regression is one of the simplest approach for predictive analysis using linear algebra.
- It is a popular regression learning algorithm that learns a model which is a linear combination of the input features.
- Linear regression **does not need feature scaling** necessary, but it can converge faster with it.
- It is actually very similar to SVM with linear kernel. Hypothesis function of SVM is $f(x) = sign(wx - b)$ (where $sign$ is a function that returns $+1$ if its input is positive number, and returns -1 if its input is a negative number) and for LR it is $f(x) = wx + b$).

The rest of this section describes univariate and multivariate linear regression. Definition of their cost functions and gradient descent is similar since univariate is a special form of multivariate linear regression.

Univariate Linear Regression

- Very simple, this is also known as Linear regression with 1 variable (feature).
- **Hypothesis function**

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{2.1}$$

- **Cost function**
  - Always convex, so it always converge to 1 local minimum = global minimum. Loss function is squared error loss (sometimes only with $\frac{1}{m}$ instead of $\frac{1}{2m}$). Here, the cost function is given by the **average loss**, also called the **empirical risk**. Intuitively, **squared penalties** are also advantageous because they **exaggerate the difference between the true target and the predicted one** according to the value of this difference. We might also use the powers 3 or 4, but their derivatives are more complicated to work with:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.2}$$

    where
    - J is cost function (in the terms that we will want to minimize it = find the best values for all the parameters, in this case θ_0, θ_1).
    - m is # of training samples in dataset.

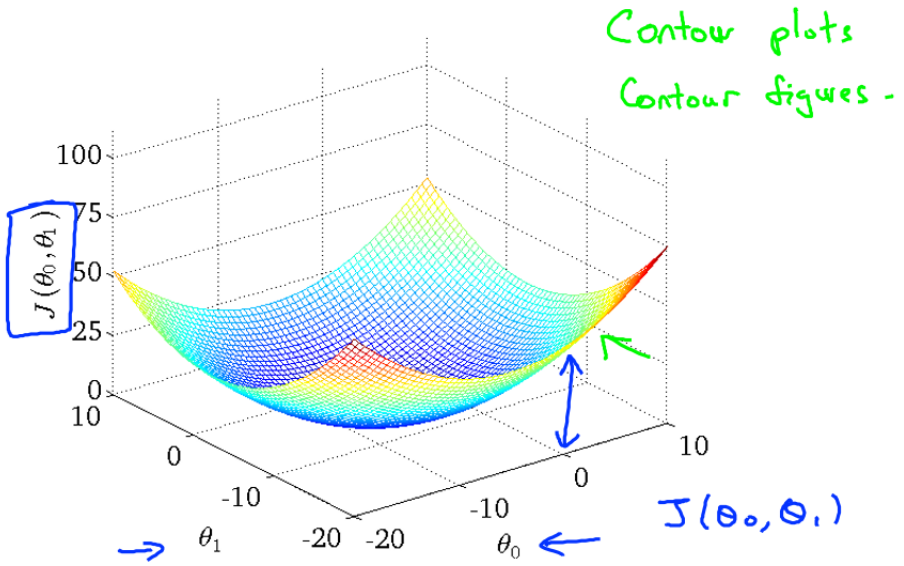


Figure 2.1: An example of visualization a cost function of univariate linear regression in contour plot.

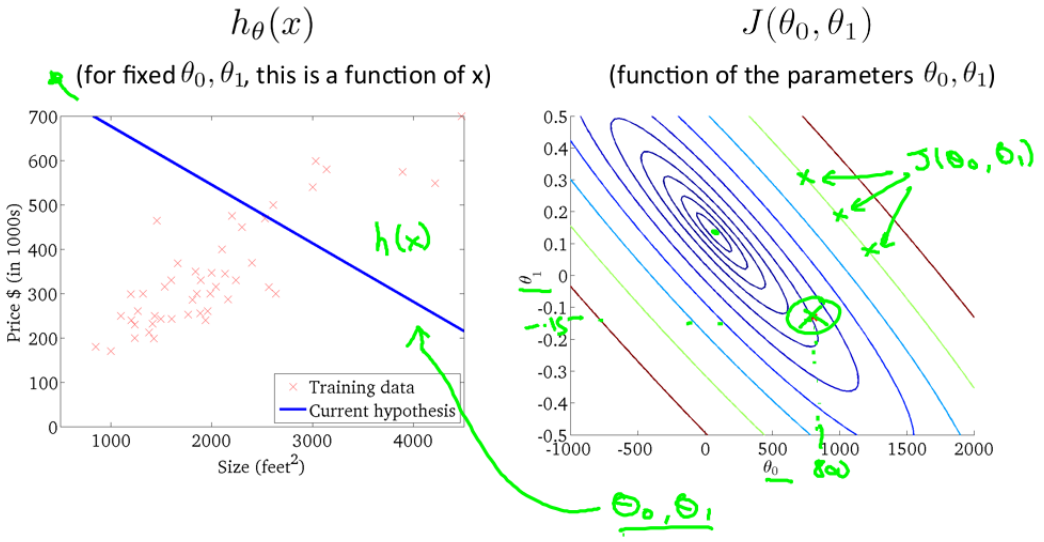


Figure 2.2: Another example of visualization a cost function of univariate linear regression in contour plot. We can see, that there can be different values for parameters, and resulting value of cost function will be the same (three green points on the right side).

- **Cost function can be regularized** in the same style like in Definition 2.5**.**

- **Gradient Descent**

repeat until convergence {

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for j = {0, 1} - univariate linear regression)

}

where:

  - α is a learning rate (control how big step we perform). If it is too small, gradient descend can be slow. If too big, it can not end up in local minimum (it fails to converge), or it can even diverge.
  - The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$.
  - All thetas have to be update simultaneously - in parallel way across all parameters, or using temporary variables.
  - If we are already (initial step) in local minimum, the algorithm will have partial derivative of J equals to 0 (slope will be horizontal), and gradient descent will not perform any steps (also not to global minimum if there is any).
  - More and more it gets to minimum, derivative is closer and closer to 0 and gradient descend performs smaller and smaller steps - no need to decrease α over time (but all this probably depends on a function).
  - Following graph shows when partial derivative of J is a positive number or negative number:

Regardless of the slope's sign for $\frac{d}{d\theta_1} J(\theta_1)$, θ_1 eventually converges to its minimum value. The following graph shows that when the slope is negative, the value of θ_1 increases and when it is positive, the value of θ_1 decreases.

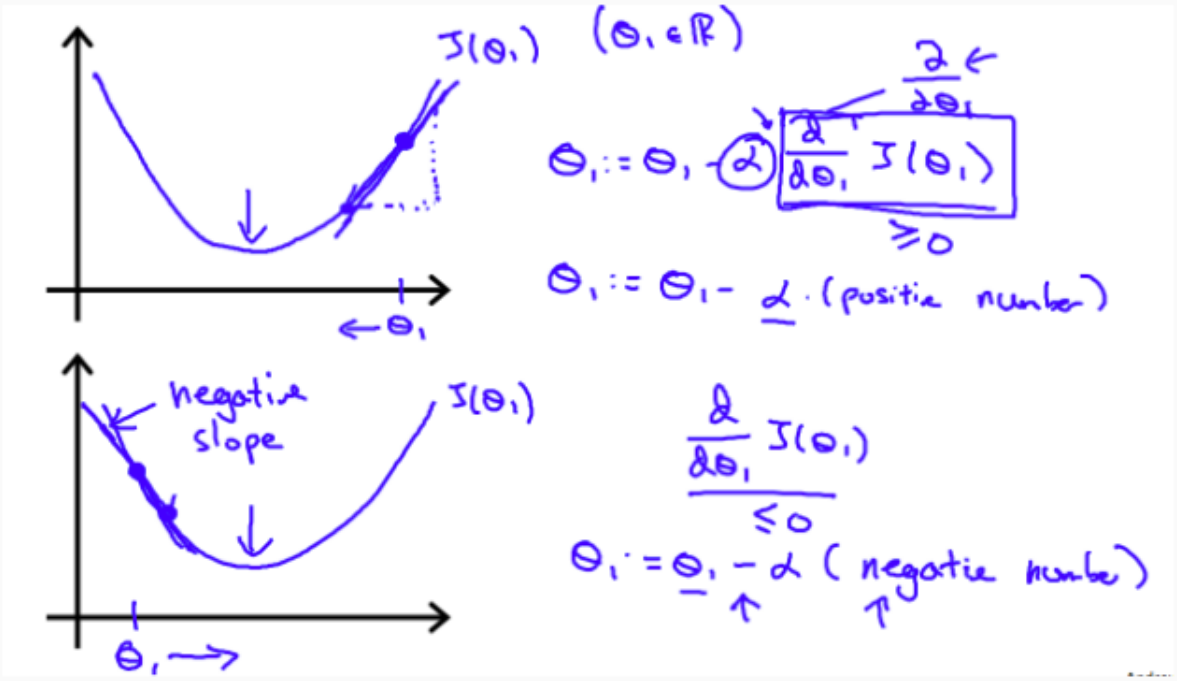


Figure 2.3: Different signs of slope for partial derivative of J.

– Partial derivation of $J(\theta_0, \theta_1)$ is the following (both results on the right side)[1]:

$$for\ j = 0: \qquad \frac{\partial}{\partial \theta_{\mathbf{0}}} J(\theta_0, \theta_1) = \frac{1}{m} * \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$for\ j = 1: \qquad \frac{\partial}{\partial \theta_{\mathbf{1}}} J(\theta_0, \theta_1) = \frac{1}{m} * \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) * \boldsymbol{x^{(i)}}$$

- Gradient descent here is a special case of one with multivariate linear regression, see Figure 2.4.

[1] https://www.coursera.org/learn/machine-learning/supplement/U90DX/gradient-descent-for-linear-regression

Multivariate Linear Regression

- **Hypothesis** has more variables (features, let's say n is their amount):

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n \tag{2.3}$$

- **Cost function** is almost the same as for univariate linear regression:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.4}$$

- **Cost function can be regularized** as follows (and this is the same for univariate linear regression):

$$J(\theta) = \frac{1}{2m} [\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2] \tag{2.5}$$

  - Cost function will be always convex here, so no matter if we use regularization or not, gradient descent will still converge to the global minimum.

- **Gradient descent** is as follows:

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)}$$

...

}

In other words:

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j := 0...n}$$

}

Figure 2.4: Gradient descent for multivariate linear regression

2.2 Polynomial Regression

- **Polynomial regression is a special case of linear regression.**
- One common pattern within machine learning is to use linear models trained on nonlinear functions of the data. This approach maintains the generally fast performance of linear methods, while allowing them to fit a much wider range of data.
- For example, **a simple linear regression can be extended by constructing polynomial features from the coefficients**.
- Sometimes we want to combine multiple features and new feature is created. Then we may want to use polynomial regression to learn from such modified data.
- TL;DR - it is a "feature" how to create new matrix of polynomial features, it is not "model" itself.
- We can use it if our hypothesis function need not be linear. We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).
- An example of hypothesis function, cubic parameters $n = 3$):

$$h_\theta(x) = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \theta_n x^n \tag{2.6}$$

- One important thing to keep in mind is, if you choose your features this way then **feature scaling becomes very important**.

2.3 Logistic Regression

Deals with classification problems, no regression as the name tells us. This model maps random real number to interval of discrete values $[0, 1]$ (probability).

- Feature scaling is good to have as well - faster run.
- For non-linear decision boundary - we can also use polynomial regression "trick", as in the case of linear regression.
- **Hypothesis**:

$$h_\theta(x) = g(\theta^T x) \tag{2.7}$$

where

  - $g(z) = \frac{1}{1+e^{-z}}$ is sigmoid / logistic function, where z is a real number.

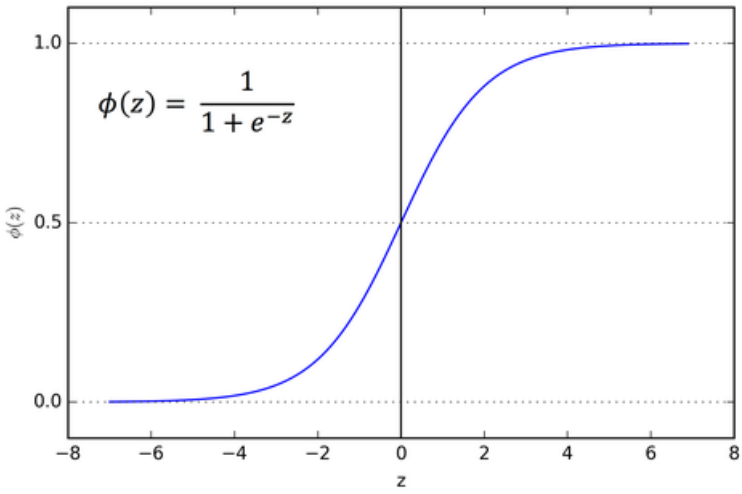


Figure 2.5: Sigmoid function

  - $h_\theta(x)$ gives us a probability, that the output is equal to 1 on input x.
    - For example, if $h_\theta(x) = 0.7$ on some input x, that means that the probability of positive output (label '1') on x is 70%.
    - So, mathematically: $h_\theta(x) = p(y = 1|x; \theta)$ (probability that y=1, given x, parametrized by θ).
  - $h_\theta(x)$ is always in **range [0, 1], because of sigmoid function.**
    - $h_\theta(x) \geq 0.5 \ldots y = 1$, so $\theta^T * x \geq 0 \Rightarrow y = 1$

  - $h_\theta(x) < 0.5 \ldots y = 0$, so $\theta^T * x < 0 \Rightarrow y = 0$
- Without function g, the hypothesis is same as in linear regression.
- So the hypothesis function is basically the following:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T * x}} \tag{2.8}$$

- **Cost function**
  - Is always greater or equal zero.
  - Is computed by using logarithms. It cannot be the same like in the case of linear regression, because the cost function would be non-convex (because of sigmoid function) = many local minima. We want convex function, with 1 local (=global) minima.
  - **In linear regression**, we minimized the empirical risk, defined as the average squared error loss, also known as **mean squared error**, or MSE. **In logistic regression**, we maximize the likelihood of our training set according to the model. In statistics, the **likelihood function** defines how likely the observation (a simple example) is according to our model. So the optimization criterion in logistic regression is called **maximum likelihood**. So instead of minimizing the average loss, we now maximize the likelihood of the training data according to our model. For a complete definition, please follow the mathematical equations below. It is basically $\prod_{i=1 \ldots N} h_\theta(x)^{y_i}(1-h_\theta(x))^{(1-y_i)}$- which is just that it results in h_θ if $y_i = 1$ and $1 - h_\theta$ otherwise. There is the product operator $\prod$ in objective function instead of sum operator $\sum$ which was used in linear regression. This is because the likelihood of observing N labels for N examples is the product of likelihoods of each observation = assuming that all observations are independent of one another, which is the case. In practice, because of *exp* function used in the model, it is more convenient to maximize the **log-likelihood** instead of **likelihood**: $\sum_{i=1 \ldots N}[y_i \, ln \, h_\theta(x) + (1 - y_i) \, ln \, h_\theta(x))]$. Because ln **is strictly increasing function**, maximizing this function is the same as maximizing its argument, and the solution to this new optimization problem is the same as the solution to the original problem. Typical optimization procedure here is **gradient descent**.
  - **Definition**:

    $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$

    where

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & y = 1 \\ -log(1 - h_\theta(x)) & y = 0 \end{cases}$$

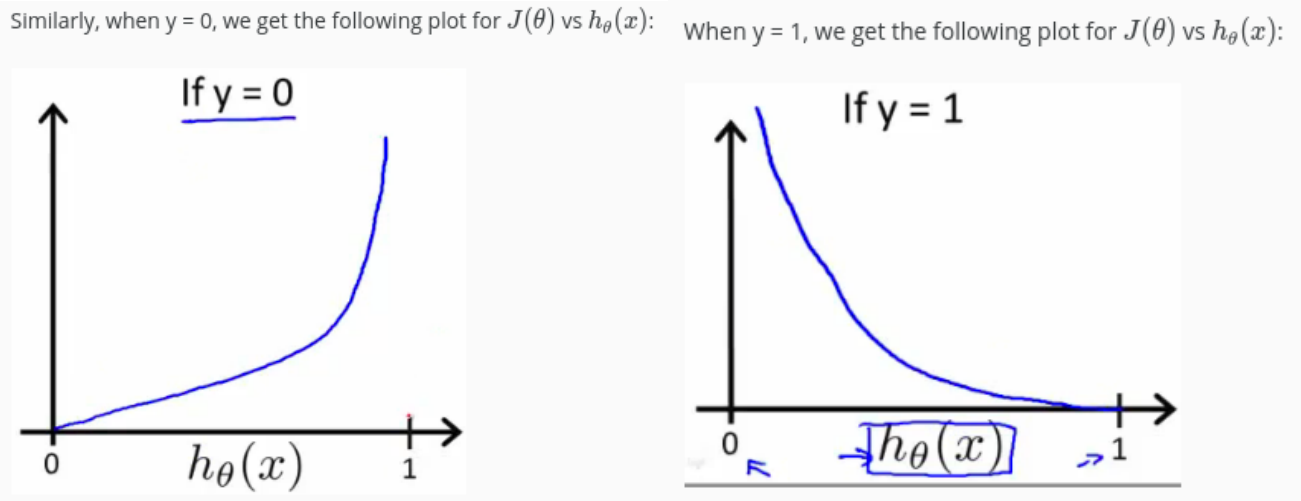


Figure 2.6: Cost function of Logistic Regression for y=0 and y=1.

– Some observations:

$$Cost(h_\theta(x), y) = 0 \quad if\ h_\theta(x) = y \tag{2.9}$$

$$Cost(h_\theta(x), y) \to \infty \begin{cases} if\ y = 0, & h_\theta(x) \to 1,\ or \\ if\ y = 1, & h_\theta(x) \to 0 \end{cases} \tag{2.10}$$

– So as a result, cost function can be re-written as follows:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} * log(h_\vartheta(x^{(i)})) + (1 - y^{(i)}) * (log(1 - h_\theta(x^{(i)}))] \tag{2.11}$$

– Another view on cost function for logistic regression is in figure below.

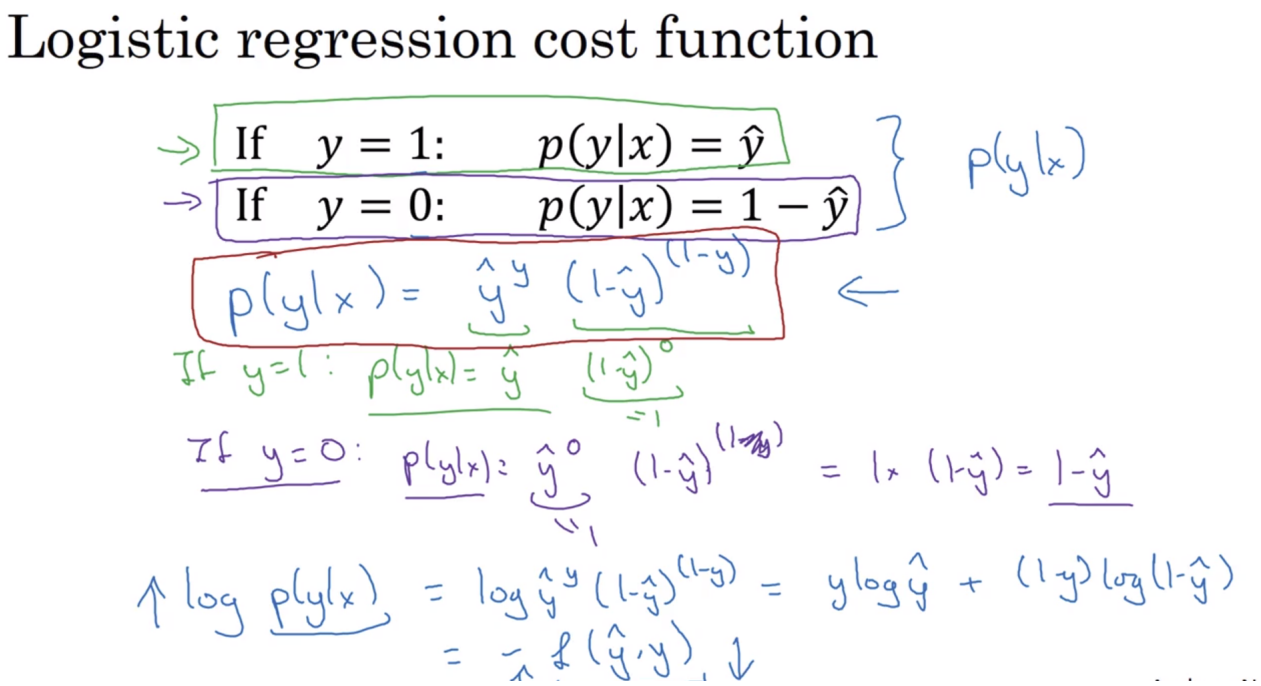


Figure 2.7: Cost function calculation for logistic regression. Minus sign in the end is there because we want to minimize the loss function. This is what the loss function on a single example looks like.

- **Cost function with regularization**:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} * log(h_\vartheta(x^{(i)})) + (1-y^{(i)}) * (log(1-h_\theta(x^{(i)}))] + \boldsymbol{\lambda}\frac{\mathbf{1}}{\mathbf{2m}}\sum_{\boldsymbol{j=1}}^{\boldsymbol{n}}\boldsymbol{\theta_j^2} \quad (2.12)$$

  - Where the second sum means to explicitly exclude the bias term θ_0(j is going from 1).
  - Cost function here will be always convex, so no matter if we use regularization or not, gradient descent will still converge to the global minimum.

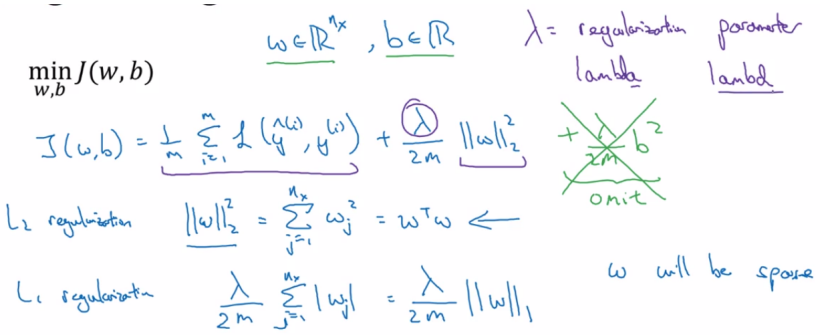


Figure 2.8: Logistic regression with L1 and L2 regularization.

- **Gradient descent** is the same as in linear regression. Only hypothesis has changed to sigmoid.
  - For the completeness, following is **gradient descent** (after the step of derivation) with **regularization**:

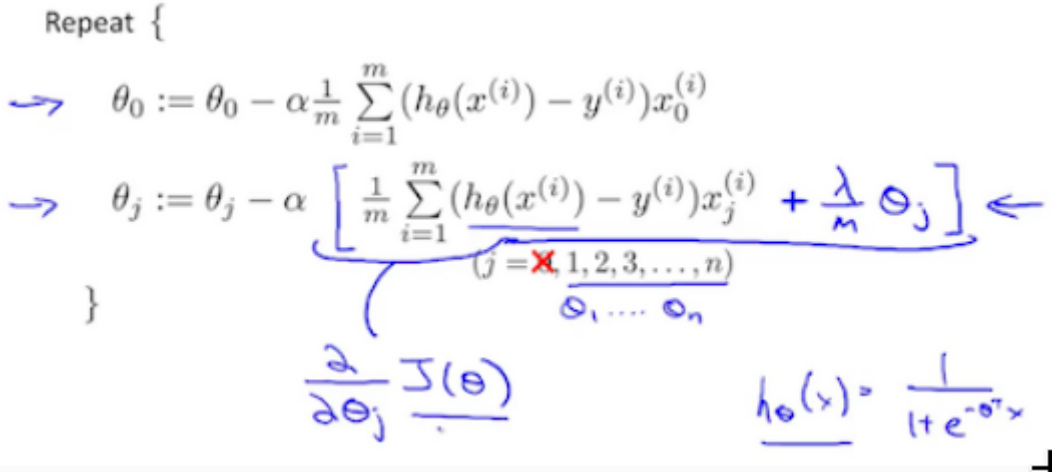


Figure 2.9: Gradient descent of regularized Logistic Regression (after derivation).

3 Support Vector Machines

- Supervised learning technique, for binary classification problems.
- Linear classifier, but we can use "kernel trick" - map our data to higher-dimensional space and then they can be more easily separated. The most real-world problems involve non-separable data for which no hyperplane exists that successfully separates the positive from negative instances in the training set. One solution to the inseparability problem is to map the data onto a higher- dimensional space and define a separating hyperplane there. This higher-dimensional space is called the transformed feature space, as opposed to the input space occupied by the training instances. With an appropriately chosen transformed feature space of sufficient dimensionality, any consistent training set can be made separable. A linear separation in transformed feature space corresponds to a non-linear separation in the original input space.
- SVMs are basically just a clever reincarnation of Perceptrons, with Kernel trick. They expand the input to a very large layer of non-linear non-adaptive features. They only have one layer of adaptive weights (from the features to decision units). They have very efficient way of fitting the weights that control overfitting. They have a clever solution of simultaneously doing feature selection and finding weights on the remaining features.
- SVM requires labels to be $+1$ and -1 (TBD: is that true??).
- Distance between hyperplanes is given by $\frac{2}{||w||}$ where $||w||$ is Euclidean norm of w, given by $\sqrt{\sum_{j=1}^{D}(w_j)^2}$, where D is dimension of w, i.e. number of dimensions in feature vector, and $wx - b = 0$ is the decision boundary.
- Something like logistic regression, but a little change in hypothesis. **Actually, LR is something like SVM with linear kernel. It depends on the implementation, some can perform slightly better, but very similar. They are different only in the missing sign operator in SVM. And the hyperplane in the SVM plays the role of the decision boundary - it's used to separate 2 groups of examples from one another, such it has to be as far from each group as possible. On the other hand, in linear regression, the hyperplane[1] is chosen to all training examples as possible. See**

[1] Actually, this decision boundary terminology depends on a number of dimensions of the input features. If a sample from a training dataset has just 1 feature, we are talking about a line, if 2 features, then it is a plane, and if 3 or more features, then it is a hyperplane.

Section 2.1 for more details. Tips what to use:

- # features > # training samples = (10,000 vs 10-1,000), use logistic regression or SVM without kernel. Because we have so many features and we don't have much data, linear function will probably do fine. We do not have enough data to fit very complicated non-linear function.
- # features < # training samples
  - # features is relatively small (up to 1,000) and # training samples is not extremely high (up to 10,000) = use SVM with (Gaussian) kernel.
  - # features is relatively small (up to 1,000) and # training samples is high (like 50,000 or 1 million or more) = maybe add more features and then logistic regression or SVM without kernel.
- Note: ANN works with all three situations well, but can be slower to learn.

- We can understand SVM from Logistic regression. The following figures capture what is needed to change in Logistic regression so that we have SVM.

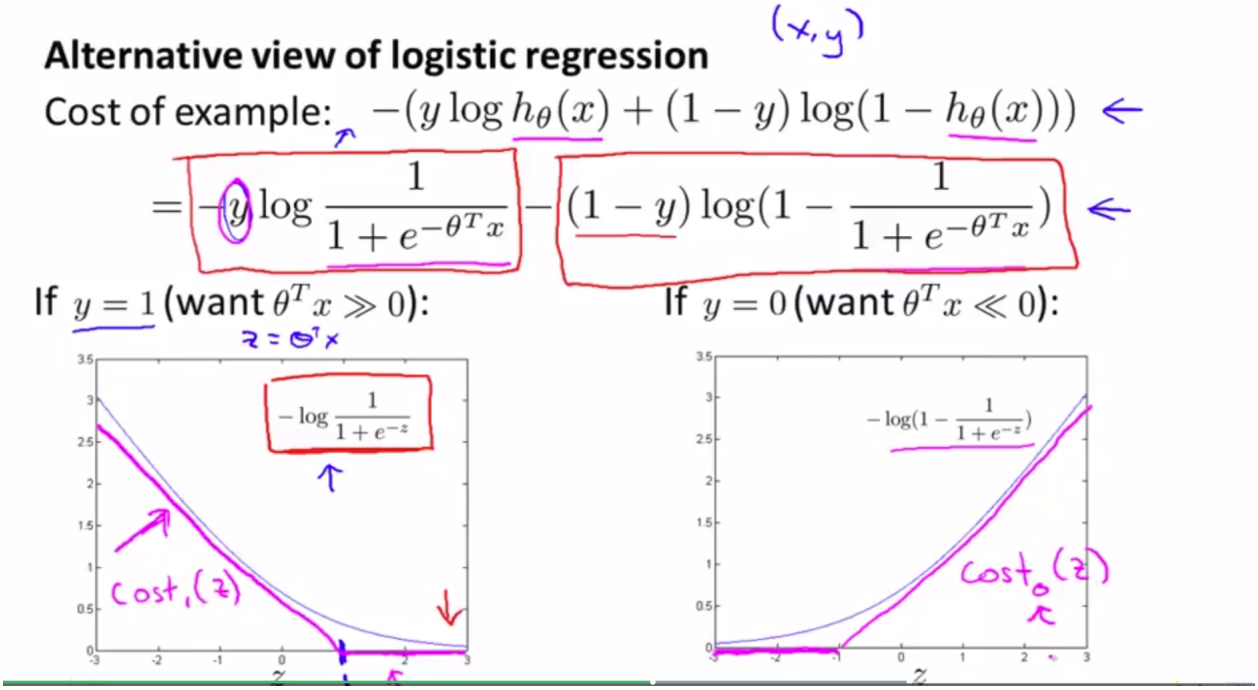


Figure 3.1: This figure plots cost function of logistic regression (and its modified version to get intuition about SVM) for 1 sample from dataset. In SVM, if we plot its cost function, instead of logistic regression curves (blue ones) we want to have slightly different (pink ones). $cost_0(z)$ just describes a situation if $y = 0$ (so z is much bigger than 0 - if you want to see this, plot sigmoid function - much bigger than 0 means almost 1 on sigmoid), analogically for $cost_1(z)$. The new cost function for SVM will be flat for some values (on the left plot see from -1 to 3, and then straight line from the point of where $z = -1$. And analogically for the right plot. This gives SVM some computational advantage, for an easier optimization problem that would be easier to solve.

Consider the following minimization problems:

1. $\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$
2. $\min_\theta C\left[\sum_{i=1}^{m} y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$

Figure 3.2: These equations show, that we can mathematically transform the first cost function to the second cost function. The first one is ust a classic cost function, used for example in logistic regression (plus regularization). Just a detail, there is no minus sign on the beginning - its because it is inside of cost function (just multiplied by -1, no big magic it's the same as in logistic regression in Section 2.3). So, we can multiply the whole equation by m and divide by regularization parameter λ, and we got C - so this is the first step from logistic regression to SVM. These 2 optimization problems will give the same value of θ (=same value of θ gives the optimal solution for both problems) if $C = \frac{1}{\lambda}$.

- **The choice of hyper-parameter C** ($\frac{1}{\lambda}$) - usually chosen as the best result on cross-validation data (and σ hyper-parameter in Gaussian kernel function is also chosen usually with CV):
  - very large C (low λ) = lower bias, high variance - overfitting. Not good for outliers as well. SVM will try to find the largest margin by completely ignoring misclassification.
  - small C (big λ) = the opposite, higher bias, lower variance - underfitting. Making classification errors is more costly, so SVM tries to make fewer mistakes by sacrificing the margin size. Larger margin is better for generalization.

  Note: **A linearly separable dataset can usually be separated by many different lines.** Varying the parameter C will cause the SVM's decision boundary to vary among these possibilities. For example, for a very large value of C , it might learn larger values of θ in order to increase the margin on certain examples.

  So, C regulates the trade-off between classifying the training data well and classifying future examples well (generalization).

- **Penalty hyperparameter** can be also used, for misclassification of training example of specific classes.

- SVM is sometimes called as Large margin classifier, because SVM separates classes as most as possible by a margin, see the following figure.

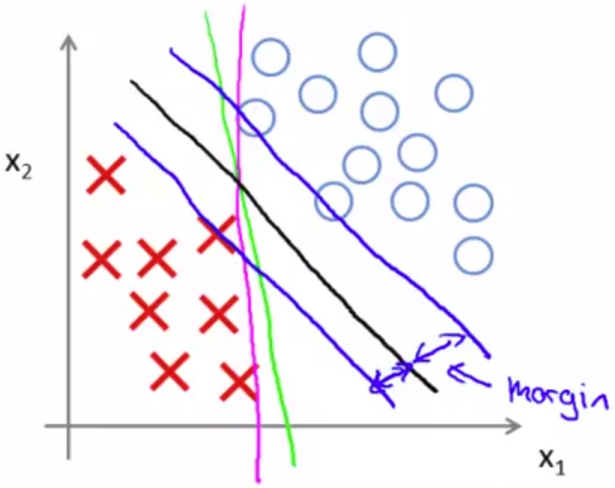


Figure 3.3: An example of decision boundary of SVM classifier for 2 classes which are linearly separable. It will chose black line, instead of green or purple, which also separate the classes - but the black one is with the biggest margin (from both sides).

– Why? For a brief mathematical intuition, let's consider vector inner product (length of vector and so on) on the following figure.

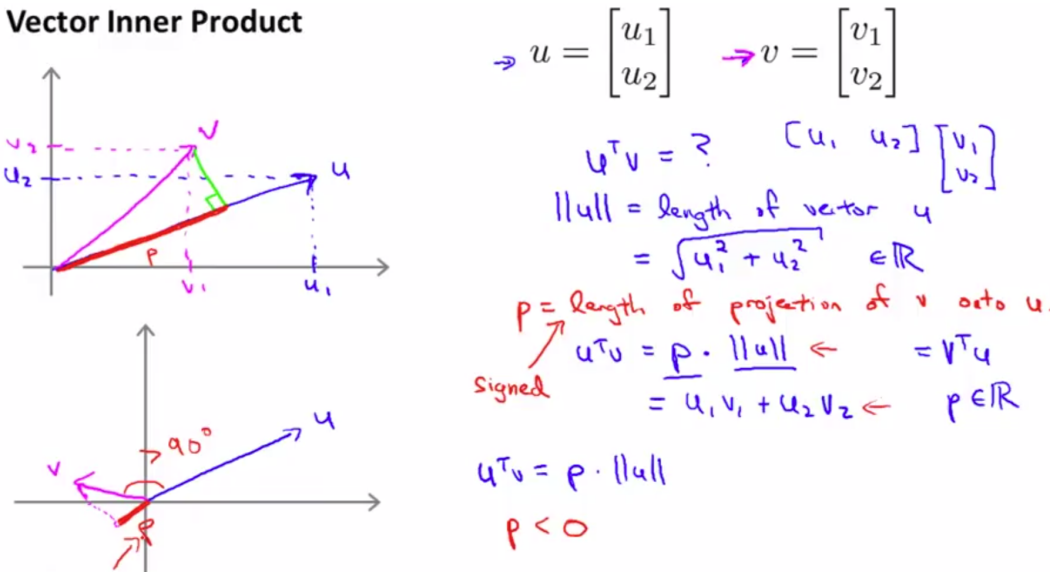


Figure 3.4: Vector inner product explanation.

– And now, transform this into SVM.

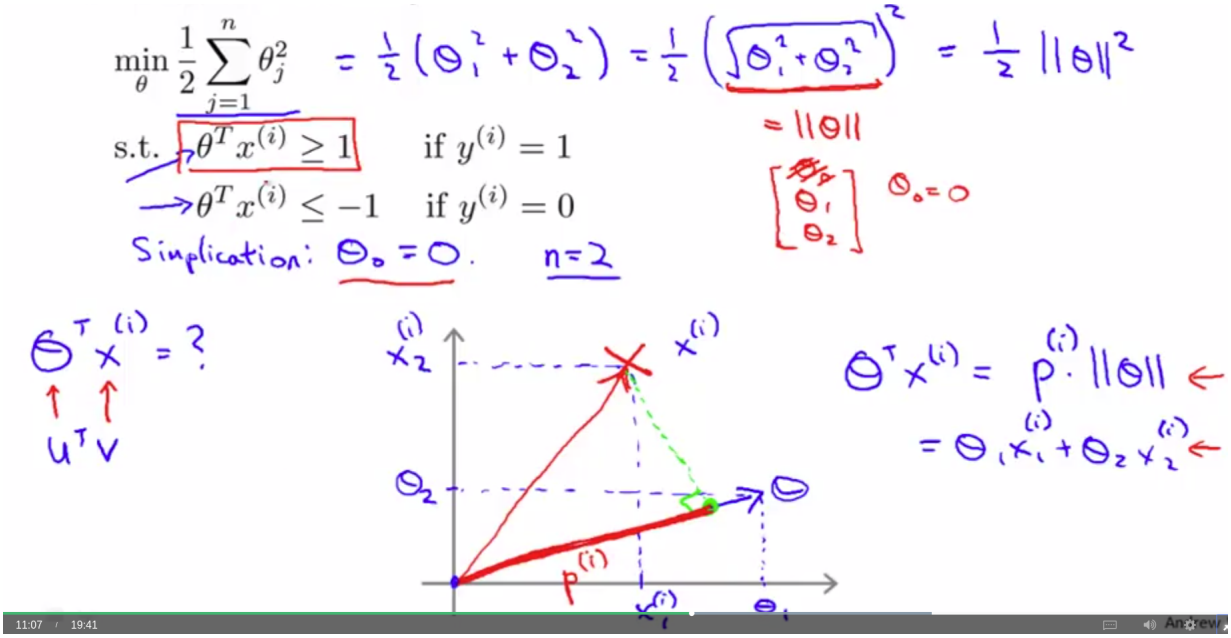


Figure 3.5: SVM decision boundary explained.

– Further, an illustration in the following 2 examples.

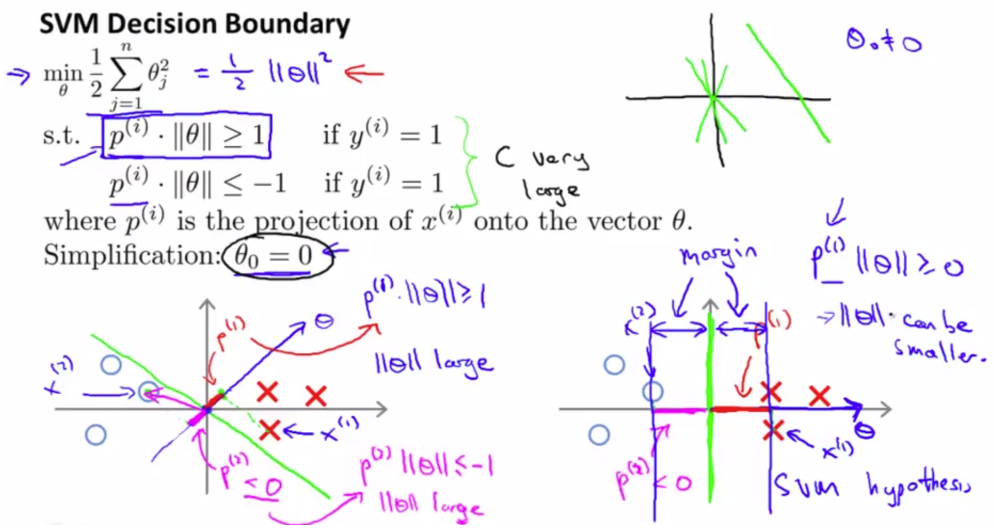
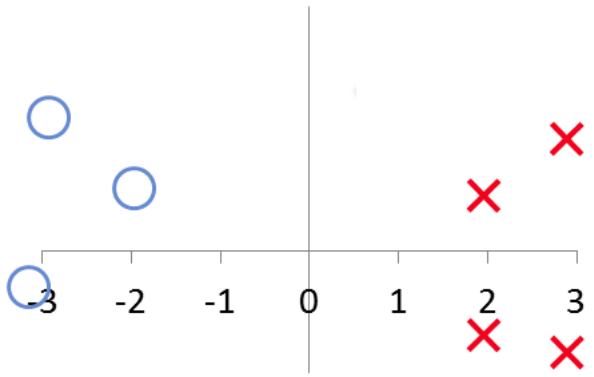


Figure 3.6: An example of SVM decision boundary.

The SVM optimization problem we used is:

$$\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

$$\text{s.t. } \|\theta\| \cdot p^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$$
$$\|\theta\| \cdot p^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$$



where $p^{(i)}$ is the (signed - positive or negative) projection of $x^{(i)}$ onto θ. Consider the training set above. At the optimal value of θ, what is $\|\theta\|$?

Figure 3.7: Another example of SVM decision boundary. Correct answer is that $||\theta|| = 1/2$.

- The algorithm uses so called landmarks, what are some points around which the learning is concentrated.
- It uses trick, which can deal with infinite number of features. SVMs are well suited to deal with learning tasks where the number of features is large with respect to the number of training instances. The model complexity of an SVM is unaffected by the number of features encountered in the training data (the number of support vectors selected by the SVM learning algorithm is usually small).
- To extend SVM to cases in which the data is not linearly separable, we have to introduce the **hinge loss function**: $max(0, 1 - y_i(wx_i - b))$. If wx_ilies on the correct side of the decision boundary, the hinge loss function is 0. For data on the wrong side of the decision boundary, the function's value is proportional to the distance from the decision boundary. So then we wish to minimize the following cost function: $C||w||^2 + \frac{1}{N}\sum_{i=1}^{N} max(0, 1 - y_i(wx_i - b))$, where the hyperparameter C determines the trade-off between increasing the size of the decision boundary and ensuring that each $x_i$lies on the correct side of the decision boundary. Minimizing $||w||$ is equivalent to minimizing $\frac{1}{2}||w||^2$and the latter term makes it possible to

perform quadratic programming optimization later on. SVMs that optimize hinge loss are called **soft-margin SVMs**, and original formulation is referred to as **hard-margin SVM**.

3.1 Kernels

A trick used in SVM (and other models as well) to make the model to be a non-linear classifier. SVM can be adapted to work with datasets that cannot be separated by a hyperplane in its original space. If we manage to transform the original space into a space of higher dimensionality, we could hope that the examples will become linearly separable in this transformed space. **Kernel trick - using a function (kernel) to implicitly transform (map) the original space into a higher dimensional space during the cost function optimization.** However, we don't know a priori which mapping will work for our data. This is why kernel functions (also known as kernels) are useful, because they do not perform this transformation explicitly.

- As an example, let's consider a hypothesis to be:

$$h_\theta(x) = \begin{cases} 1 & if\ \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 ... \geq 0 \\ 0 & otherwise \end{cases}$$

- So basically this is what we got already for **polynomial regression** (see the beginning of Chapter 2 and Section 2.2). Such higher order polynomial is a one way to come up with more features. But it has some disadvantages like it is computationally expensive.
- **There is another way how to create new features**. Let's have now a new notation: $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + ...$, where $f_1 = x_1$,$f_2 = x_2$,$f_3 = x_1 x_2$, ...
- f_i can be calculated as for example for given x: $f_1 = similarity(x, l^{(1)})$, where $l^{(i)}$is so called landmark. A new feature $f_i$then depends on proximity to landmarks. This *similarity* **(=kernel function, in this case Gaussian kernel)** can be calculated as follows (for a given i, which is from 1 to number of examples in dataset):

$$f_i = exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2}) = exp(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(i)})}{2\sigma^2}) \tag{3.1}$$

  Explanation
  - The dividend (numerator) is just squared (to have non negative numbers) Euclidean distance between point x and a landmark. This can be also written as $k = (x, l^{(i)})$.
  - If x is close to one of the landmarks - the dividend is close to 0, so the whole fraction is close to 0, so the result is 1 (like yes, they are similar).
  - If $x$is far away from one of the landmarks - the dividend is some big number (even squared), so the result is close to 0.

- Each landmark defines a new feature.
- Parameter sigma determines the slope of similarity function, see on the following plot. Higher sigma means higher bias (features f_i vary more smoothly), lower variance. Lower sigma means lower bias, higher variance (features f_i vary less smoothly).
- So, if you are overfitting, it would be good to decrease C and increase σ^2.

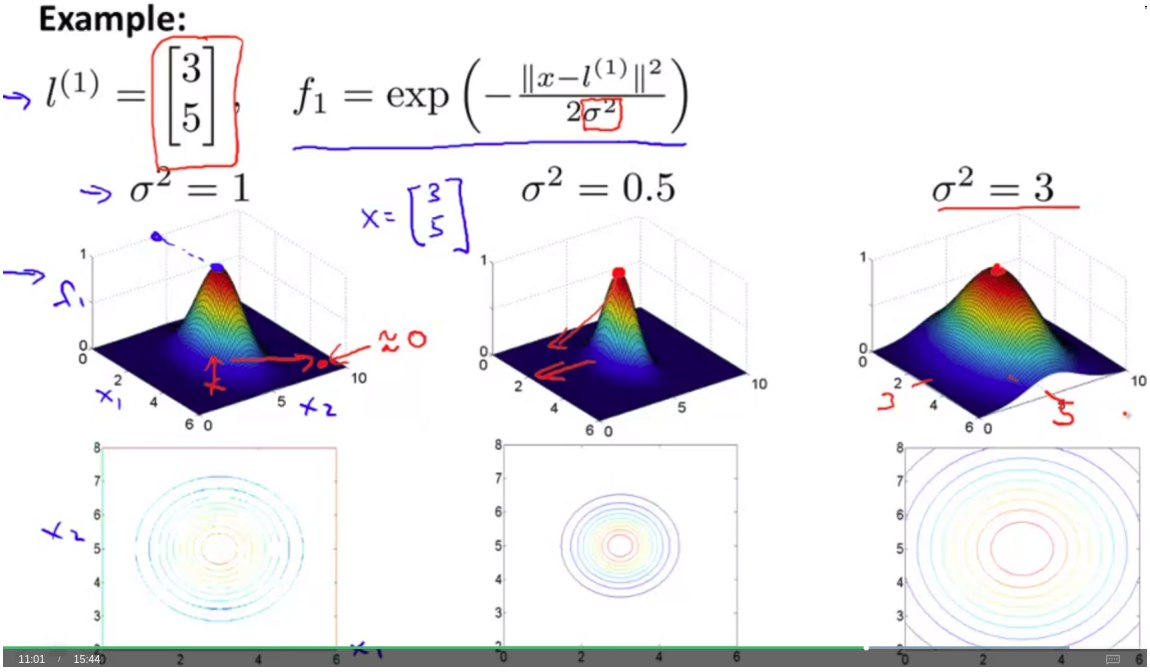


Figure 3.8: An example of different values for σ parameter. Landmark point is on the top of graph, and we can see how the different sigma values influence a similarity growth of landmark with respect to x.

Prediction

- Is done by using landmarks, sigma parameter, theta parameters, and similarity function, which displays the following figure.

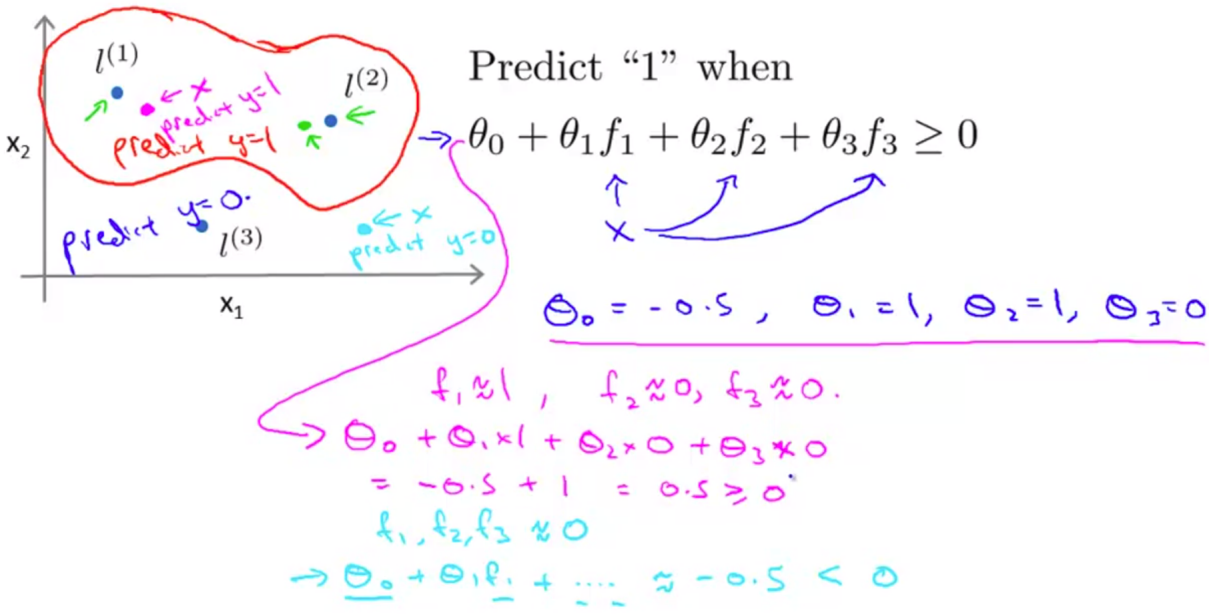


Figure 3.9: An example of prediction using gaussian kernel and 3 landmarks. More far away a given point is from some landmark, the more likely it will be classified as 0. If a given point is nearby to some landmark, it is classified as 1.

Landmarks

– We put 1 landmark for each training sample. So each landmark is on the same position as a given training sample.

Choice of kernel (similarity function)

Kernel function has to satisfy condition called "Mercer's Theorem" to make SVM runs correctly and don't diverge. Not all similarity functions make valid kernels. By the way, Andrew Ng said that he uses mostly the first 4 types of kernels (the other types are very rarely used). BTW, why we do not apply kernel trick to some other algorithms like logistic regression? You can, and then you would have new features, but we don't do it because computational trick we do in SVM with kernels do not generalize well to other algorithms. SVM and kernels tend to go well together.

- **Linear kernel** (=no kernel) - when we have a lot of features and too little data. Using non-linear kernel would probably cause overfitting.

  predict $y = 1$ if $\theta^T x \geq 0$

- **Gaussian kernel** (also known as RBF kernel - radial basis function kernel - where the usage of parameter σ^2 which needs to be chosen) - in this one it is recommended to use feature scaling in the first step, so that all features have the same "weight"

and some are not being ignored because are smaller, and some other are bigger and are more dominant. The similarity measure used by the Gaussian kernel expects that the data lie in approximately the same range.

$$f_i = exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2}),\ where\ l^{(i)} = x^{(i)}$$

If $\sigma^2$is large, we have higher bias, lower variance classifier. Analogically if it is small. This parameter determines bias-variance trade-off. Term $||x - l^{(i)}||^2$ is called Euclidean distance between two feature vectors, and is given by the following equation: $d(x, y) = \sqrt{\sum_{j=1}^{D}(x^{(j)} - y^{(j)})^2}$.

It can be shown, that the feature space of this kernel has an infinite number of dimensions. By varying the hyperparameter σ, we can choose between getting a smooth or curvy decision boundary in the original space.

- **Polynomial kernel** - it has 2 parameters. What parameter to add, and a degree of the polynomial. This kernel is not used that much and performs usually worse than Gaussian kernel. If the degree is 2, we are talking about quadratic kernel.

$$k(x, l) = (x^T l + constanttoadd)^{degree}$$

- **String kernel** - sometimes used if the input data are text strings.
- **Chi-square kernel**
- **Histogram kernel**
- **Intersection kernel**

4 Artificial Neural Networks

- Models inspired by the structure and function of biological neural networks.
- In each hemisphere of our brain, we have a primary visual cortex V_1, containing 140M of neurons, with tens of billions of connections between them. And there are entire series of visual cortices - V_2, V_3, V_4, and V_5- doing progressively more complex image processing.
- State of the art technique for many modern applications. More will be in Chapter 5 later. Difference between Artificial Neural Networks and Deep Neural Networks? Probably that Deep NN have 2 or more hidden layers.
- These models can handle non-linear problems.
- Neural networks even with just a single hidden layer can be used **to approximate any continuous function to any desired precision**.[1]
- It is not robust to missing values, ANN requires complete records to do their work.
- **Neural networks can be feedforward, or recurrent.** In this chapter, we will discuss the first one, feedforward. They don't have any loops in their graph and can be organized in layers. More on this topic please see Chapter 5.
- Shallow network (1 hidden layer) might need to be very big; possibly much bigger than a deep network to have similar accuracy. This is based on a number of papers proving that shallow networks would in some cases need exponentially many neurons. From a previous research:[2] "In this paper we provide empirical evidence that shallow nets are capable of learning the same function as deep nets, and in some cases with the same number of parameters as the deep nets." However, this has been never fully proven. Shallow networks do not have the same accuracy and we can only guess why. Some ideas:
  - Maybe a shallow network would need more neurons then the deep one?
  - Maybe a shallow network is more difficult to train with our current algorithms (e.g. it has more nasty local minima, or the convergence rate is slower, or whatever)?

[1] http://neuralnetworksanddeeplearning.com/chap4.html
[2] https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w

  - Maybe a shallow architecture does not fit to the kind of problems we are usually trying to solve (e.g. object recognition is a quintessential "deep", hierarchical process)?

- It was proven, that neural network with 1 hidden layer which contain in theory unlimited number of neurons, can solve any function with any desired precision (it is approximation, it cannot compute any function exactly) = **universality theorem**. We cannot say, that deeper networks are better for every problem (however, this depends on the amount of data, regularization, etc...)! However, more deep nets can catch very complex relations. Also, a given problem is abstracted into more details with deeper network. However, back to universality theorem, not that neural networks just approximate some function, they cannot universally approximate discontinuous functions (that make sharp, sudden jumps). Neural networks compute continuous functions of their input. But in practice this is not usually an important limitation, since even if a function we would really like to compute is discontinuous, it's often the case that a continuous approximation is good enough.

- Universality theorem - tells us, that we can use only 1 hidden layer. So why we are even using deep networks? The reasons are practical. Deep nets have a hierarchical structure, which makes them particularly well adapted to learn the hierarchies of knowledge that seem to be useful in solving real-world problems. Deeper nets do a better job than shallow networks at learning hierarchies of knowledge.

- NN are Turing-complete. Also, it is possible to create an implementation of Turing machine with ANN.

- Neural Networks were designed to simulate neurons in the brain. However, in brain we have 13-15 milliards of neurons, each one can be connected (not directly, there is 20nm gap) with 5k other neurons. Neuron has:
  - a cell body (Nucleus),
  - input wires (Dendrides), and
  - output wire (Axon).

  Neuron sends signals to other neurons with Axon, and the other neuron accepts incoming message via Dendrides. Neurons communicate with electric signals (spikes). In a neural network with positive inputs, positive weights are most often used to exhibit other neurons and negative weights are most often used to inhibit other neurons. **Synapses adapt**, and that is the most important property.

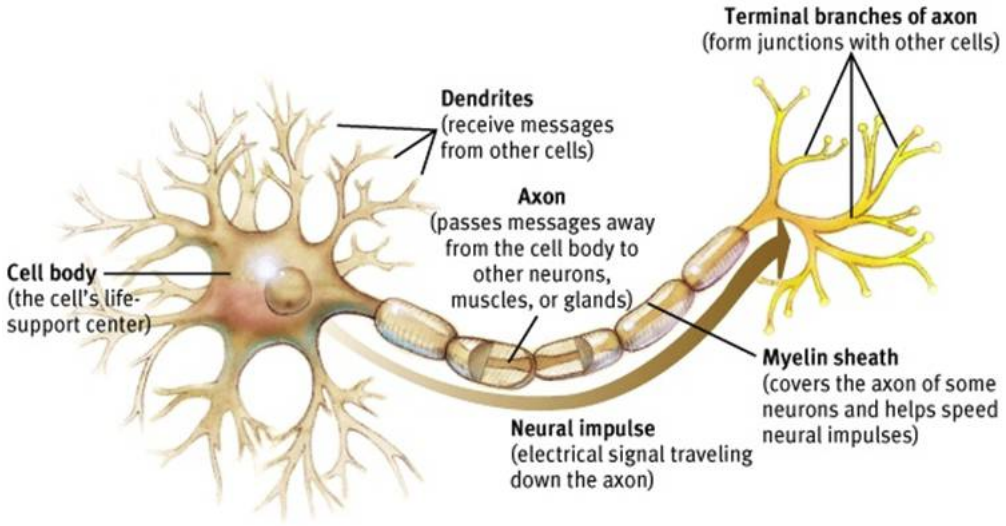


Figure 4.1: A simple neuron scheme.

- How the brain works:
  - Each neuron receives inputs from other neurons. A few neurons also connect to receptors (it's a large number of neurons, but only a fraction of the all neurons). Cortical neurons use spikes to communicate.
  - The effect of each input line on the neuron is controlled by a synaptic weight. This weight can be positive or negative.
  - The synaptic weights adapt so that the whole network learns to perform useful computations.

  You have about 10^{11}neurons, each with about 10^4 weights.

  Cortex is made of general purpose stuff that has the ability to turn into special purpose HW in response to experience. In fact, cortex looks pretty much the same all over. It has rapid parallel computation (once you learned) and flexibility (ability to compute new functions).

History

- 1943 - Warren McCulloch and Watler Pittse created a simple mathematical model of artificial neural cell. Parameters of this model had mostly bipolar values. They proved, that with simple neural networks, it is possible to perform calculation of any logical and arithmetical function.
- 1949 - Donal Hebb - The Organization of Behavior - learning algorithm for synapses of neurons. It took almost 10 years to researchers to find application for this area.

- 1957 - Frank Rosenblatt - model from 1943 (McCulloch and Pittse) has been generalized for real numbers. The author defined Perceptron model and learning algorithm for Perceptron. Defined learning algorithm could find weight vector of parameters, based on training data.
- 1969 - Monsky and Papert - published a book called "Perceptrons" that analyzed what they could do and showed their limitations. Many people thought these limitation applied to all neural network models. These 2 researchers thought (and actually the whole community of researchers regarding this topic) that they proven that Perceptron is not good. However, Perceptron learning algorithm is still being used. For example for tasks with enormous feature vectors that contain many millions of features.
- 1986 - David Rumelhartr, Geoffrey Hinton, and Ronald Williams - backpropagation. This algorithm is the most popular even today (2019) for supervised learning of multi-layer Perceptron.
- 1988 - Teuvo Kohonen - Self-Organizing Map. Unsupervised method, which is based on competition between parallel neurons. Neurons from themselves change their inner state and behavior.

Initialization of weights

Initialization of weights (=thetas) cannot be all zeroes! The best strategy is to use **random initialization** and ideally all the values are different, somewhere close to zero. If there are **only zeroes**, it would mean that **all hidden units are computing the same function of the input**. This is called the problem of **symmetric ways** (weights are being the same). And therefore also deltas are same (see later below). **Randomness** performs **symmetry breaking**. If we generate 1 random number and all weights would have this value, then this is again, the problem of symmetric ways (every unit in the network will get the same update after backpropagation).

- It is absolutely fine to initialize the weights for logistic regression to zeros, but in neural network, such approach would not work with gradient descent (actually, it is not only because of number 0, but because when all neurons have the same number, then they will compute the same function). However, to initialize a bias term to zero is fine.
- Initialization with Gaussian distribution: if the weights in hidden layers are initialized using normalized Gaussian, then activations will often be very close to 0 or 1, and learning will proceed very slowly (saturation of hidden neurons problem). This happens when standard deviation is too big. So it is good to normalize it, according to the number of input weights to a given hidden neuron. For example, $1/\sqrt{n_{in}}$, where $n_{in}$is the number of input weights to a given hidden neuron. Then, $z = \sum_j w_j x_j + b$ will be a Gaussian random variable with mean 0 and if for example, $j = 500$, so there are 500 input neurons, then standard deviation would be

$\sqrt{3/2} = 1.22$.[3] If we would not do this normalization $1/\sqrt{n_{in}}$, then standard deviation would be $\sqrt{501} = 22.4$, which is Gaussian distribution not sharply peaked at all, so z would be pretty large (probably much bigger than 1 or much lower than -1), and output of $\sigma(z)$ from the hidden neuron would be very close to either 1 or 0 - which would means that it would be saturated.

- One effective strategy for random initialization is to randomly select values for $\theta(l)$ uniformly in the range $[-\varepsilon_{init}, \varepsilon_{init}]$. This range of values ensures that the parameters are kept small and makes the learning more efficient. One effective strategy for choosing ε_{init} is to base it on the **number of units in the network**. A good choice is

$$\varepsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}, \tag{4.1}$$

  - where $L_{in} = s_l$ and $L_{out} = s_{l+1}$are in out the number of units in the layers adjacent to $\theta^{(l)}$ so, for first θ (after the input layer), it L_{in} is a size of the input layer, and L_{out} is a size of a hidden layer which goes right after the input layer.

- Once we have ε_{init}, then we can calculate a random weight for a given theta in the following way:

$$rand(L_{out}, 1 + L_{in}) * 2 * \varepsilon_{init} - \varepsilon_{init} \tag{4.2}$$

- So, different hidden layers have different range of pseudo-random numbers, and there is a good variety if we are using uniform distribution of pseud-random generator.

- If we would initialize weights randomly with great values (± 10 or even bigger range), then the cost will start very high. If you would If you train ANN longer you will see better results, but initializing with overly large random numbers slows down the optimization.

- There are 2 known and effective approaches - you randomly generate real number (in the following formulas it is named as $generated\,num$) with Gaussian distribution with $mean = 0$ and $variance = 1$ and then multiply this number with another number:

  - **Xavier initialization**:

$$generated\,num * \sqrt{\frac{1}{layers\,dims[l-1]}} \tag{4.3}$$

[3] https://stats.stackexchange.com/questions/194082/how-to-find-the-variance-in-this-neural-network-related-question

– **He initialization** (2015) is very similar to Xavier initialization and works well with ANN with ReLU activations:

$$generated\,num * \sqrt{\frac{2}{layers\,dims[l-1]}} \tag{4.4}$$

Basics

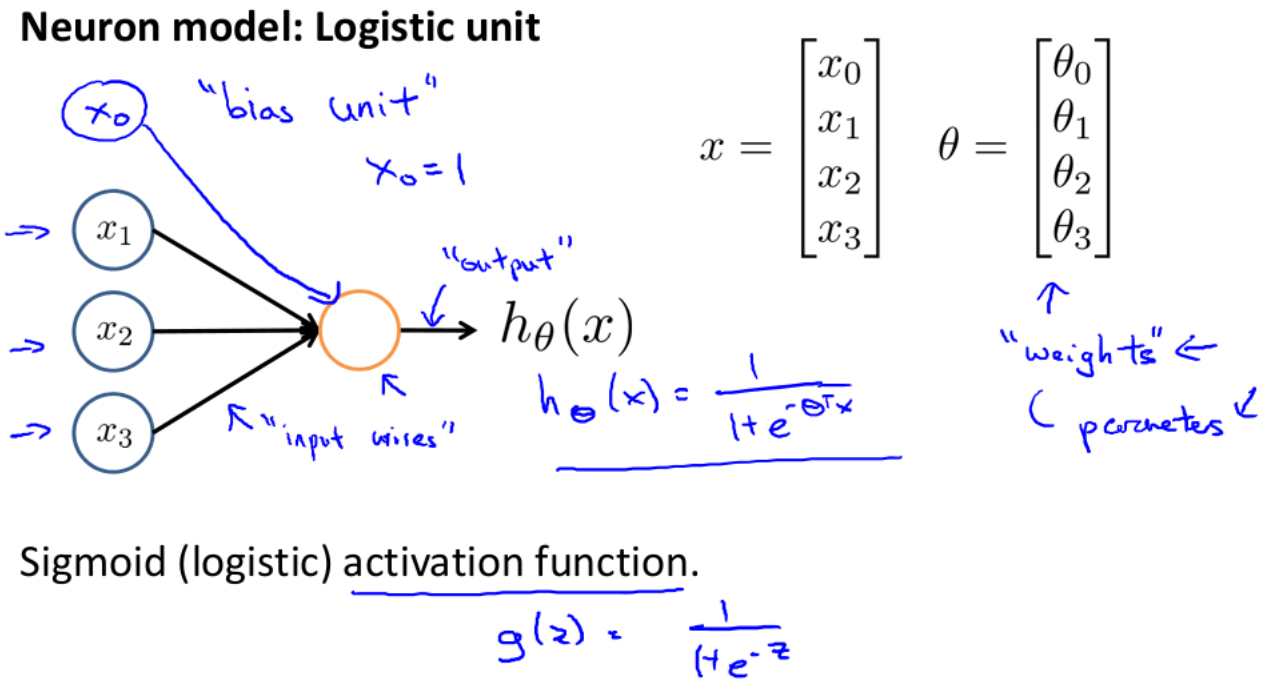


Figure 4.2: An example of a simple artificial neuron. This is basically just like logistic regression, but in slightly different notation. In fact, in Multi-layer Perceptron, the output layer is always like logistic regression. x_0 is usually 1 and there is also bias term is sometimes defined as $w_0 = -\theta$, where θ is a threshold for activation. However, with some math, we can sum over all (including bias) for activation function and threshold will become to be 0. Without bias, we would restrict possible solutions - so bias helps in learning basically.

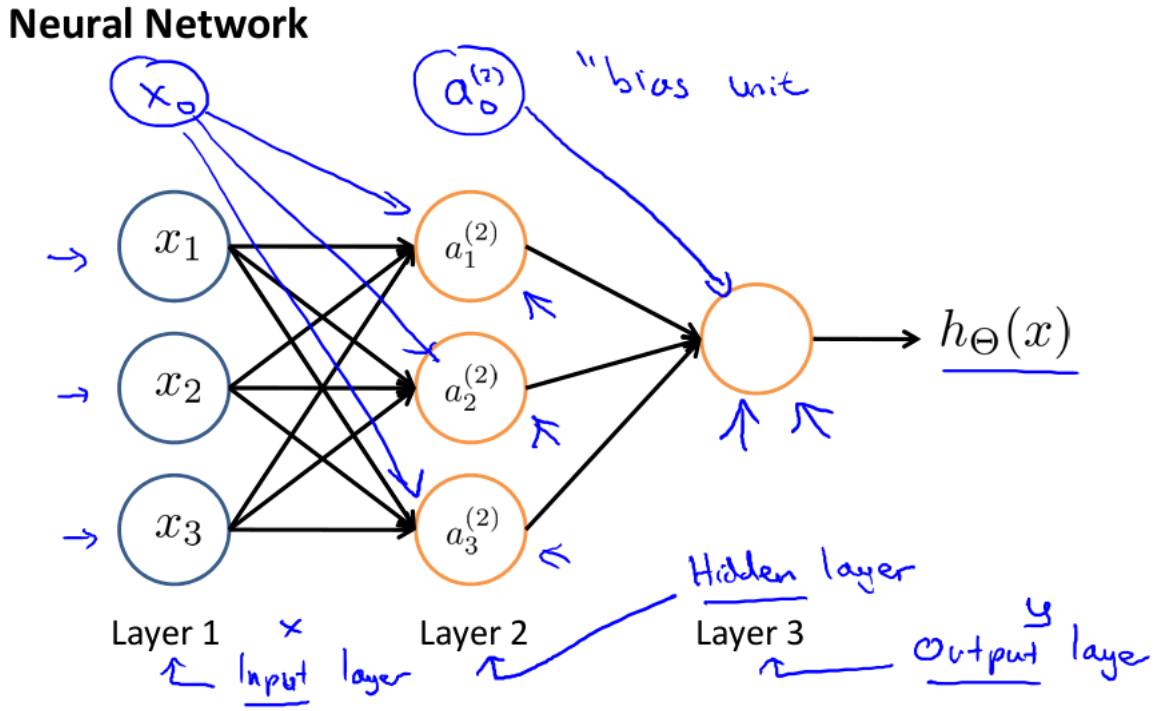


Figure 4.3: An example of artificial neuron with 1 hidden layer (also called "**shallow**"). We can see, that on each layer except the output layer, there is one unit called 'bias' (mostly value 1), which is there always implicitly. It is used for an initial shift of function and it is used as a helper constant for possible faster learning. Intermediate or "hidden" layer nodes $a_1^{(2)} \dots a_n^{(2)}$ will be always called "activation units". The way the neurons are connected, is called architecture. Each layer basically computes more complex features from a previous layer, so we can get very interesting nonlinear hypotheses. Architecture on this figure is that all input layers are connected with all neurons in hidden layer = they are densily connected. Dense layer = fully connected layer, it is the same thing.

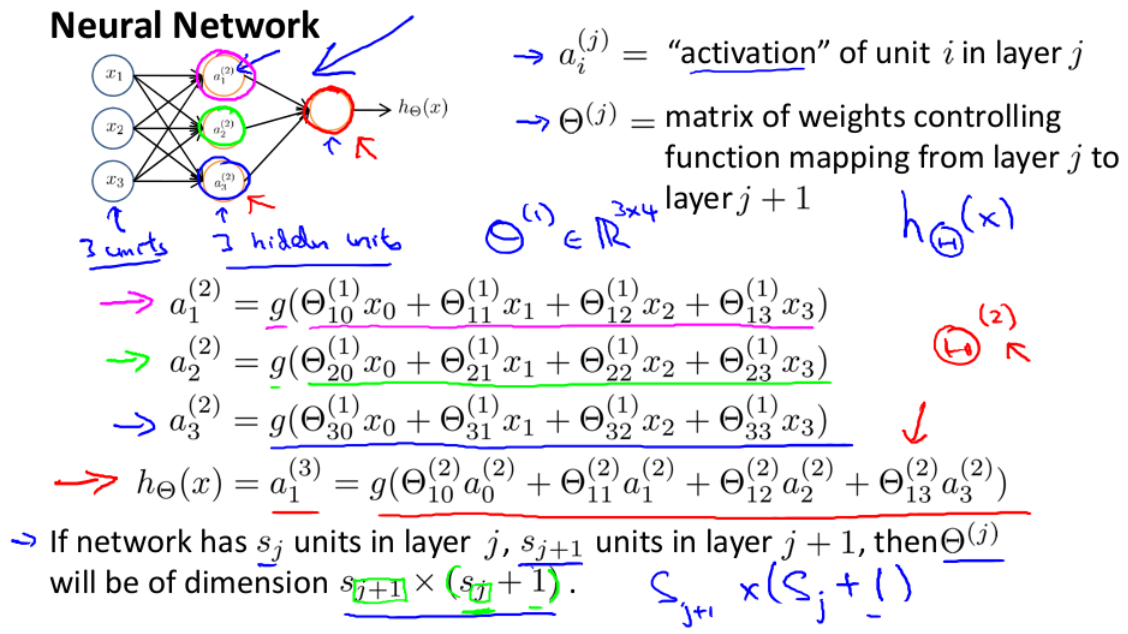


Figure 4.4: An example of artificial neuron with 1 hidden layer and the calculation of the output as well as some defined notation. If we would want to do a multiclass classification, there would be more neurons on the output layer (so more different hypotheses, each for 1 class). Computation of the hypothesis on the output layer (from left to right, so from the inputs sequentially) is called **Forward propagation** (or forward pass). There is a **basis function** g, that is used for computing a combination of all input signals into 1 value. This value is passed into an activation function (also known as **transfer function**) and the result determines the final output of a given neuron. Typical basis function is **Linear Basis Function**, or **LBF** (there exist more types, for example **Radial Basis Function**, or **RBF**).

Cost function

This is similar (although more generalized) than the one for logistic regression. **It is non-convex.** Optimization algorithms can get stuck in local minima. But in practice, this is usually not a huge problem (it usually gets a good local minimum even if is does not get the global optimum) - Andrew Ng. The following formula is **cross-entropy cost** with **L2 regularization cost** (another name for **L2 regularization** is **Euclidean norm**, or another one is **weight decay** - by the way, this can be the same for quadratic cost - in more simple notation: $C = C_0 + \frac{\lambda}{2n} \sum_w w^2$ where C_0 is the original, unregularized cost function).

Definition:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} [y_{(k)}^{(i)} log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - h_\theta(x^{(i)})_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (\theta_{j,i}^{(l)})^2 \tag{4.5}$$

- Where
  - L = total number of layers in the network.
  - s_l = number of units (not counting bias unit) in layer l.
  - K = number of output units/classes.
  - The double sum simply adds up the logistic regression costs calculated for each cell in the output layer.
  - The triple sum simply adds up the squares of all the individual thetas in the entire network.
  - Regularization parameter λ - regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. If this value is small, we prefer to minimize the original cost function, and if it is large, we prefer small weights. Regularization just do that the network prefers to learn small weights (btw, no biases; we are not regularizing biases). Btw, if cost function is unregularized, then the length of the weight vector is likely to grow, all other things being equal. Over time this can lead to the weight vector being very large indeed. This can cause the weight vector to get stuck pointing in more or less the same direction, since changes due to gradient descent only make tiny changes to the direction, when the length is long. So, unregularized cost function probably causes that it is harder for our learning algorithm to properly explore the weight space, and consequently harder to find good minima of the cost function. Unregularized runs of training will occasionally get "stuck" - they are caught in local minima of the cost function. So different runs (may) provide quite different results. By contrast, regularized runs (may) provide much more easily replicable results.

  - The i in the triple sum does not refer to training example i.
  - Cost function is always bigger than zero! All cost functions must be. All individual terms in the sums are negative, since both logarithms are of numbers in range 0 and 1, and there is a minus sign out of the front of the sums.
  - If the neuron's actual output is close to desired output for all training inputs, then cross-entropy will be close to zero.
- However, if we consider simple non-multiclass classification ($k = 1$) and discard regularization, the cost is computed with:

$$cost(t) = -y^{(t)} log(h_\theta(x^{(t)})) + (1 - y^{(t)}) log(1 - h_\theta(x^{(t)})) \tag{4.6}$$

  Derivatives of the cost function is basically δ values. For example, $\delta_2^{(1)}$ is the "error" for a_2^1 (unit 2 in layer 1).

  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$

  - just to recall, the derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are.
  - so, why derivation? Because if we want to search for a minimum of some function, gradient tells us in which direction a given function grows. So when we are going in the opposite direction, a function decreases.
- **Cross-entropy cost function** vs **quadratic cost function.** For quadratic function, learning can be slow (see derivation of sigmoid activation function using quadratic cost function - neuron saturation[4]). Neuron saturation can be solved using softmax function on output neurons, or by using cross-entropy cost function (so now we are talking about **saturation of the output neurons**, not hidden neurons - for this, we can use a proper weight initialization algorithm). Cross-entropy has a benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down - compute partial derivative of the cross-entropy with respect to the weights (and let's substitute $a = \sigma(z)$ into $C = -\frac{1}{n} \sum_x [y.ln(a) + (1 - y).ln(1 - a)]$, and apply the chain rule twice, obtaining:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x (\frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)}) \frac{\partial \sigma}{\partial w_j} = -\frac{1}{n} \sum_x (\frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)}) \sigma'(z) x_j = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1 - \sigma(z))} (\sigma(z) - y)$$

[4] If we have sigmoid activation function it is very close to zero or one, its derivative (for example, during backpropagation) is close to zero. This results that a weight in the final layer will learn slowly if the output neuron is either at low or high activation (cca 0 or 1), similar situation is for biases.

- Since $\sigma^{'}(z) = \sigma(z)(1-\sigma(z))$, equation above can be even more simplified to $\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$. This expression tells us, that the rate at which the weight learns is controlled by $\sigma(z) - y$, so by the error in the output. The larger the error, the faster the neuron will learn, what is exactly what we would expect. This avoids slow learning caused by $\sigma^{'}(z)$ term in analogous equation for the quadratic cost $C = \frac{(y-a)^2}{2}$, whose partial derivatives are $\frac{\partial C}{\partial w} = (a - y)\sigma^{'}(z)x = a\sigma^{'}(z)$ and the same for bias, $\frac{\partial C}{\partial b} = (a-y)\sigma^{'}(z) = a\sigma^{'}(z)$ - there, when a neuron's output is close to 1 (or zero), the curve of sigmoid gets very flat, and so $\sigma^{'}(z)$ gets very small = learning is very slow. When using cross-entropy, $\sigma^{'}(z)$ term is canceled out. and so we no longer need to worry about being it very small. And cross-entropy was specially chosen to have this property.
- By the way, all the previous can be also calculated for the bias term:

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x(\sigma(z) - y)$$

- Cross-entropy is almost always a better choice than quadratic cost, provided the output neurons are sigmoid neurons. If there are linear activation functions only, quadratic cost function is appropriate and will not cause the learning slowdown. Why? Because $\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n}\sum_x(a^{L-1}_k(a^L_j - y_j))$ and $\frac{\partial C}{\partial b^L_j} = \frac{1}{n}\sum_x(a^L_j - y_j)$ - it is because of derivatives of linear function. So, choosing cost function depends on activation functions that are used, but mostly cross-entropy is a good choice.
- Learning rate - for both functions, it is not possible to say precisely what it means to use the same learning rate when the cost function is changed. It must be chosen with experimenting to find near-optimal performance given also other hyper-parameter choices (very rough general heuristic for relating the learning rate for cross-entropy and quadratic cost, quadratic cost learns an average of 6x slower and this is the same for learning rate; but don't take too seriously, it can be just as a starting point).
- How was cross-entropy created? As noted earlier, if we have sigmoid activation function and we would derive quadratic cost function: $\frac{\partial C}{\partial w} = (a - y)\sigma^{'}(z)x = a\sigma^{'}(z)$, and $\frac{\partial C}{\partial b} = (a - y)\sigma^{'}(z) = a\sigma^{'}(z)$ (substitution of $x = 1$ and $y = 0$) there is a learning slowdown problem - because of neuron saturation. Now we can create a new cost function in which term $\sigma^{'}(z)$ would disappear. The easiest example would be $\frac{\partial C}{\partial w_j} = x_j(a-y)$ and $\frac{\partial C}{\partial b} = a - y$. This would mean, the greater the initial error, the faster the neuron learns, and it would eliminate the problem of learning slowdown. In fact, from these equations we can derive cross-entropy. From chain

rule, we have $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma^{'}(z)$ and using $\sigma^{'}(z) = \sigma(z)(1-\sigma(z)) = a(1-a)$,it becomes $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}a(1-a)$. Then, comparing to $\frac{\partial C}{\partial b} = a - y$, we obtain $\frac{\partial C}{\partial a} = \frac{a-y}{a(1-a)}$ and integrating this expression with respect to a gives us: $C = -[y\,ln(a) + (1-y)ln(1-a)] + constant$ for some constant of integration. This is the contribution to the cost from a single training example, x. To get full cost function, we must average over training examples: $C = -\frac{1}{n}\sum_x [y\,ln(a) + (1-y)ln(1-a)] + constant$, where the constant here is the average of the individual constants for each training example.

  - Roughly speaking, cross-entropy is a measure of surprise. In particular, a neuron is trying to compute the function $x \rightarrow y = y(x)$. But instead it computes the function $x \rightarrow a = a(x)$. Suppose we think of a as out neuron's estimated probability that $y = 1$ and $1 - a$ is estimated probability that the right value for y is 0. Then, cross-entropy measures how "surprised" we are, on average, when we learn the true value for y. We get low surprise if the output is what we expect, and high surprise if the output is unexpected.

- A few nested summations have been added to account for multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.
- In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.
- Regularization can set a weights of a neuron so close to zero, that it is basically zeroing out an impact of a given neuron. So if the regularization is high, a lot of hidden units will have smaller impact and therefore the model itself is basically smaller.
- **Dropout regularization technique**: go through each neuron in each hidden layer, and set a probability of eliminating a node in ANN pseudo-randomly to some value. Dropout has similar effect to L2 regularization, but L2 can be more adaptive to the scale of different inputs. With dropout, cost function J is not well defined and it is not guaranteed that it is going downhill on every iteration. Cost function J is harder to calculate. Gradient checking technique does not work with dropout, because with every iteration, dropout is randomly eliminating different subsets of the hidden units. There isn't an easy way to compute cost function that dropout is doing gradient descent on. Dropout is used during training not for testing - you don't want to flip coins to decide which hidden units to eliminate. And that's because when you are

making predictions at the test time, you don't really want your output to be random. And why dropout works? Because we will not rely on any 1 feature, but weights are spread out. There is a parameter $keep\,prop$ and can be set to each layer to be different (for example bigger hidden layers have relatively low dropout, for example 0.5 with 7 hidden units, and the last layers which are very small, like 1-2 neurons, $keep\,prop = 1.0$ and so no dropout is done on these last layers; btw, you can also use dropout for the input layer but it is not being done so often - and if yes, then with very small $keep\,prop$). Dropped neurons don't contribute to the training in both the forward and backward propagations of the iteration. When you shut some neurons down, you actually modify your model. The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time. Steps:

1. Create a matrix $D^{[l]}$ which has the same shape as $A^{[l]}$. Set each entry in $D^{[l]}$to be 0 with probability $1-keep\,prop$, or 1 with probability $keep\,prop$ by thresholding randomly generated values in $D^{[l]}$.
2. Set $A^{[l]} = A^{[l]} * D^{[l]}$ (you are shutting down some neurons).
3. Divide $A^{[l]}$ by $keep\,prop$ - this technique is called inverted dropout. By doing this you are assuring that the result of the cost will still have the same expected value as without dropout. For example, if $keep\,prop$ is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value.

And then backpropagation:

1. You had previously shut down some neurons during forward propagation, by applying a mask $D^{[1]}$ to $A^{[1]}$. In backpropagation, you will have to shut down the same neurons, by reapplying the same mask $D^{[1]}$to $dA^{[1]}$.
2. During forward propagation, you had divided $A^{[1]}$ by $keep\,prob$. In backpropagation, you'll therefore have to divide $dA^{[1]}$ by $keep\,prob$ again (the calculus interpretation is that if $A^{[1]}$ is scaled by$keep\,prob$, then its derivative $dA^{[1]}$ is also scaled by the same $keep\,prob$).

- Dropout is basically a form of model averaging - extreme bagging. It is an efficient way to average a large number of neural nets. The training sets are very different for the different models, but they're also very small. The sharing of the weights between all the models means that each model is very strongly regularized by the others. And this is a much better regularizer than things like L2 or L1 penalties. Those penalties pull the weights toward zero. By sharing weights with other models, a models gets regularized by

something that's going to tend to pull the weight towards the correct value. At test time - we use all of the hidden units, but halve their outgoing weights. This exactly computes geometric mean of the predictions of all 2^H models (if we are sampling from 2^H different architectures where H is a number of hidden units, and we are randomly omitting each hidden unit with probability of 0.5). **Input layer** - we can use dropout here too, but with a higher probability of keeping an input unit. This trick is already used in "**denoising autoencoders**".

- It is also possible to use **Early stopping**, but this has some downside. Usually, you want orthogonalization - you want to be able to think about 1 task at a time, not multiple at the same time. For example, Andres NG never uses Early stopping (but Geoffrey Hinton does). Usually we want to separately optimize cost function J (gradient descent, momentum, RMS prob, etc) and then solve overfitting (regularization etc), so to work on all task independently - this keeps the exploration of hyperparameters well behaved - don't optimize cost function and regularize at the same time. He is instead uses just L2 regularization, but it is needed to try different values of lambda which is more computationally expensive.
  - Early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate. This makes setting the number of epochs very simple. In particular, it means that we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters. Instead, that's taken care of automatically. Furthermore, early stopping also automatically prevents us from overfitting. This is, of course, a good thing, although in the early stages of experimentation it can be helpful to turn off early stopping, so you can see any signs of overfitting, and use it to inform your approach to regularization.
  - To implement this, we need to say what it means that the classification accuracy has stopped improving. We can watch the last few (for example 10) epochs, and decide to terminate if the performance was not improved. But there can be different strategies. This would ensure, that we won't stop too soon, in response to bad luck in training, but also that we are not waiting around forever for an improvement that never comes.
  - However, sometimes networks can plateau near a particular classification accuracy for quite some time, only then they will begin to improve again. That may be too aggressive. Maybe you can try multiple number of last epochs for deciding whether the accuracy was or was not improved - so this number (number of epochs to monitor for early stopping) would be another hyperparameter.

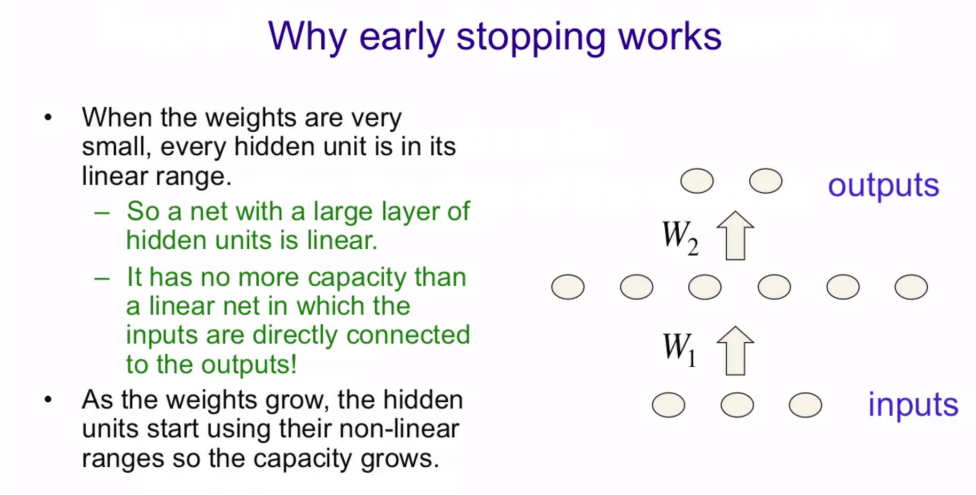


Figure 4.5: Early stopping is a way of preventing overfitting. It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse (it can be difficult to estimate when the performance is getting worse). The capacity of the model will be limited, because the weights have not had time to grow big. When the weight's very small, if the hidden unit's a logistic units, their total inputs will be close to zero, and they'll be in the middle of their linear range. That is, they'll behave very like linear units. What that means is, when the weights are small, the whole network is the same as a linear network that maps the inputs straight to the outputs.

- Another ways of doing regularization of neural networks is **adding noise to the input data**, or to the **network weights**, or to the **hidden activations**. Actually, adding Gaussian noise (with the mean equals to zero) to the weights of a multi-layer non-linear net is not exactly equivalent to using L2 weight penalty, but it seems that it actually may work even better, especially in RNNs.

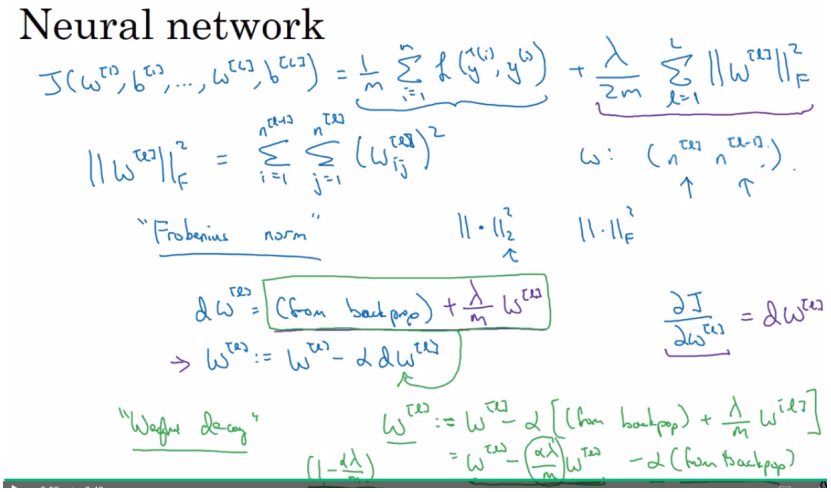


Figure 4.6: ANN with regularization using "Frobenius norm" (but it is similar to L2 norm).

Forward Propagation and Backpropagation

"Backpropagation" is neural-network terminology for minimizing the cost function, just like what we were doing with gradient descent in logistic and linear regression (actually backpropagation is gradient descending process). That is, we want to minimize our cost function J using an optimal set of parameters in theta.

We perform forward and backward pass sample by sample.

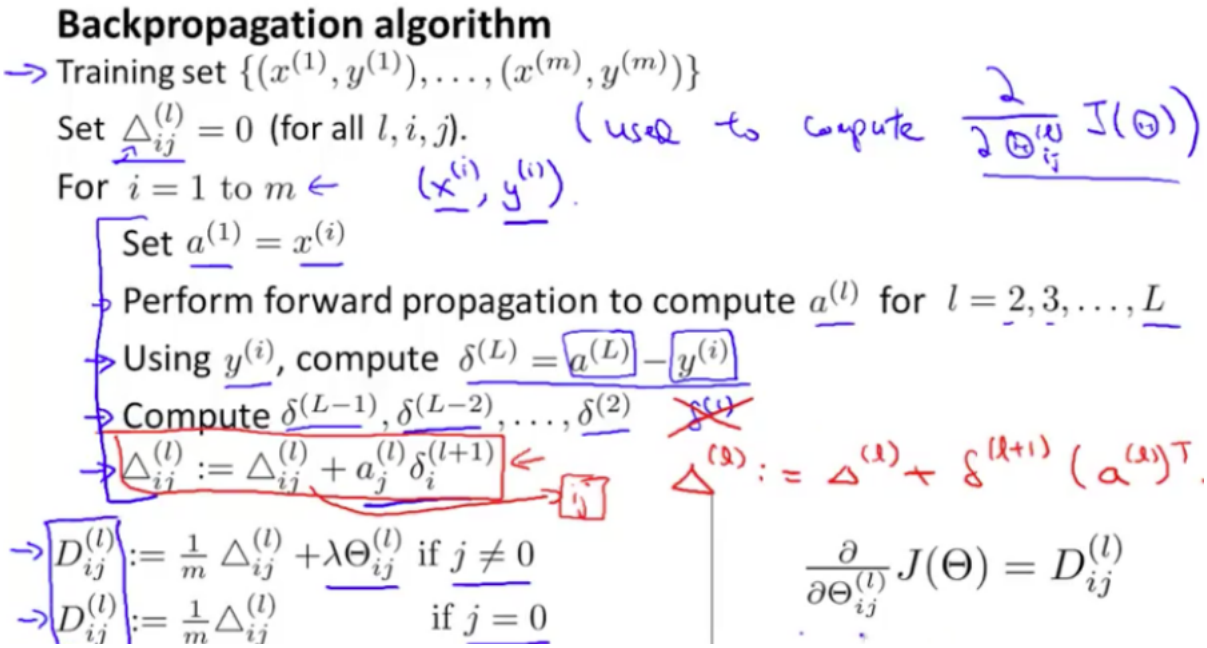


Figure 4.7: A simplified version of Backpropagation algorithm.

Back propagation Algorithm

Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$

- Set $\Delta_{i,j}^{(l)}$:= 0 for all (l,i,j), (hence you end up having a matrix full of zeros)

For training example t =1 to m:

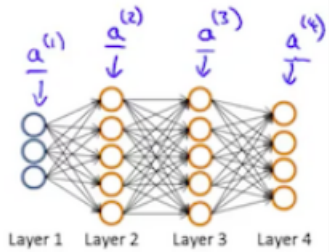
1. Set $a^{(1)} := x^{(t)}$
2. Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L

Gradient computation

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \, .* \, a^{(l)} \, .* \, (1 - a^{(l)})$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$.

Figure 4.8: Backpropagation algorithm (I).

The g-prime derivative terms can also be written out as:

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

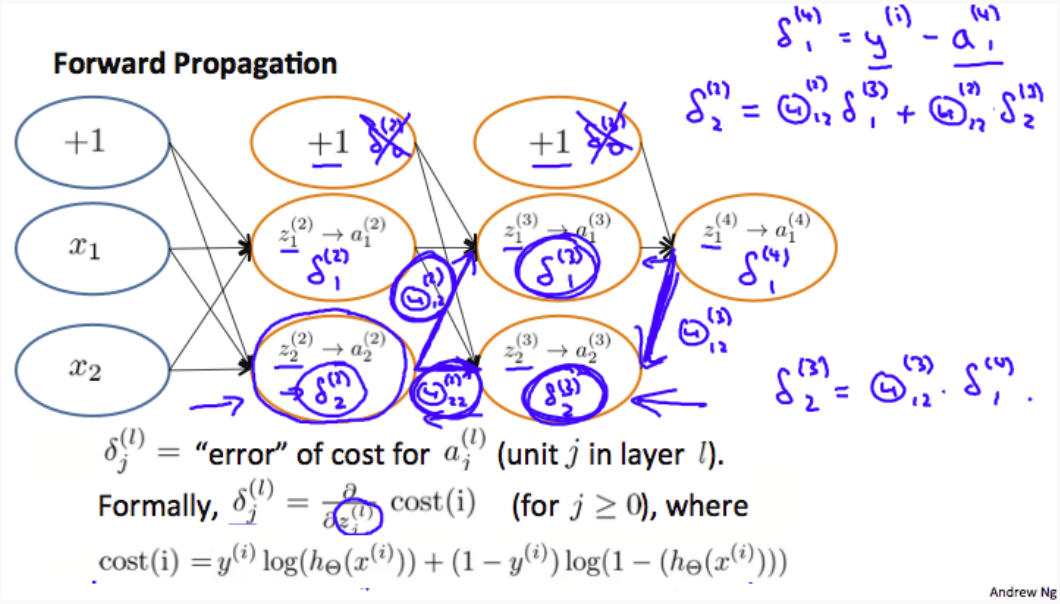
Hence we update our new Δ matrix.

- $D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda\Theta_{i,j}^{(l)}\right)$, if j≠0.
- $D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)}$ If j=0

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)}$

Figure 4.9: Backpropagation algorithm (II).

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some $\delta_j^{(l)}$:



In the image above, to calculate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ values found to the right of each edge. So we get $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$. To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$.

Figure 4.10: An example of calculating delta values in Backward pass from a values of Forward Propagation for a simple ANN. Delta represents an error of cost function.

Gradient Checking

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to** Θ_j as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

A small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

Once you have verified **once** that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

Figure 4.11: Gradient Checking algorithm for making sure that our implementation of Backpropagation is correct. More information is described in Subsection 1.9.

Another view on Backpropagation

Backpropagation is based around 4 fundamental equations (for any activation function). Together, they give us a way of computing both error σ^{layer} and the gradient of the cost function. They can be proven by applying chain rule.

- **An equation for the error in the output layer**, σ^L. The first term on the right, $\frac{\partial C}{\partial a_j^L}$, just measures how fast the cost is changing as s function of the j-th output activation. The second term on right, $\sigma^{'}(z_j^L)$, measures how fast the activation function σ is changing at z_j^L. Everything is easily computed - z_j^L is calculated anyway (forward pass), and $\sigma^{'}(z_j^L)$ is just a small overhead. The exact form of $\frac{\partial C}{\partial a_j^L}$ will of course, depend on the form of cost function. In case of quadratic cost, which is $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$, $\frac{\partial C}{\partial a_j^L}$ equals to $a_j^L - y_j$.

$$\sigma_j^L = \frac{\partial C}{\partial a_j^L} . * \sigma^{'}(z_j^L) \tag{4.7}$$

- **An equation for the error σ^{layer} in terms of the error in the next layer, $\sigma^{layer+1}$**. We can think of this intuitively as moving the error backward through the network, with Hadamard product. Combining 4.7 and 4.8 equations, we can compute the error σ^l for any layer in the network. We start by using 4.7 to compute σ^L, then apply Equation 4.8 to compute σ^{L-1}, then apply Equation 4.8 again to compute σ^{L-2},and so on, all the way back through the network.

$$\sigma^{layer} = ((w^{layer+1})^T \sigma^{layer+1}) . * \sigma^{'}(z^L) \tag{4.8}$$

- **An equation for the rate of change of the cost with respect to any bias in the network**. So the error σ_j^{layer} is exactly equal to the rate of change $\partial C \,/\, \partial b_j^{layer}$.

$$\frac{\partial C}{\partial b_j^{layer}} = \sigma_j^{layer} \tag{4.9}$$

- **An equation for the rate of change of the cost with respect to any weight in the network.** Nice consequence of this equation is that when the activation $a_k^{layer-1}$ is close to 0, the gradient term $\partial C \,/\, \partial w$ will also tend to be small. In this case, we will say that the weight learns slowly - it is not changing much during gradient descent. In other words, weights output from low-activation neurons learn slowly.

$$\frac{\partial C}{\partial w_{jk}^{layer}} = a_k^{layer-1} \sigma_j^{layer} \tag{4.10}$$

Putting it together

- Pick some ANN architecture (connectivity pattern between neurons):
  - Number of input (dimension of features) and output (number of classes) layers is known.
  - Number of hidden layers - reasonable default is to have just 1 hidden layer
  - Number of hidden units in a hidden layer - same # of units in all layers (recommended, by Andrew Ng), usually the more the better (but more computationally expensive). For example (Andrew Ng), 3 or 4 times more than # of features (at least a bit more hidden units than # of features is a useful thing - Andrew Ng).
- Training an ANN (forward and then backward pass in a for loop over the training examples is the simplest way):
  - Randomly **initialize weights**. Small values near to zero (but not zeroes).
  - Implement **forward propagation** to get value of hypothesis for any training sample.
  - Implement **cost function** $J(\theta)$.
  - Implement **backpropagation** to compute partial derivatives terms $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$ (partial derivatives of $J(\theta)$ with respect of the parameters) = get activations and delta terms for all the layers.
  - Implement **numerical gradient check** to compare $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$ computed using backpropagation and numerical estimate of gradient of $J(\theta)$ give similar values (numerical estimates and the vector of deltas must give similar values). Otherwise there is a bug in the implementation. After we can see that backprop is implemented correctly, turn off gradient checking.
  - Use gradient descent or some advanced optimization method (BFGS, L-BFGS, or Conjugate Gradient) with backprop for minimizing $J(\theta)$ as a function of parameters θ (learning).

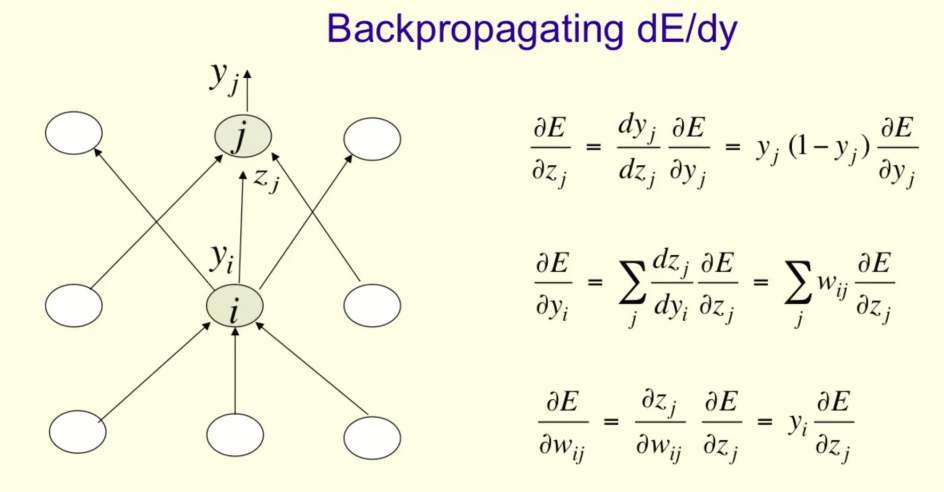


Figure 4.12: Backpropagation explanation through derivatives using chain rule.

- Applying gradient descent learning algorithm in a regularized neural network using cost function C_0 and L2 regularization can be following: we need to compute partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for all the weights and biases in the network: $\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w$ and $\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$ - so we can see that biases are not affected by regularization. It is possible to modify this and regularize also biases, but its basically a convention to regularize just weights (also, having a large bias doesn't make a neuron sensitive to its inputs in the same way as having large weights; also, allowing large biases gives our networks more flexibility in behavior - large biases make it easier for neurons to saturate, which is sometimes desirable). So we just use backpropagation, as usual, and then we will add $\frac{\lambda}{n}w$ to the partial derivative of all weight terms. So, learning for weights becomes: $w \to w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n}w = (1 - \frac{\eta\lambda}{n})w - \eta\frac{\partial C_0}{\partial w}$, and for biases it remains the same: $b \to b - \eta\frac{\partial C_0}{\partial b}$, where, of course, η is the learning rate. So, from weight update formula, we can see, that it is the same as without any regularization, we just rescale weight w by a factor $1 - \eta\frac{\lambda}{n}$. This rescaling is sometimes referred to as weight decay, since it makes the weights smaller. For stochastic gradient descent with mini-batch of size m, it is almost the same: $w \to (1 - \frac{\eta\lambda}{n})w - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial w}$ - so we are averaging over a mini-batch of m training examples, the sum is over training example x in the mini-batch, and $C_x$is unregularized cost for each training example. Regarding biases, it follows the same principle: $b \to b - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial b}$.
- Tips:
  - small ANN (# of hidden units) is more prone to underfitting, model is too simple. Benefit - computationally cheaper.
  - larger ANN (# of hidden units) is more prone to overfitting, model is too complex. Disadvantage - computationally more expensive.

- step by step increasing # of hidden layers. Regularization addresses overfitting. We can plot a learning curve to find out cross validation error with increasing # of hidden layers. Final candidate should be one with the lowest cross validation error.
- An example, CV error is much larger than training error. Increasing the number of hidden units is not likely to help, because it suffers from high variance.

Finite Difference Approximation

- An alternative to backpropagation. This procedure approximates the gradient of the error with the respect to the weights (think of the definition of a derivative). In effect, the procedure states that we approximate the gradient and then take a step of the steepest descent method.
- Although this method works, the **backpropagation algorithm finds the exact gradient much more efficiently**.
- Process:
  1. For each weight parameter w_i , perturb w_i by adding a small (say, 10^{-5}) constant *epsilon* and evaluate the error (call this E_i^+)
  2. Now reset w_i back to the original parameter and perturb it again by subtracting the same small constant ϵ and evaluate the error again (call this E_i^-).
  3. Repeat this for each weight index i.
  4. Upon completing this, we update the weights vector by: $w_i \leftarrow w_i - \eta \frac{(E_i^+ - E_i^-)}{2\epsilon}$ for some learning rate η.

Activation functions

(from here[5])

[5] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

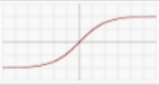
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figure 4.13: Various activation functions cheatsheet.

- Except to **regression problem**, where the **output layer should use a linear activation function**, so that the predicted output can go from 0 to ∞, we never use linear activation function. We would compute a linear function of the inputs (no matter how many hidden layers and hidden neurons we use). Anyway, hidden layers in regression problem still not use a linear activation function.

- **Sigmoid or Logistic activation function**
  - The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.
  - Range: $(0, 1)$.
  - Derivative of sigmoid activation function is $g(z)' = \frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}(1 - \frac{1}{1+e^{-z}}) = g(z)(1 - g(z)) = a(1 - a)$
  - They can compute any function.
  - The logistic sigmoid function can cause a neural network to get stuck at the training time. One of the problems of using sigmoid functions is that in regions (on diagram, where z is bigger than 5, or lower than -5), the slope of the function causes to be gradient nearly 0, and so learning becomes really slow because when you implement gradient descent and gradient is zero the parameters just change very slowly. There is no such problem in ReLU (which is very popular and used now) where the gradient is equal to 1 for the positive input - and the whole gradient descent is working much faster.
  - **The softmax function** (aka softmax regression) is a more generalized logistic activation function which is used for multi-class classification (if number of classes is 2, then this can be proven):

$$\sigma(z) = \frac{e^z}{\sum_{k=1}^{K} e^z} \tag{4.11}$$

    * For 1 example in dataset: K is a number of classes, z (input) is K-dimensional vector of arbitrary real values, and output is also K-dimensional vector of real values where each entry is in range $(0, 1)$, and all entries add up to 1.
    * **Example**: we may use softmax layer $z^{[l]}$ that is doing this calculation. For instance, $z^{[l]}$ will have on the input a 4x1 dimensional vector of values 5, 2, -1, 3. Then, we got: $\begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$ and $sum\,of\,all\,elements = 176.3$. Then, $a = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$ and this is cca (rounding errors) summed to a probability 1.0.
    * The name "soft max" is an opposite to "hard max" function, whose output would be like this: $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$.

* This is a way of forcing the outputs of a neural network to sum to 1 (all neurons in the output layer), so they can represent a probability distribution across discrete mutually exclusive alternatives.
* Softmax can be used with log-likelihood cost function, that is $C = -ln(a_j^L)$.

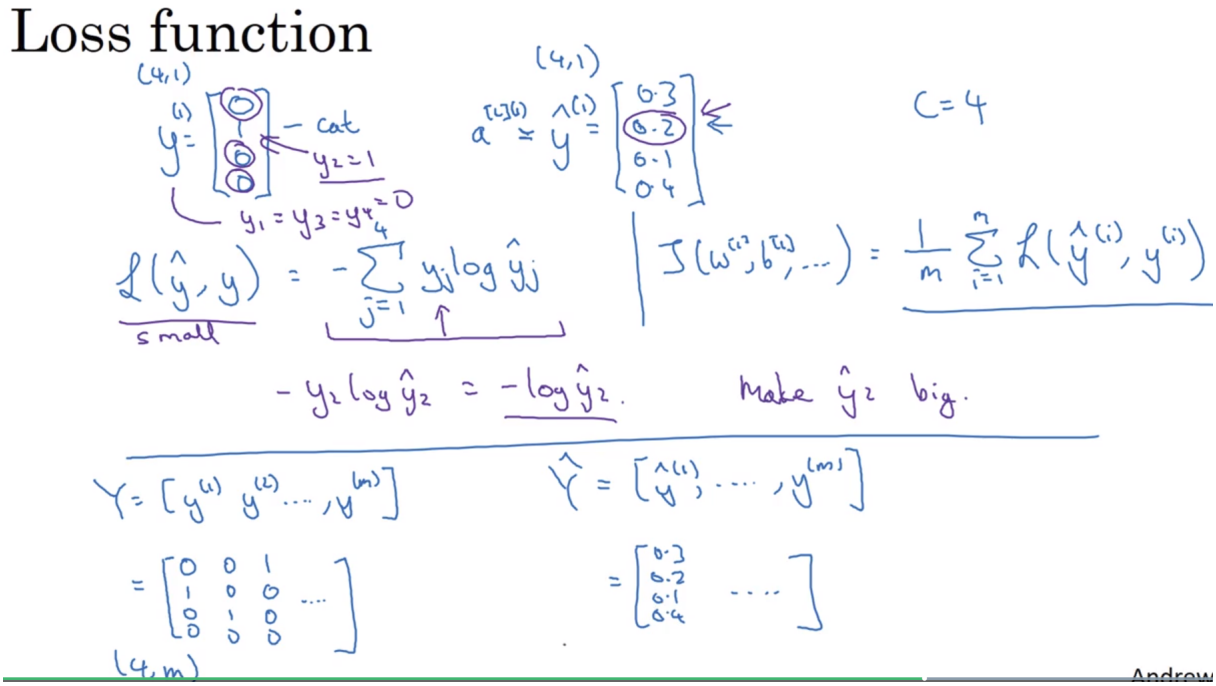


Figure 4.14: Loss function computation for softmax, it is still a scalar value (not a vector).

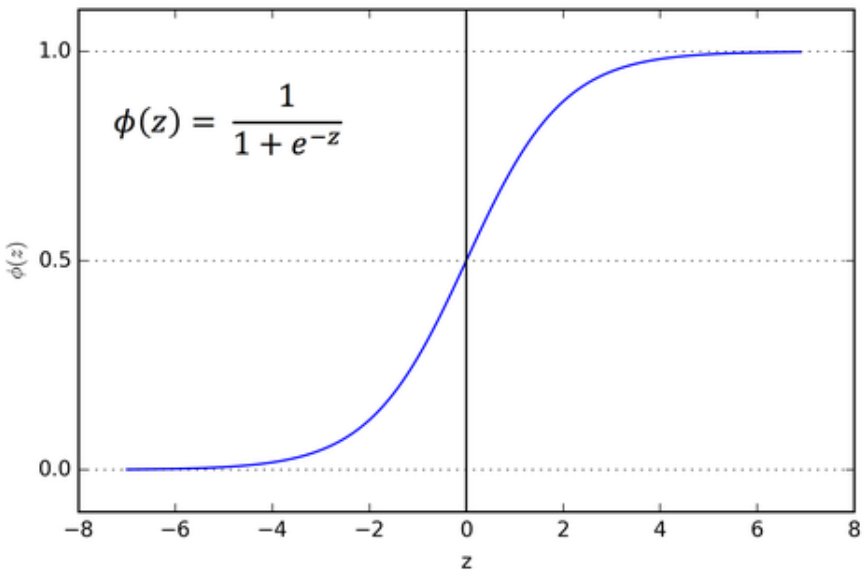


Figure 4.15: Sigmoid activation function.

- **Tanh or hyperbolic tangent activation function**
  - Range $(-1, 1)$. It is basically shifted and rescaled version of sigmoid. It is almost always preferred to use rather than sigmoid, e**xcept the last, output** layer where we want output to be between 0 and 1 - in that case it is better to use sigmoid function for binary classification problem. It works better than sigmoid for neurons in hidden layers, because the mean of its output is closer to zero, and so it centers the data better for the next layer. This makes the learning process for the next layer easier.
  - This activation function tanh is also sigmoidal (s-shaped).
  - The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
  - They can compute any function.
  - It is basically calculated as $f(x) = \frac{sinh\, x}{cosh\, x}$.
  - Hyperbolic tangent function is closely related to sigmoid neuron:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{4.12}$$

    and with a little algebra, it can be easily verified that $\sigma(z) = \frac{1+tanh(z/2)}{2}$, that is, tanh is just a rescaled version of sigmoid function (with the same shape). So the output from tanh ranges from -1 to 1, not from 0 to 1. And this means that in a network based on tanh neurons, you may need to normalize your outputs (and maybe even inputs, depending on the application) a little differently than in sigmoid networks.
  - There are some empirical evidence that tanh sometimes performs better than sigmoid. Suppose we are using sigmoid neurons, so all activations in our network are positive. Let's consider the weights $w_{jk}^{layer+1}$ input to the j-the neuron in the $layer + 1$ layer. The rules for backpropagation tell us that the associated gradient will be $a_k^{layer}\delta_j^{layer+1}$. Because the activations are positive, the sign of this gradient will be the same as the sign of $\delta_j^{layer+1}$. This means, that if $\delta_j^{layer+1}$ is positive, then all weights $w_{jk}^{layer+1}$will decrease during gradient descent, while if $\delta_j^{layer+1}$ is negative, then the weights will increase. In other words, all weights to the same neuron must either increase together, or decrease together. That is a problem, since some of the weights may need to increase while others need to decrease. That can only happen if some of the input activations have different signs. That suggests replacing the sigmoid by an activation function, such as tanh, which allows both positive and negative activations. Since tanh is symmetric about zero, $tanh(-z) = -tanh(z)$, the activations in hidden layers would be equally balanced between positive and negative. That would help ensure that there is no systematic

bias for the weight updates to be one way or the other. This argument is just a suggestion, heuristic, not a proof that tanh neurons outperform sigmoid neurons. For many tasks, tanh is found empirically to provide only a small or no improvement in performance over sigmoid neurons.

- However, when derivative of this activation function is close to 0, then the learning process is very slow. ReLU to the rescue!
- Derivative is $g(z)' = \frac{d}{dz}g(z) = 1 - (tanh(z))^2 = 1 - g(z)^2 = 1 - a^2$

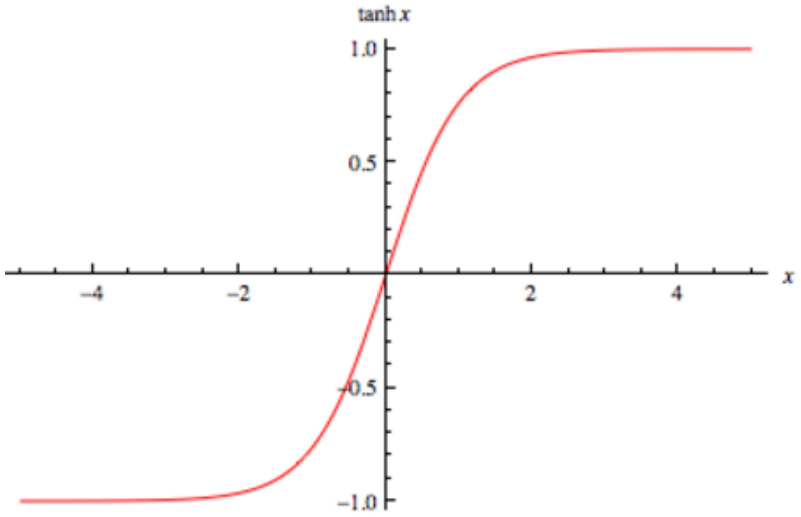


Figure 4.16: Hyperbolic tangent activation function (Tanh).

- **ReLU (Rectified Linear Unit) activation function**
  - The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.
  - Range: $[0, \infty)$, defined as

$$relu(z) = max(0, z) \tag{4.13}$$

  - **ReLU has one disadvantage**, all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately, which in turns affects the resulting graph by not mapping the negative values appropriately. When $z = 0$, ReLU is not differentiable, but in practice we will almost never encounter the precise zero.
  - They can compute any function.

  - Derivative is $g(z)' = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \\ undefined & z = 0 \end{cases}$
  - We don't precisely know when and why ReLU should be preferable. However, tanh and sigmoid neurons both can saturate. By contrast, increasing the weighted input to a ReLU will never cause it to saturate, so there is no corresponding learning slowdown. On the other hand, then the weighted input to ReLU is negative, the gradient vanishes, and so neuron stops learning entirely.

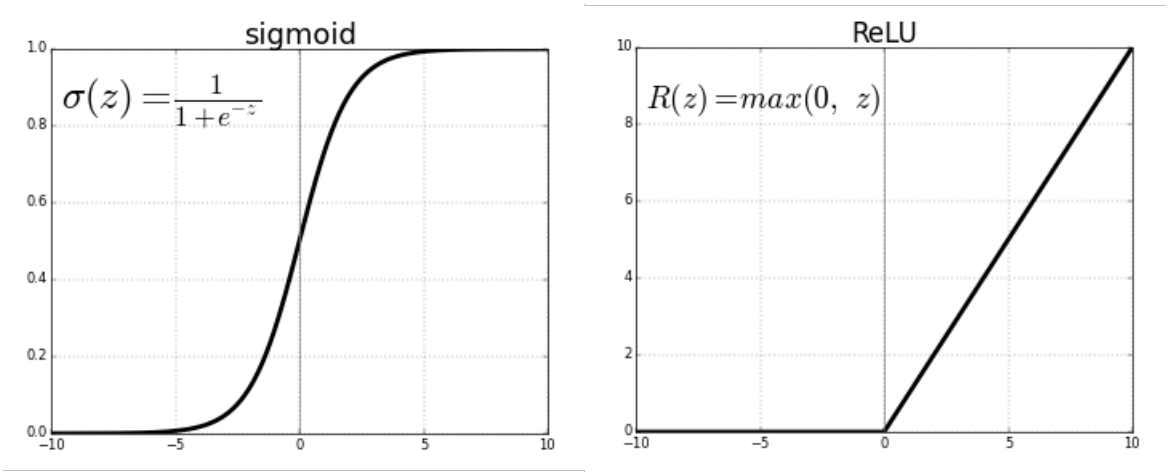


Figure 4.17: Sigmoid (on the left) vs ReLU (on the right) activation function.

- **Leaky ReLU**
  - It is an attempt to solve the dying ReLU problem.
  - The leak helps to increase the range of the ReLU function. Usually, $a = 0.01$ or so. When $a \neq 0.01$ then it is called Randomized ReLU.
  - So it can be defined as

$$R(z) = max(a * z, z) \tag{4.14}$$

  - Derivative is $g(z)' = \begin{cases} 1 & z > 0 \\ a & z < 0 \\ can\ be\ set\ to\ 1 & z = 0 \end{cases}$

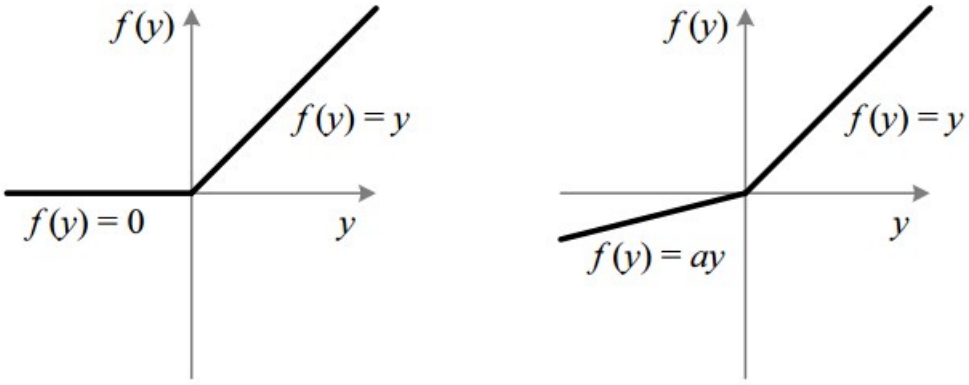


Figure 4.18: ReLU (on the left), vs Leaky ReLU (on the right) activation functions.

Bayesian approach

- The main idea behind this is that instead of looking for the most likely setting of the parameters of the model, we should consider all possible settings of the parameters and try to figure out for each of those possible settings, how probable it is, given the data we observed. This is extremely computationally expensive. After we've computed the posterior distribution across all possible settings of the parameters, we can then make predictions by letting each different setting of the parameters make its own prediction. And then, averaging all those predictions together, weighting by their posterior probability. This is also very computationally intensive. We can use complicated models even when we don't have much data.
- We're now used to the idea of overfitting, When you fit a complicated model to a small amount of data. But that's basically just a result of not bothering to get the full posterior distribution over the parameters. So, when you don't have much data, you should use a simple model. This is true, IF you assume that fitting a model means finding the single best setting of the parameters. If you find the full posterior distribution, that gets rid of overfitting. If there's very little data, the full posterior distribution will typically give you very vague predictions, because many different settings of the parameters that make very different predictions will have significant posterior probability. As you get more data, the posterior probability will get more and more focused on a few settings of the parameters, and the posterior predictions will get much sharper.
- So, in other words, in full Bayesian learning, we don't try and find a single best setting of the parameters. Instead, we try and find the full posterior distribution over all possible settings.
- The Bayesian framework assumes that we always have a prior distribution for everything. That is, for any event that you might care to mention, I must have some prior probability that such event might happen. The problem might be very vague.
- **Explanation of using Bayesian approach to fitting models using a simple coin-tossing example is below.**
  - Our data gives us likelihood term. We combine it with our prior and then we get a posterior. Given enough data, even if your prior is wrong, you'll end up with the right hypothesis. But that may take an awful lot of data if you thought that things were very unlikely under your prior.
  - Suppose you don't know anything about coins except that they can be tossed and when you toss a coin you get either a head or a tail. And we're also going to assume you know that each time you do that it's an independent decision.
  - So our model of a coin is going to have one parameter P. This parameter P determines the probability that the coin will produce a head. What happens

now if we see 100 tosses and there are 53 heads. What is a good value for P? The answer is also called maximum likelihood - pick the value of P that makes the observations the most probable.

- Let's compute that (it is a probability of a particular sequence containing 43 heads and 47 tails) by deriving $P(D) = p^{53}(1-p)^{47}$:

  $\frac{dP(D)}{dp} = 53p^{52}(1-p)^{47} - 47p^{53}(1-p)^{46} = (\frac{53}{p} - \frac{47}{1-p})[p^{53}(1-p)^{47}]$
- If we ask, how does the probability of observing that data depend on p, we can differentiate with respect to p, and we get the expression shown above. If we then set that derivative to zero, we discover, that the probability of the data is maximized by setting P to be 0.53. So this is maximum likelihood. You can calculate it manually:
  1. $(\frac{53}{p} - \frac{47}{1-p})[p^{53}(1-p)^{47}] = 0$.
  2. If this has to be 0, then either left or right part must be 0. Let's try left part.
  3. $\frac{53}{p} - \frac{47}{1-p} = 0$
  4. $\frac{53(1-p)-47p}{p(1-p)} = 0$
  5. $\frac{53-53p-47p}{p-p^2} = 0$
  6. $\frac{53-100p}{p-p^2} = 0$
  7. This is 0 if nominator is 0.
  8. $53 - 100p = 0$
  9. $\frac{53}{100} = p = 0.53$
- There are some problems with picking the parameters that are most likely to generate the data. What if we only tossed the coin once and we got 1 head? It does not really makes sense to say that $p = 1$ for the next attempt (trial).
- It is even reasonable to give a single answer? If we don't have enough data, we are unsure about p.
- **What we really do is to refuse to give a single answer. Instead, we are computing probability distributions across all possible answers.**
- We will start with a prior distribution over p. In this case, let's use uniform distribution - any bias is equally likely.
- **Now multiply the prior probability of each parameter value by the probability of observed "head" given that value.** So, for example, if we take the value of $P = 1$ which says that coins come down heads every time then the probability of observing a head would be 1. There would be no alternative. And if we take the value of $P = 0$, the probability of observing a head would be 0. And if we take it to 0.5, the probability of observing head is 0.5. Now we got unnormalized (area under such curve does not sum to 1,

but in probability distribution, the probabilities of all alternative events must sum to 1) posterior probability.

  - The last thing is scaling, so that we got obtained area under curve summed to 1. This is called **posterior distribution**.
  - We can now do this for "tail" and then we can also do it for another 98 times. After 53 heads and 47 tails, we get a very sensible posterior distribution that has its peak at 0.53, assuming a uniform prior. Area under the curve sums to 1 and it have its mean equals to 0.53. This is basically Gaussian distribution.

- Approximating full Bayesian learning in a neural net - this is possible if NN has only a few parameters. We will put a grid of parameter space and evaluate $P(W|Data)$ at each grid-point. This is still expensive, but it does not involve any gradient descent and there are also no local optimum issues. We are not following a path in the space, we are just evaluating a set of points in the space. Once we've decided on the posterior probability to assign to each grid-point, We then use them all to make predictions on the test data. That's also expensive. But when there isn't much data, it'll work much better than maximum likelihood or maximum a posterior.

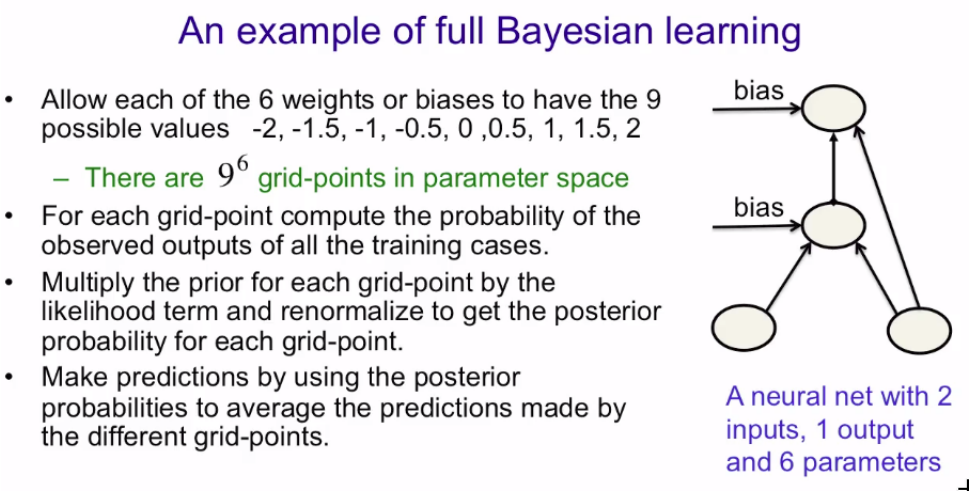


Figure 4.19: An example of full Bayesian learning.

- Full Bayesian learning practical for neural networks that have thousands, and perhaps even millions of weights can be possible with **Markov Chain Monte Carlo method (MCMC)**. That uses a random number generator to move around the space of weight vectors in a random way, but with a bias towards going downhill in the cost function. If we do this right, we get a beautiful property, which is that we sample weight vectors in proportion to their probability in the posterior distribution. And that means by sampling a lot of weight factors, we can get a

good approximation to the full Bayesian method. The number of grid points is exponential in the number of parameters. So we can't make a grid for more than a few parameters. This is enough data so that most of the parameter vectors are very unlikely. Only a tiny fraction of the group points will make a significant contribution to the predictions. We can just evaluate this tiny fraction. Idea is that it may be good enough to just sample weight vectors according to their posterior probabilities. In standard backpropagation, we keep moving the weights in direction that decreases the cost - so in the direction that increases log likelihood plus the log prior, summed over all training cases. Eventually, the weights settle into a local minimum or get stuck on a plateau, or just move so slowly that we run out of patience. Other approach, for sampling weight vectors, is that we add some Gaussian noise to the weight vector after each update. So the weight vector will never settle down. It keeps wandering around, but it tends to prefer low cost regions of the weight space. We may save the weights after each 10,000 steps for example. Wonderful property of Markov Chain Monte Carlo is, that if we use just the right amount of noise, and if we let the weight vector wander around for long enough before we take a sample, we will get an unbiased sample from the true posterior over weight vectors.

- So, we learned a probability distribution over parameters of the model. Then, at test time, we can use this distribution to get predictions with the highest possible accuracy by sampling a lot of parameters using some sampling procedure (such as MCMC) and average the predictions obtained by using each parameter setting separately. This method makes sure that we use a lot of models and also choose the models in proportion to how much we can trust them.
- Very clever idea from 2012 - **full Bayesian learning with mini batches**. Sampling noise (because we are using mini-batches) is the noise that an Markov Chain Monte Carlo method needs! And because of this, it is possible to use full Bayesian learning with lots of parameters!

4.1 Types of neural networks

ANN can be classified into multiple groups based on their characteristic properties.

Based on architecture

- **Fully connected network**. All neurons are interconnected.

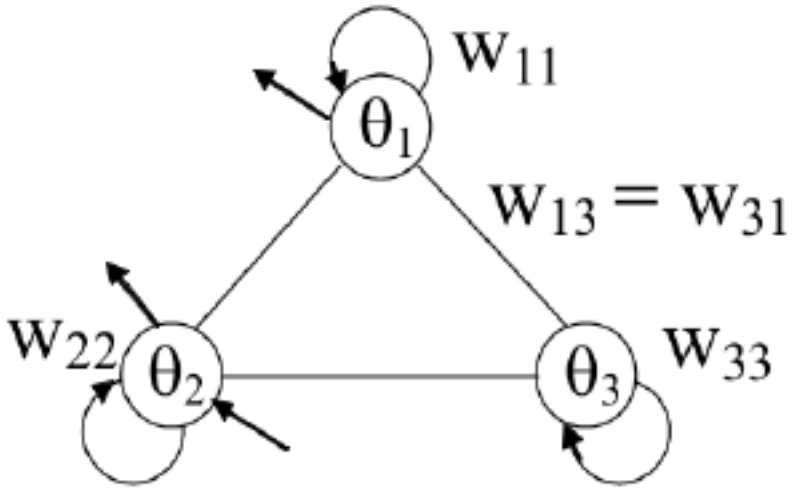


Figure 4.20: Fully connected network example.

- **Fully connected symmetric network**. The same as type above, but weights between 2 neurons are in each direction the same ($w_{ij} = w_{ji}$).
- **Multi-layer network**. Neurons are divided into individual layers. There are input, output, and hidden layers.

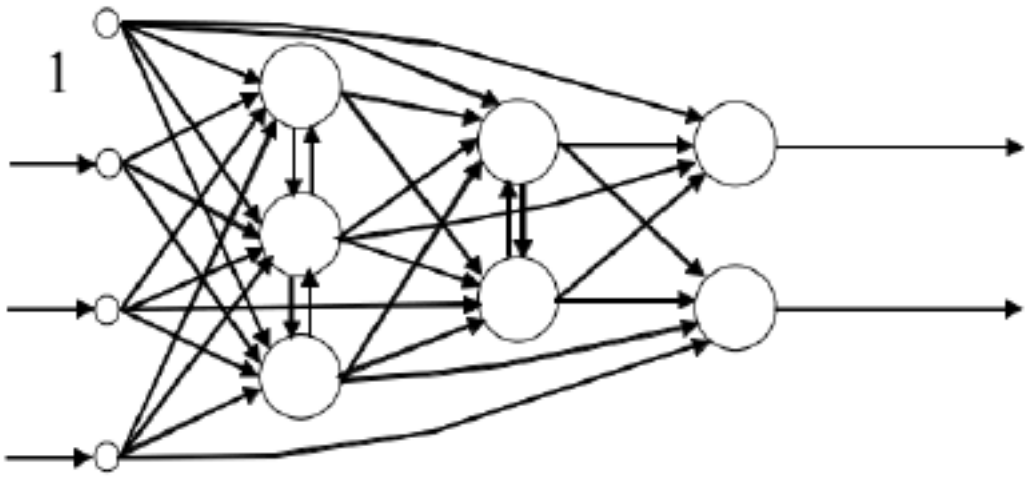


Figure 4.21: Multi-layer network example with 1 input, 1 output, and 3 hidden layers.

- **Acyclic network**. The same as multi-layer network, but without any cycles.
- **Forward network**. Acyclic network, but neurons in a given layer are connected only with neurons from the next layer. This architecture is the most common one. There are no loops, information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops. However, there are other architectures where this is possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

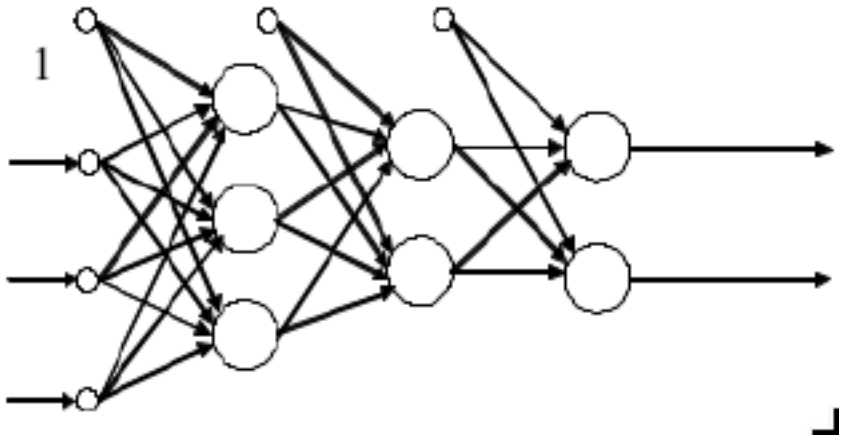


Figure 4.22: Forward network example with 1 input, 1 output, and 3 hidden layers.

Based on a way of learning

- **Correlation learning** - direct application of Hebb principle.
- **Competitive learning** - weights are updated only to a winning neuron.
- **Adaptation learning**
  - With teacher (supervisor) - training data are tuples - input, and desired output. Weights update is based on the output of neural network and expected output.
  - Without teacher (unsupervised) - no outside criterion for estimating the correctness of results. Training data contain only input values.

Based on application

There are lot of applications possible: **classification**, **clustering**, **vector quantization** (assign an input vector to the closest representative vector), **association**, **function approximation**, **prediction**, **identification of system** (approximation of behavior of some system), and **optimization**.

Based on a way of computing

If neurons are changing their inner state independently, or their calculation is controlled centrally are divided to

- **synchronous** - 1 timestep = update of all neurons in the network.
- **asynchronous** - 1 timestep = update of only 1 neuron in the network.

Basic categories of deep neural networks based on architecture, learning methods, and application

- **Deep neural networks with unsupervised approach or with generative learning.** For capturing correlations of observed data for pattern analysis or for data synthesis. These networks are trying to find correlations in data based on data itself, without any class labels.
  - **Energy-based deep models**. Typical model in this category is Deep autoencoder. This is a good choice for pre-training of another deep neural network. There are many types of autoencoders, for example another one is called Denoising autoencoder, that removes noise in data.
  - **Deep Boltzmann machine** is another type. This is a special case of more general **Boltzmann machine**. The variables in each layer are not interconnected. Always the next hidden layer captures more and more complex correlations in data, and therefore they have a potential to learn complex inner data representations. **Restricted Boltzmann machine** is when we restrict the number of hidden layers to 1. This last model is useful for initialization of weights in another neural network.
- **Deep neural networks with supervised approach.** These networks are trying to capture dependencies of input data and their labels. These networks are sometimes called **Discriminative deep networks**.
- **Hybrid deep neural networks.** These are using both generative and descriptive elements. Generative element is only a helpful element for final, discriminative model. Here, generative model is used for better optimization of a given problem, for example they are used for initialization of weights of discriminative model.

4.2 Perceptron

- Perceptron is a single layer neural network and a multi-layer Perceptron is called Neural Networks[6].
- It is a linear (binary) classifier. Supervised. Developed in the 1950s and 1960s by Frank Rosenblatt, inspired by earlier work of Warren McCulloch and Walter Pitts.
- **Weights show the strength of the particular node.**
- **A bias value allows to shift the activation function curve up or down. In more biological terms, bias is a measure of how easy it is to get the Perceptron to fire. For a Perceptron with a really big bias, it's extremely easy to output a '1'. If the bias is very negative, then it is very difficult to output '1'.**
- Hypothesis:

$$y(x) = f(\theta^T x) \tag{4.15}$$

where

  - θ_0 is null coefficient of vector θ (vector of weights, w in the figure below).
  - $x_0 = 1$ (always), and x are input values usually in the interval [-1, 1].

[6] https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

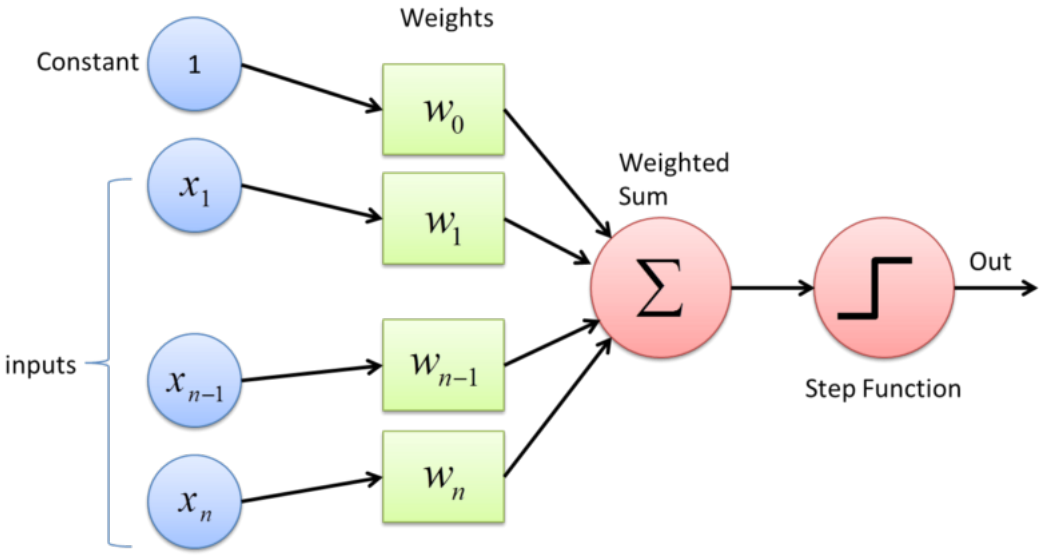


Figure 4.23: A scheme of Perceptron. It has several binary inputs and produces a single binary output (0 or 1 and it is determined by whether the weighted sum is less then or greater than some threshold value, which is again a parameter of the neuron, as the weights).

- In more precise algebraic terms:

$$output = \begin{cases} 0 & if \ \sum_j w_j x_j \leq threshold \\ 1 & if \ \sum_j w_j x_j > threshold \end{cases} \tag{4.16}$$

which can be simplified (we use dot products instead of sums, and $b = -threshold$):

$$output = \begin{cases} 0 & if \ w.x + b \leq 0 \\ 1 & if \ w.x + b > 0 \end{cases} \tag{4.17}$$

- Activation function is hard limiter (-1 or +1).

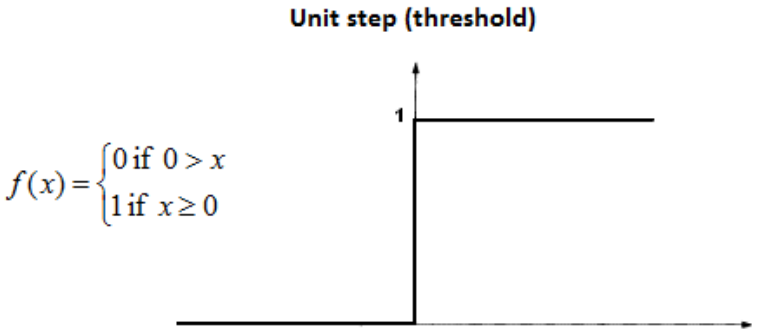


Figure 4.24: Hard limiter function.

- Iteratively go through all the learning samples, when the sample is classified incorrectly, change the weight vector:

$$\theta^{\tau+1} = \theta^{\tau} + x_n * t_n \tag{4.18}$$

- Learning algorithm in example:
  - Suppose we have a 2-dimensional input $x = (0.5, -0.5)$ connected to a neuron with weights $w = (2, -1)$ and the bias $b = 0.5$. Furthermore, suppose the target for x is $t = 0$. In this case, we will use a binary threshold neuron for the output so that $y = \begin{cases} 1 & if\, x^T w + b \geq 0 \\ 0 & otherwise \end{cases}$
  - The weights and bias be after applying one step of the perceptron learning algorithm will be: $w = (1.5, -0.5)$, $b = -0.5$. This is because target should be 0, but perceptron outputted 1. So all weights (and bias) will be **decreased** by a given input value. If there is an opposite situation, that target was 0 and it should be 1, then we would **increase** weights (and bias) by input values.

    In this example, b was 0.5 and there was a value 1. So $0.5-1 = -0.5$. Similarly with the first weight, it was equal to 2 and there was 0.5 on its input. So $2 - 0.5 = 1.5$. Analogically with the second weight, $-1 - (-0.5) = -0.5$.
  - So, Perceptron convergence procedure works by ensuring that every time the weights change, they get closer to every "generously feasible" set of weights. However, this type of guarantee cannot be extended to more complex networks in which the average of 2 good solutions may be a bad solution. So multi-layer neural networks uses something different than perceptron learning procedure. "They should never been called multi-layer perceptrons. It is my fault and I am sorry.", Geoffrey Hinton. Instead of weights get closer to a good set of

weights - the actual output values get closer to target values. That is done in MLP and it means, that there can be a different sets of weights that work well, averaging two good sets of weights may give a bad set of weights. See smoothness of sigmoid function.

4.3 Multi-layer Perceptron

This is basically a perceptron with at least 1 hidden layer, and the activation function can be different, for example **sigmoid**. So it is a very different model than a simple perceptron, and even the learning procedure is different. Some details are described in the beginning of this chapter.

- If we have a small change in some weight (or bias) in the network, it would be good to cause only a small corresponding change in the output from the network. And this property makes learning possible. This is not possible with a simple perceptron, or when a network contains perceptrons. The reason is, that small change in the weights or bias of any single perceptron can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. And this flip may then cause the behavior of the rest of the network to completely change in some very complicated way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to desired behavior. This can be overcomed with using **sigmoid neuron**. They are similar to perceptrons, but modified so hat small changes in their weights or bias will cause only a small change in their output.
- **BTW, the same weights and biases on all layers all applied to all examples (in a given mini-batch or batch, ...) during the forward pass. This is not what happens in a living organism!**

4.4 Radial Basis Function Network

- This is three-layer feedback network (1 hidden layer), in which each hidden unit implements a radial activation function, and each output unit implements weighted sum of hidden units outputs.
- Training is divided into 2 stages:
  - centers and widths of the hidden layer are determined by clustering algorithms
  - weights connecting the hidden layer with the output layer are determined by Singular Value Decomposition (SVD) or Least Mean Squared (LMS) algorithms.
- The problem of selecting the appropriate number of basis functions (which control the complexity and the generalization ability of RBF networks) remains a critical issue for RBF networks. Too few basis functions - RBF networks cannot fit the training data properly due to limited flexibility. Too many basis functions yield poor generalization abilities, they are too flexible.

4.5 Bayesian Neural Networks

- Conventional ANN aren't well designed to model uncertainty associated with the predictions they make.[7] Conventional ANN has parameters (weights and biases, scalar values) which is trying to optimize via maximum likelihood estimation. On the other hand, Bayesian approach focuses on distributions associated with each parameter, generally referred to as posterior densities, estimated using Bayesian rule.

- Having a distribution instead of a single value is a powerful thing. For one, it becomes possible to sample from a distribution many many times and see how this affect the predictions of the model. If it gives consistent predictions, sampling after sampling, then the net is said to be "confident" about its prediction.

- Mostly used for regression, but we can use this network also on classification tasks.

- It is basically a typical ANN, with uncertainties, probabilities, and relationships in data. We need to take into account uncertainties of inputs and outputs - this is all different in comparison to traditional ANN, where we don't have to fully understand what a network is doing.

- In practical terms, uncertainties mentioned earlier are in weights of ANN. So, instead of 1 solution for NN weights we will take into account the whole space of solutions - NN weights will be random variables. For each solution we will define its probability as $P(w|D, H_i)$, which defines with how probability the weights w are correct in model H_i(so output y) given input data D (so input x). It can be calculated as follows, with intergral which requires marginalization over all possible values that parameters (weights w) can assume in the model, and that become quickly too hard to compute, which is often not doable in practice.

$$p(w|x, y) = \frac{p(x, y|w).p(w)}{\int p(y|x, w).p(w)\, dw} \tag{4.19}$$

- Because of computational requirements, pseudo-numerical approaches can be chosen instead to the solution to integrals in Definition 4.19:
  - Approximating integrals with **Markov chain Monte Carlo** (MCMC, a class of algorithms for sampling from a probability distribution) - instead of computing exact integrals.
    - Pros & Cons: this method has great results, however, it is the slowest one (it takes a long time to converge).
  - Using black-box **variational inference** (e.g. using edward).

[7] https://medium.com/@joeDiHare/deep-bayesian-neural-networks-952763a9537?fbclid=IwAR1tTCMUIkg1DNxsEHI8bKFMmESkc5-hD5ixXriNOsUMWZocotnbdiYBYXw

    - Variational inference is an approach to estimate a density function by choosing a distribution we know (e.g. Gaussian) and progressively changing its parameters until it looks like the one we want to compute, the posterior. Changing parameters no longer requires mad calculus; it's an optimization process, and derivatives are usually easier to estimate than integrals. This "made-up" distribution we are optimizing is called variational distribution.
    - Pros & Cons: Faster than previous approach, but it still might get slow for very deep Bayesian net, and performance isn't always guaranteed to be optimal.
  - Using **Monte Carlo dropout** (MC dropout). This is Bayesian interpretation of dropout regularization technique.[8].
    - In MC dropout, it could be used to perform variational inference where the variational distribution is from a Bernoulli distribution (states are "on" and "off") and "MC" refers to the sampling of the dropout, which happens in a "Monte Carlo" style.
    - In practice, turning a conventional network into a Bayesian one via MC dropout can be as simple as using dropout for every layer during training AS WELL AS testing; this is equivalent to sampling from a Bernoulli distribution and provides a measure of model's certainty (consistency of predictions across sampling)
    - Pros & Cons: It is easy to turn an existing deep net into a Bayesian one. it is faster than other techniques, and does not require an inference framework. However, sampling at test time might be too expensive for computationally-demanding (e.g. real-time) applications.

- As Occam's Razor defines, we can compare models between each other, based on evidence. So we will find the most probable model and this model is always the simplest one. **The simplest possible model can generalize well and such model has less problems with overfitting. On the other hand, it can generalize too much.**
- **Another advantage of these networks is that we don't need to divide our data to training, validation, and testing set.**
- Learning may use Kullback-Leibler divergence. This tells us, how much a given system (or distribution) differs from some other referential one, with the usage of entropy. If this divergence is 0, then both systems are the same. More far away from 0 the divergence is, systems are more different. KL divergence is not

[8] In dropout, neurons are randomly turned on or off during training to prevent the network to depend on any specific neuron.

symmetrical.[9] We can compute KL divergence as difference between entropies of system A and system B.

- BNN is alternative to ANN, but learning is very different. Learning in ANN starts from "nothing" - initial parameters are set to 0, or values close to 0, and a learning algorithm sequentially improves its output so that it matches training data. BNN starts in state, where a network represents all functions BNN is able to describe. Sequentially it restricts this set until there are only ones which sufficiently enough represent training data.
- Regarding speed, BNN is ~2x slower than ANN, but BNN may converge faster.

[9] KL distance between A and B is not the same as between B and A.

4.6 Hopfield Network

- Fully connected recurrent symmetric network, which uses Linear Basis Function and as an activation function it can use logistic sigmoid or hyperbolic tangents, or (if we want to model discrete Hopfield network, then Heaviside Step Function - something step-like).
- Inputs can be binary or bipolar.
- Learning is based on Hebb principle.
- It is recurrent network, connections between neurons are oriented cycles.

4.7 Evolutionary Algorithms and Artificial Neural Networks

- Evolutionary algorithm are capable of optimization of complex problems. Neural networks needs to set a great amount of parameters. Additionally, there is no exact approach how to choose the best topology.
- Encoding can be divided into 2 groups:
  - **Direct encoding** - genes represent individual neurons and their connections
  - **Indirect encoding** - usually determines, how to create a given network. Indirect encoding can be beneficial for creating more compact genomes, which can generate much bigger networks, or repetitive occurrence of the same subnets or patterns.
- There are many challenges in such connection of these two:
  - **Competing Convention** - a situation, when there exist multiple different genomes for a given neural network, but they differ only in permutation of neurons in genome. During crossover (genetic operation), there can be created useless individuals of population. The bigger a network is, the more such permutations exist, and more "corrupted" individuals are being created.
  - **Length of genomes is not the same**. Standard genetic algorithms work with the same length of genomes. For this and previous reason (from Competing Convention problem), crossover operation is not being used for neuroevolutionary algorithms - instead, only mutation is mostly used. However, there are research works, that with proper encoding and usage of more complex crossover operations, it is possible to evolve a neural network.
- A few algorithms: **GNARL** (GeNeralized Acquisition of Recurrent Links), or more recent **EANT** (Evolutionary Acquisition of Neural Topologies), **CE** (Cellular Encoding), **NEAT** (NeuroEvolution of Augmenting Topologies).

4.8 Fuzzy Neural Network

- **Fuzzy set vs Crisp set**
  - **Crisp set i**s a set as a collection of elements—e.g $A = x|x \in R and x > 0$ where a general universe of discourse is established (in our example, set of real numbers) and each distinct point in that universe is either a part of our set or not. Fuzzy sets are also similar to crisp sets but they are more generalized. So crisp sets can be said to be a specific type of fuzzy set. Here in crisp sets however, an element cannot be just a partially part of a set. Let us define the value 0 and 1 as a **membership value** $mA(x)$ for crisp sets. It is a **binary value** i.e it can only be 0 or 1 for crisp sets.
  - **Fuzzy set** is similar to crisp set, but the **membership value** is a **continuous value** between [0, 1]. We can define a particular value in our universe of discourse as being part of that set with partial participation.

Assuming X to be the universe of discourse, A ∈ X and B ∈ X, the fuzzy set operations are defined as follows:

a. *Comparison (Is A = B?)*: `iff mA(x) = mB(x) for all x in X`

b. *Containment (Is A ⊂ B?)*: `iff mA(x) < mB(x) for all x in X`

c. *Union:* `m(A ∨ B)(x) = max(mA(x), mB(x)) for all x in X`

d. *Intersection*: `m(A ∧ B)(x) = min(mA(x), mB(x)) for all x in X`

Figure 4.25: Definition of all basic fuzzy set operations.

- There are many types of fuzzy neural networks which can work very differently:
  - **Neural network based on fuzzy operators** - Fuzzy Hopfield, FAM (Fuzzy Associative Memory) - these found application in pattern recognition, classification, and signal processing. For example, one of the first was Fuzzy Min-Max Classifier, original idea from 1992.
  - **Regular fuzzy neural network** works with an idea, that each neuron is basically a fuzzy neuron - which weights, inputs, and outputs are fuzzy sets (and there is also fuzzy arithmetic used). Such network is basically a feedforward network. Learning algorithm is very important part of them.
  - **Neuro-fuzzy system.** Combination of two subsystems - neural networks and fuzzy regulator. Based on their cooperation, these systems can be divided into 3 groups:

  - **cooperative**, where both systems work independently - neural net learns (from training data) rules for fuzzy system, which is using these rules as a solution to final problem. For learning, there can be Self Organizing Maps, or Fuzzy associative memories.
  - **concurrent**, where neural net is used for preprocessing data to fuzzy system. But the architecture can be an opposite - first a fuzzy system (for data preprocessing) and then neural network.
  - **hybrid**, also known as Inferential neural networks. This is one of the most popular approach.

- Fuzzy Min-Max Classifier is working with hyperboxes.[10]
  - Learning of them is in two phases:
    1. **Expansion phase.** The first thing we do is find the most suitable hyperbox for expansion. We take all the hyperboxes belonging to the class Y and calculate their membership values. The hyperbox with the maximum membership value is the most suitable candidate for expansion.
    2. **Contraction phase.** This is for avoiding overlapping of hyperboxes. Suppose that from the previous phase we found a suitable hyperbox meeting the expansion criteria and we expanded the box. We need to make sure that this expanded box does not overlap with another hyperbox belonging to a different class. Overlap between hyperboxes of same class is allowed, so we'll only check for overlap between expanded box and hyperboxes of different classes.
  - So we apply the learning algorithm and derive hyperboxes based on the training data (there are "patterns" = tuples of n points = n-dimensional space on the input and from these patterns, we construct hyperboxes).

[10] https://medium.com/@apbetahouse45/understanding-fuzzy-neural-network-with-code-and-graphs-263d1091d773

5 Deep Learning

They are separated from previous category because of a great popularity and massive growth in the field. They are concerned with building much larger and more complex neural networks.

- **Deep neural network is simply a feedforward network with many hidden layers.**[1]
- Ideally - start with 1 hidden layer and then, with the usage of CV, add more layers / hidden neurons.
- Any function is NN and any NN is a function.
- Very deep NN - local minimums are kinda similar.
- Extraction of features and learning can be done by the algorithm itself. Usually this requires more data and is more computationally expensive, since we did not provide any features. And it usually works better, since we as humans can not know what is the best and what is not for computers, since they have different perspective.
- Consider object recognition/detection problem. Representation of some hidden layers = deeper and deeper layer generates higher-level features. From the beginning, we perhaps do not understand it by our senses, but later on, the outcome perhaps will make sense.
  - For example, in the first steps, DNN will learn to find edges in an image. In the next layers, DNN will learn that these edges can be combined to something more complicated, and later on DNN will learn something more sophisticated, detection of some complicated things (we may be able to see and even understand them) and so on.
  - Actually, first hidden-layers tend to capture universal and interpretable features, like shapes, curves, or interactions that are very often relevant across domains.
- Major DL trends, from Nuts and Bolts of DL[2]:
  - General DL

[1] https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w

[2] https://www.youtube.com/watch?v=F1ka6a13S9I&t=421s&index=2&list=WL

  - Sequence models - RNN, LSTM, GRU
  - Image 2D/3D - CNN
  - Other - Reinforcement learning, Unsupervised learning
- There exist typically 2 types / techniques:
  - Convolutional Neural Networks (CNNs)
  - Recurrent Neural Networks (RNNs)
- training a NN:
  - Steps:
    1. Choose suitable architecture - how many hidden layers and how many elements. The number of input units - number of features. Number of output units - number of classes. Hidden - either 1 or more (usually the same number as the number of features) and then number of units on each layer.
    2. Random weight initialization (thetas)
    3. Implement forward propagation for obtaining hypothesis for each input
    4. Implement code for calculating cost function e) implement backpropagation for calculating partial derivations of cost function.
  - Typically we iterate through all training examples and perform forward a back propagation on every training example. So we are obtaining all activations and deltas for all the layers.
  - Then we can calculate partial derivation of cost function over theta.
  - Then we use gradient checking for comparing derivations from the previous step: backprop vs numerical result of gradient of cost function. They should have similar values. Then we can disable gradient checking.
  - Then we use gradient descent or other advanced optimization methods (LB of GS, contract gradient) s backpropagation for minimizing of the cost function.
- Various layers, for example:
  - dropout - randomly set weight of some neuron to 0.
- **Vanishing/exploding gradients**
  - One of the problems of training neural network, especially very deep neural networks, is data vanishing and exploding gradients. What that means is that when you're training a very deep network your derivatives or your slopes can sometimes get either very, very big or very, very small, maybe even exponentially small, and this makes training difficult.

- In RNN, vanishing gradient problems are occurring more often than exploding gradients. Vanishing gradients are more difficult to detect and resolve. In exploding gradients, values will become **NaN**, which is a result of a computation overflow) and the network stops working.
- If the weights are bigger than identity matrix, then with a very deep network the activations can explode (exploding gradients). And if weights are just a little bit less than identity matrix, and you have a very deep network, the activations will decrease exponentially (vanishing gradients).
- A proper random initialization helps to reduce this problem. How to initialize ANN depends on activation functions (for instance, Xavier initialization when the activation function is *tanh*).
- For vanishing gradients problem, it is solved by LSTMs and GRUs, and if you're using a deep feedforward network, this is solved by residual connections.
- Also, for exploding gradients, there is a solution - **gradient clipping** (mostly used in RNN) - it will clip the gradients between two numbers to prevent them from getting too large.
- Also, for learning there are modified algorithms for RNNs to deal with vanishing/exploding gradients - Backpropagation Through Time (BPTT), or Hebbian learning.

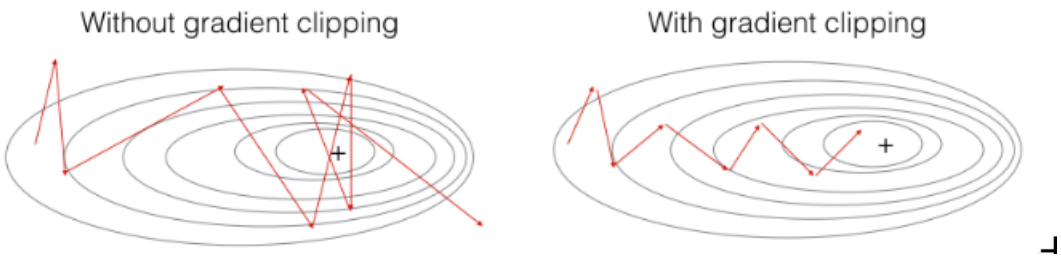


Figure 5.1: Visualization of gradient descent with and without gradient clipping, in a case where the network is running into slight "exploding gradient" problems. There are different ways to clip gradients; a simplest one is an element-wise clipping procedure, in which every element of the gradient vector is clipped to lie between some range $[-N, N]$. More generally, you will provide some max value (say 10), and if any component of the gradient vector is greater than 10, it would be set to 10; and if any component of the gradient vector is less than -10, it would be set to -10. If it is between -10 and 10, it is left alone.

- **Greedy Layer-Wise Training** - for training deep neural networks, for initialization of weights with unsupervised method as a pre-training = for example, we can

use Autoencoder or Restricted Boltzmann machine. Each hidden layer is trained separately. When a given layer is trained, the output will be the input for the next layer, that is also trained separately. This is only a replacement to random initialization of weights though! But probably, if the data dependencies are not so sophisticated, it is better to use a simple ANN (with more deep network, there is a bigger change to get stuck in local optima - overfitting). But deep network pre-trained with Autoencoder or RBM, vs classical neural net - the first iterations have better performance, but then if the network is more deep, it overfits more easily.

5.1 Autoencoder

- Two-layer neural network, where the length of input is the same as the length of output. Also, the first layer has less neurons than the output layers. It is basically a feed-forward neural network with encoder-decoder architecture. It is trained to reconstruct its input, so the training example is a pair (x, x). We want the output $\hat{x}$ of he model $f(x)$ to be as similar to the input x as possible.
- Cost function is usually MSE (when features can be any number) or the binary cross-entropy (features are binary and units of the last layer of the decoder have the sigmoid activation function).
- The aim is to encode input data and then decode such data again: $\hat{x} = decode(encode(x))$.
- Deep autoencoders are from 1980s, but they simply couldn't be trained well enough for them to do significantly better than PCA. After methods of pre-training deep networks one layer at a time were developed, these methods were applied to pre-training deep autoencoders, and for the first time, researchers got much better representations out of deep autoencoders than we could get from principal components analysis. Because it turned out that it is very difficult to optimize deep autoencoders using backpropagation - small initial weights will cause that back-propagated gradient dies. Now we have much better way to optimize them - using unsupervised layer-by-layer pre-training. The first successful training/using of deep autoencoders was in 2006.
- Autoencoders are also being used for data comprimation.
- Using backpropagation to generalize PCA.
- The objective function of an autoencoder is to reconstruct its input, i.e., it is trying to learn a function f, such that $f(x) = x$ for all points x in the dataset. Clearly there is a trivial solution to this. ff can just copy the input to the output, so that $f(x) = x$ for all x. However, The network does not learn to do this, because it has constraints, such as bottleneck layers, sparsity and bounded activation functions which make the network incapable of copying the entire input all the way to the output.
- The decoder network may have different number of layers and hidden units as the encoder network as long as it produces output of the same shape as the data, so that we can compare the output to the original data and tell the network where it's making mistakes. Autoencoder is just neural network that is trying to reconstruct its input. There is hardly any restriction on the kind of encoder or decoder to be used.
- Another way of extracting short codes for images is to hash them using standard hash functions (this is called **semantic hashing**). These functions are very fast

to compute, require no training and transform inputs into fixed length representations. It is more useful to learn an autoencoder to do this, because autoencoders can be used to do semantic hashing, where as standard hash functions do not respect semantics , i.e, two inputs that are close in meaning might be very far in the hashed space. Autoencoders are smooth functions and map nearby points in input space to nearby points in code space. They are also invariant to small fluctuations in the input. In this sense, this hashing is locality-sensitive, whereas general hash functions are not locality-sensitive.

- RBMs and single-hidden layer autoencoders can both be seen as different ways of extracting one layer of hidden variables from the inputs. However, they are different:
  - RBMs are undirected graphical models, but autoencoders are feed-forward neural nets.
  - RBMs define a probability distribution over the hidden variables conditioned on the visible units while autoencoders define a deterministic mapping from inputs to hidden variables.
- Autoencoders seem like a very powerful and flexible way of learning hidden representations. You just need to get lots of data and ask the neural network to reconstruct it. Gradients and objective functions can be exactly computed. Any kind of data can be plugged in. However, these models have also limitations. There is no simple way to incorporate uncertainty in the hidden representation $h = f(v)$. A probabilistic model might be able to express uncertainty better since it is being made to learn $P(h|v)$.
- Boltzmann machine is basically a strongly regularized Autoencoder.
- **Denoising autoencoder** (2008) is a special type of autoencoder add a noise (usually a Gaussian noise) to the input vector by setting many of its components to zero (like dropout, but for inputs rather than for hidden units). Pre-training is very effective if we use stack of denoising autoencoders. It is as good (or even better) than pre-training with RBMs.
  - **A ladder network** is a denoising autoencoder with an upgrade. The encoder and the decoder have the same number of layers. The bottleneck layer is used directly to predict the label (using the softmax activation function). The network has several cost functions. In the ladder network, not just the input is corrupted with the noise, but also the output of each encoder layer (during training). When we apply the trained model to the new input x to predict its label, we do not corrupt the input.
- **Contractive autoencoder** is another kind of autoencoder (2011) and it is another way to regularize an autoencoder for trying to make the activities of the hidden units as insensitive as possible to the inputs. But they cannot just ignore the

inputs because they must reconstruct them. This can be achieved by penalizing the squared gradient of each hidden activity with respect to the inputs. They work well for pre-training.

- **Conclusions about pre-training (Boltzmann Machines + Autoencoders):**
  - There are now many different ways to do layer by layer pre-training that discovers good features. When our data set does not have a huge number of labels, this way of discovering features before you ever use the labels is very helpful for the subsequent discriminative fine tuning. It discovers the features without using the information in the labels, and then the information in the labels is used for fine tuning the decision boundaries between classes. It's especially useful if we have a lot of unlabeled data so that the pre-training can be a very good job of discovering interesting features, using a lot of data.
  - For very large labeled data sets however, initializing the weights that are going to be used for supervised learning by using unsupervised pre-training is not necessary, even if the nets are deep. Pre-training was the first good way to initialize the weights for deep nets, but now we have lots of other ways. However, even if we have a lot of labels, if we make the nets much larger again, we'll need pre-training again.

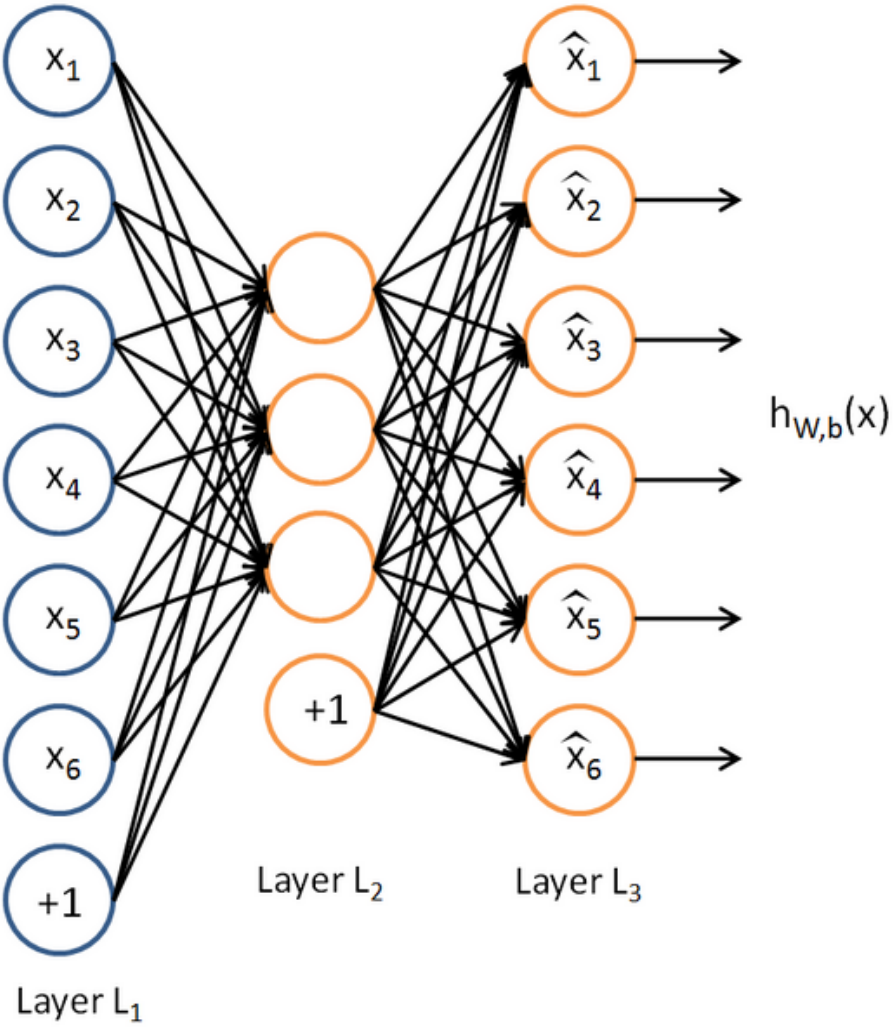


Figure 5.2: An example of autoencoder scheme. The input 6 values are encoded into 3 values (generalization) and then from these 3 values, the network reconstructs the input data $\hat{x}$. The first hidden layer represents the output of comprimation. Decomprimation part works with this first hidden layer resulting in output layer.

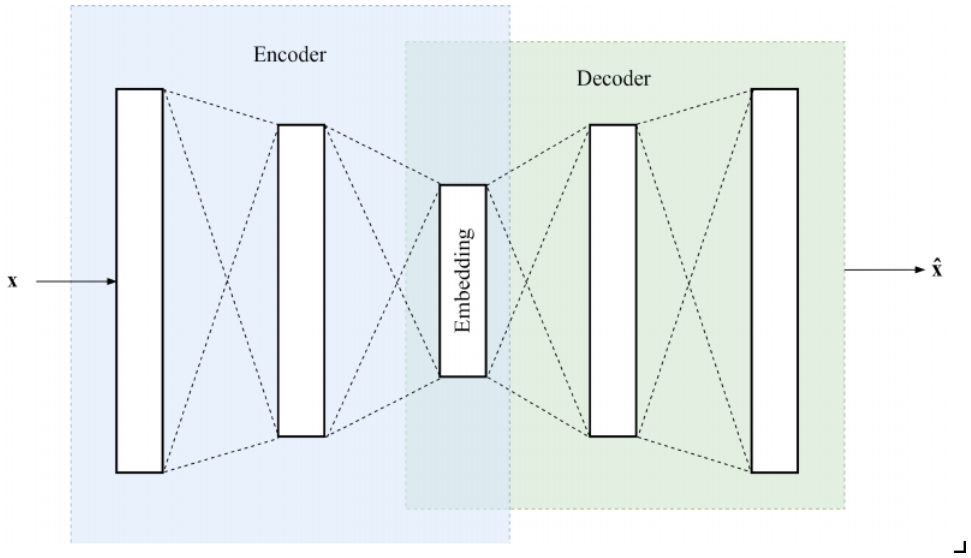


Figure 5.3: Autoencoder.

5.2 Boltzmann Machine

- Also known as stochastic Hopfield networks.
- When we pick a new state s_i for unit i, we do so in a stochastic way: $p(s_i = 1) = \frac{1}{1+exp(-\Delta E/T)}$, and $p(s_i = 0) = 1 - p(s_i = 1)$. Here, ΔE is the energy gap, i.e. the energy when the unit is off, minus the energy when the unit is on. T is the temperature. We can run our system with any temperature that we like, but the most commonly used temperatures are 0 and 1.
- When we want to explore the configurations of a Boltzmann Machine, we initialize it in some starting configuration, and then repeatedly choose a unit at random, and pick a new state for it, using the probability formula described above. When a Boltzmann Machine reaches an energy minimum, then if it's cold it will stay there. If it's warm, it might move away from it (warm one can end up anywhere, because it is truly stochastic).
- What the algorithm is trying to do is build a model of a set of input vectors, though it might be better to think of them as output vectors. What we want to do is maximize the product of the probabilities, that the Boltzmann machine assigns to a set of binary vectors in the training set. This is equivalent to maximizing the sum of the log probabilities that the Boltzmann machine assigns to the training vectors.
- So what exactly is being learned in the Boltzmann Machine learning algorithm? **Parameters than define a distribution over the visible vectors.**

5.3 Restricted Boltzmann Machines

- More simple model than Deep Boltzmann Machine (DBM) which was designed to be more easier - learning of Deep Boltzmann Machine is computationally very expensive.
- Unsupervised method. Application in classification, dimensionality reduction, topic modeling. They become very popular because classical Boltzmann Machines are computationally very expensive. RBM learn much faster. In RBM, connections between each visible neurons were removed as well as were removed connections between hidden neurons.
- There are no connections between hidden units. This makes it very easy to get the equilibrium distribution of the hidden units if the visible units are given. There are also no connections between visible units. This no connectivity (between hidden units as well as visible units) makes it possible to update all hidden units in parallel given the visible units (and vice-versa). Moreover, only one such update gives the exact value of the expectation that is being computing.

- Probabilistic graphical model that can be represented as stochastic neural network. It is a special case of Boltzmann Machine.
- Neurons consist of 2 layers - **visible** (the first layer) and **hidden** (the second layer) and they form a bipartite non-oriented graph.
- Configuration (v, h) of visible and hidden neurons has energy $E(v, h)$.
  - **Bernouli-Bernouli RBM:**

$$E(v,h) = -\sum_{k=1}^{n}\sum_{j=1}^{m} v_k h_j w_{kj} - \sum_{j=1}^{m} b_j h_j - \sum_{i=1}^{n} c_k v_k \tag{5.1}$$

    where v_i and h_j are binary values of visible and hidden neuron i and j. Variable n is a number of neurons in hidden layer, m is a number of neurons in visible layer.

  - **Gaussian-Bernoulli RBM:**

$$E(v,h) = -\sum_{k=1}^{n}\sum_{j=1}^{m} v_k h_j w_{kj} - \sum_{j=1}^{m} b_j h_j - \frac{1}{2}\sum_{i=1}^{n}(v_k - c_k)^2 \tag{5.2}$$

    where the variables and parameters have the same meaning, except that v_i are real values instead of binary ones. There are Gaussian neurons on visible layer. Visible layer consists of binary stochastic neurons, like in Bernouli-Bernouli RBM.

- For example, if we have the (binary) state of all units (both the visibles, for example $visible\,probability = \frac{1}{1+e^{-(weights * visible\,state)}}$, and the hiddens), i.e. if we have a full configuration, we can calculate the energy of that configuration, or the goodness (which is negative the energy). So, this configuration goodness can be calculated as $\frac{hidden\,state * visible\,state'.* weights}{nr\,samples}$. After that, we would like to give to some configurations a higher probability (we want to make some configurations better) by computing gradient of the goodness of a configuration, which is $\frac{hidden\,state * visible\,state'}{nr\,samples}$. Then, Contrastive Divergence gradient estimator with 1 full Gibbs update (also known as CD-1). There are multiple variations of CD-1. For example, every time after calculating a conditional probability for a unit, we sample a state for the unit from that conditional probability, and then we forget about the conditional probability. We'll sample a binary state for the visible units conditional on that binary hidden state (this is sometimes called the "reconstruction" for the visible units); and we'll sample a binary state for the hidden units conditional on that binary visible "reconstruction" state. Then we base our gradient estimate on all those sampled binary states. This is not the best strategy, but it is the simplest one. The conditional probability functions will be useful for the Gibbs update. The configuration goodness gradient function will be useful twice, for CD-1:

  - on the given data and the hidden state that it gives rise to. That gives us the direction of changing the weights that will make the data have greater goodness, which is what we want to achieve.
  - on the "reconstruction" visible state and the hidden state that it gives rise to. That gives us the direction of changing the weights that will make the reconstruction have greater goodness, so we want to go in the opposite direction, because we want to make the reconstruction have less goodness.

- This model is able to capture a probability distribution of inputs, which means that it is able to create generative model of input data.
- **Contrastive Divergence** (CD) is algorithm used for training RBM.
- Based on calculated energy of the model, we can calculate probabilities of each pair of hidden and visible neuron. Also partition function, or probability that network will assign to visible vector v, or a probability that a value of a given neuron will be equal to 1.

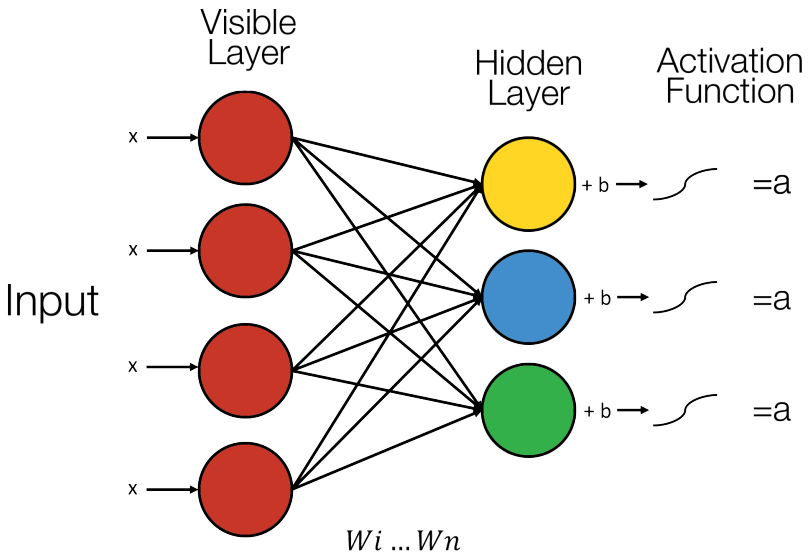


Figure 5.4: An example of Restricted Boltzmann Machine scheme. There are visible and hidden layers.

- RBM is actually the same thing as an infinitely deep sigmoid belief net with shared weights.

5.4 Deep Boltzmann Machines

- Many layers of trained RBM = stacking RBMs. They are built in the following way - hidden layer of a single RBM will be a visible layer of the next RBM. These deep neural networks are pre-trained. Pre-training is done for each layer separately for finding out the most precise model of input data.

- First, train a layer of features that receive inputs directly from pixels (or different, depending on data). Then treat the activations of the trained features as if they were pixels and learn features of features in a second hidden layer (so basically layer $v_2 = h_1$). We can repeat this as many times as we like with each new layer of features modeling the correlated activity in the features in the layer below. Resulting model is not Boltzmann Machine, but Deep Belief Net! Because bottom layer of connections are not symmetric! All edges in Boltzmann Machines must be undirected and a DBN has directed edges from the top-level RBM to each subsequent layer below. It is a graphical model that is called Deep Belief Net, where the lower layers are just like sigmoid belief nets, and the top two layers are from a Restricted Boltzmann Machine. So it is a kind of hybrid model.

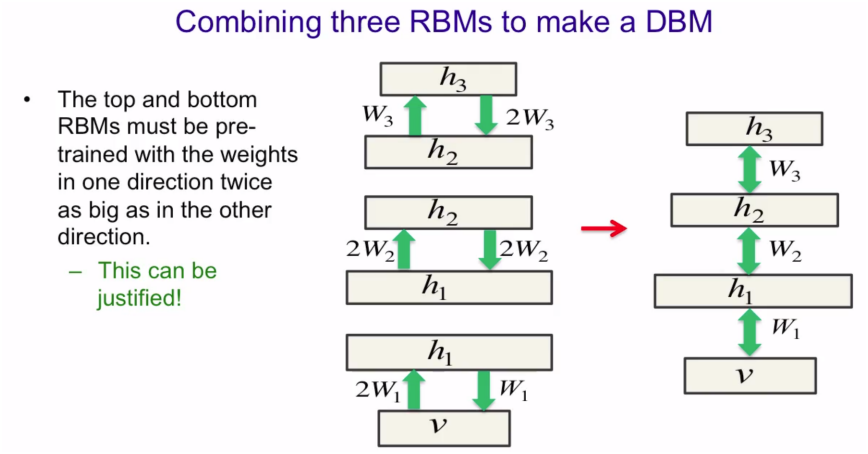


Figure 5.5: Combination of three RBM into one Deep Boltzmann Machine. However, if you train a stack of restricted Boltzmann machines and you combine them together into a single composite model what you get is a Deep Belief Net not a Deep Boltzmann machine. The trick is that the top and the bottom restrictive bowser machines in the stack have to trained with weights that it twices begin one directions the other. So, the bottom Boltzmann machine, that looks at the visible units is trained with the bottom up weights being twice as big as the top down weights. Apart from that, the weights are symmetrical. That's the opposite of what we had when we trained the first restricted Bolton machine in the stack. After having trained these three restricted Bolton machines, we can then combine them to make a composite model, and the composite model looks like this.

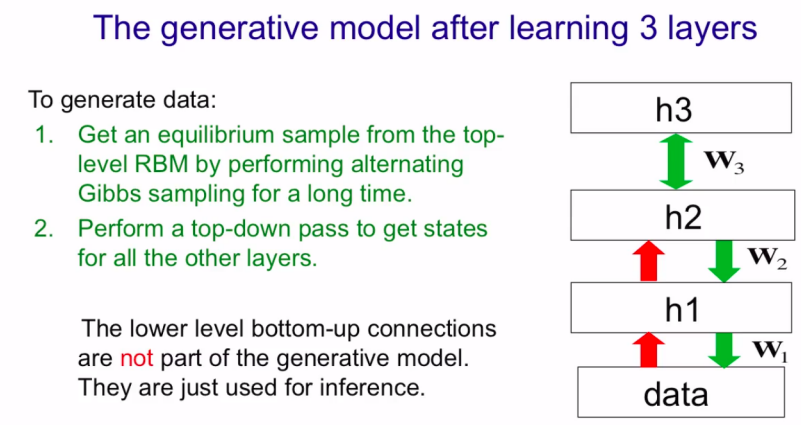


Figure 5.6: Combination of two RBM into one Deep Boltzmann Machine for generating a new data.

5.5 Belief Networks

- A belief net is a **directed acyclic graph** composed of stochastic variables (idea from 1992). They are also known as Sigmoid Belief Nets (SBNs).
- The first reason why deep belief nets are interesting, is that they are interesting example of generative models: to recognize shapes, first learn to generate images. Not only that it can read digits, it can also write them. A generative model like a DBN (deep belief net) can be used in a way that it is possible to specify the values of some of the feature neurons, and then "run the network backward", generating values for the input activations.
- The second reason why they are interesting is that they can do unsupervised and semi-supervised learning. They can learn useful features for understanding other images, even if the training images are unlabeled.
- Key component of deep belief nets are Restricted Boltzmann machines.
- We get to observe some of the variables and we would like to solve 2 problems:
  - **The inference problem** - infer the states of the unobserved variables. We can't infer them with certainty, so what we're after is the **probability distributions of unobserved variables**. And if unobserved variables are not independent of one another, given the observed variables, there is probability distributions are likely to be big cumbersome things with an exponential number of terms in.
  - **The learning problem** - adjust the interactions between variables to make the network more likely to generate the training data. So, adjusting the interactions would involve both deciding **which node is affected by which other node**, and also deciding on the **strength of that effect**.
- These are generative models, where the objective function is to model the input data rather than predicting a label.
- The idea of graphical models was to combine discrete graph structures for representing how variables depended on one another.
- There are **two types of generative neural network** composed of stochastic binary neurons:
  - **Energy based -** we connect binary stochastic neurons using **symmetric connections** to get a **Boltzmann Machine**.
  - **Causal -** we connect binary stochastic neurons in a **directed acyclic graph** to get a **Sigmoid Belief Net. It was shown in the past research, that these are slightly easier to learn than Boltzmann Machines. Also, it is easy to generate samples with this model.**

- For learning, there is **wake-sleep algorithm** and it works in two phases: positive and negative phase.
  - However, wake-sleep algorithm is a very different kind of learning, mainly because it's for directed graphical models like Sigmoid Belief Nets, rather than for undirected graphical models like Boltzmann machines.
  - The ideas behind the wake-sleep algorithm led to a whole new area of machine learning called **variational learning**. The idea is that since it's hard to compute the correct posterior distribution, we'll compute some cheap approximation to it. And then, we'll do maximum likelihood learning anyway. That is, we'll apply the learning rule that would be correct, if we'd got a sample from the true posterior, and hope that it works, even though we haven't. Now, you could reasonably expect this to be a disaster, but actually the learning comes to your rescue. So, if you look at what's driving the weights during the learning, when you use an approximate posterior, there are actually two terms driving the weights. One term is driving them to get a better model of the data. That is, to make the Sigmoid Belief Net more likely to generate the observed data in the training center. But there's another term that's added to that, that's actually driving the weights towards sets of weights for which the approximate posterior it's using is a good fit to the real posterior. It does this by manipulating the real posterior to try to make it fit the approximate posterior. It's because of this effect, the variational learning of these models works quite nicely.

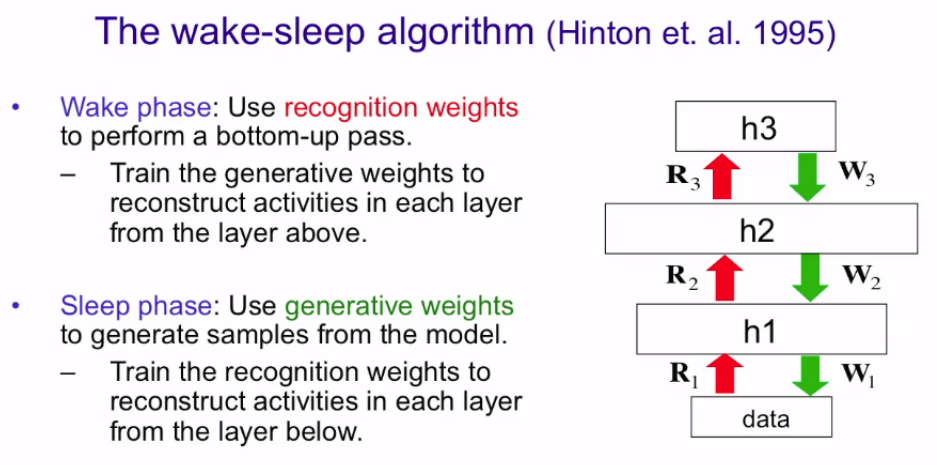


Figure 5.7: Wake sleep algorithm. It turns out that if you start with random weights and you alternate between wake phases and sleep phases it learns a pretty good model.

  - There are flaws in this algorithm
    - The recognition weights are trained to invert the generative model in

parts of the space where there is no data and it is wasteful. But it is not a big deal.

∗ The recognition weights do not follow the gradient of log probability of the data. They only approximately follow the gradient of the variational bound on this probability. And this leads to incorrect mode-averaging. It is more serious issue.

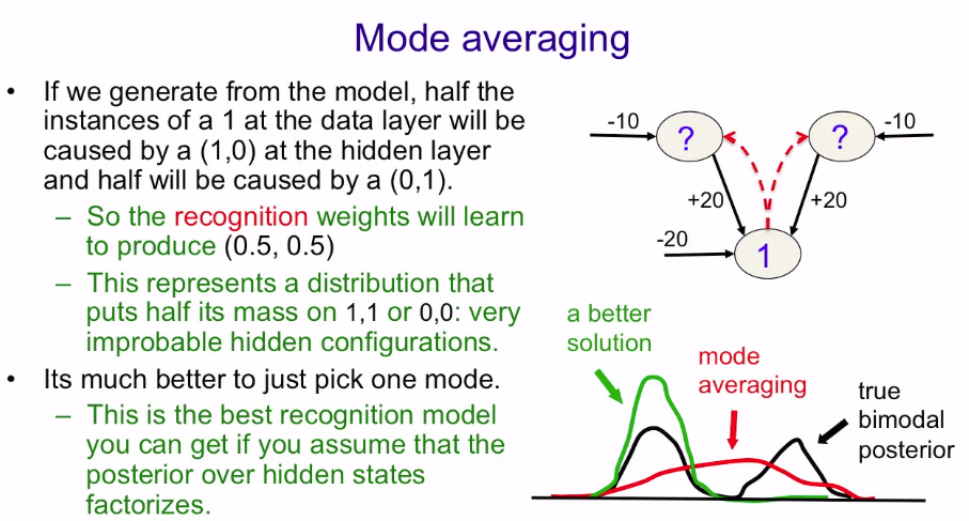


Figure 5.8: Mode averaging. Suppose we run the sleep phase, and we generate data from this model. Most of the time, those top two units would be off because they are very unlikely to turn on under their prior, and, because they are off, the visible unit will be firmly off, because its bias is -20. Just occassionally, one time in about to the -10, one of the two top units will turn on and it will be equally often the left one or the right one. When that unit turns on, there's a probability of a half that the visible unit will turn on. So, if you think about the occassions on which the visible unit turns on, half those occassions have the left-hand hidden unit on, the other half of those occassions have the right-hand hidden unit on and almost none of those occassions have neither or both units on. So now think what the learning would do for the recognition weights. Half the time we'll have a 1 on the visible layer, the leftmost unit will be on at the top, so we'll actually learn to predict that that's on with a probability of 0.5, and the same for the right unit. So the recognition units will learn to produce a factorial distribution over the hidden layer, of $(0.5, 0.5)$ and that factorial distribution puts a quarter of its mass on the configuration $(1, 1)$ and another quarter of its mass on the configuration $(0, 0)$ and both of those are extremely unlikely configurations given that the visible unit was on.

- Contrastive wake-sleep is a way of fine-tuning the model to be better at gener-

alization. Backpropagation can be used to fine-tune the model to be better at **discrimination**.

5.6 Sequence Models

- **They work with sequence data.** For example, speech recognition or, music generation (audio clip on input), sentiment classification (movie reviews), DNA sequence analysis (DNA is represented via the four alphabets A, C, G, and T. So given a DNA sequence you can label which part of this DNA sequence corresponds to a protein), machine translation (output the translation from an input sequence in a some human language in a different language), video activity recognition (it is given a sequence of video frames for recognizing some activity), name entity recognition (given a sentence for identifying people in that sentence).
- **Supervised learning**. There are a lot of different types of sequence problems. In some, both the input X and the output Y are sequences, and in that case, sometimes X and Y can have different lengths, or sometimes they may have the same length. And in some problems only either X or only Y is a sequence.
- There can be various representation, according to a given problem. For example for word representation, it is needed to have a vocabulary (some list of words sorted alphabetically, usually 100k or even a few millions of words). Then with such dictionary, we can use **one-hot representation** (vector with all zeroes except 1 number ('1') with position representing a given word in a dictionary) - so if our dictionary has 10k words and word "and" is on 200. position in a dictionary, then resulting vector will be all zeroes except 200. position, there will be '1'. And this is called a "one-hot" encoding, because in the converted representation exactly one element of each column is "hot" (meaning set to 1). If a word is not in a dictionary, new token is created <unknown> for all words outside of dictionary. So to summarize, input is a sentence of various length and 1 item in this input is a one-hot vector.

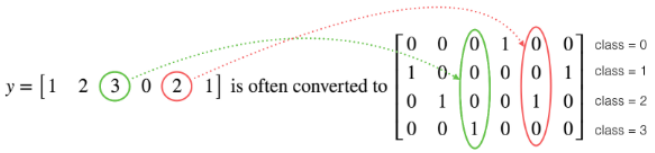


Figure 5.9: An example of one-hot encoding with 4 classes.

- Why not to use a standard networks?
  - Because inputs (and outputs) can have different length (different sentences have different number of words) and with padding to some MAX value still doesn't seems to be a good representation.
  - More serious problem is that such naive ANN doesn't share features learned across different positions of text.
- **Types**

- **Memory-less models for sequences**
  - **Autoregressive models**, predict the next term in a sequence from a fixed number of previous terms using "delay traps".
  - **Feed-forward neural networks**, generalize autoregressive models by using one or more layers of non-linear hidden units.
- **Beyond memory-less models**
  - **Linear dynamic systems,** generative models that have a real-valued hidden state that cannot be observed directly. They are stochastic (as well as HMMs, but not RNNs).
  - **Hidden Markov Models**, have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic. We cannot be sure which state produced a given output. So the state is "hidden".
  - **Recurrent neural networks**, powerful, they combine 2 things: a) distributed hidden state that allows them to store a lot of information about the past efficiently, b) non-linear dynamics that allows them to update their hidden state in complicated ways.

Recurrent Neural Networks

- Work well with sequential or time series data, as detailed with the beginning of this section. They are used to label, classify, or generate sequences.
- RNN is able to simulate deterministic finite automata (so they are able to classify finite strings).
- RNN is not a feed-forward network, because RNN contains loops. So, its graph has at least one cycle. The idea is that each unit u of recurrent layer l has a real-valued state $h_{l,u}$. This state can be seen as the memory of the unit. **Each unit** u in each layer l receives **two inputs**: a vector of states from the **previous layer** $l-1$, and the vector of states from **this layer** l, but **from the previous time step**.
- One challenge affecting RNNs is that early models turned out to be very difficult to train, harder even than deep feedforward networks. The reason is the unstable gradient problem. Recall that the usual manifestation of this problem is that the gradient gets smaller and smaller as it is propagated back through layers. This makes learning in early layers extremely slow. The problem actually gets worse in RNNs, since gradients aren't just propagated backward through layers, they're propagated backward through time. If the network runs for a long time that can make the gradient extremely unstable and hard to learn from. Fortunately, it's possible to incorporate an idea known as LSTM (idea from 1997) units into RNNs.

- RNN is just a layered net that keeps reusing the same weights, see figure below. So we can think of the recurrent net as a layered feedforward net with shared weights and then train the feed-forward net with weight constraints. So backpropagation through time (training RNN) is in time domain: **a) forward pass**, which builds up a stack of activities of all the units at each time step, **b) backward pass** peels activities off the stack to compute the error derivatives at each time step, and **c) after the backward pass** we add together the derivatives at all the different times for each weight.

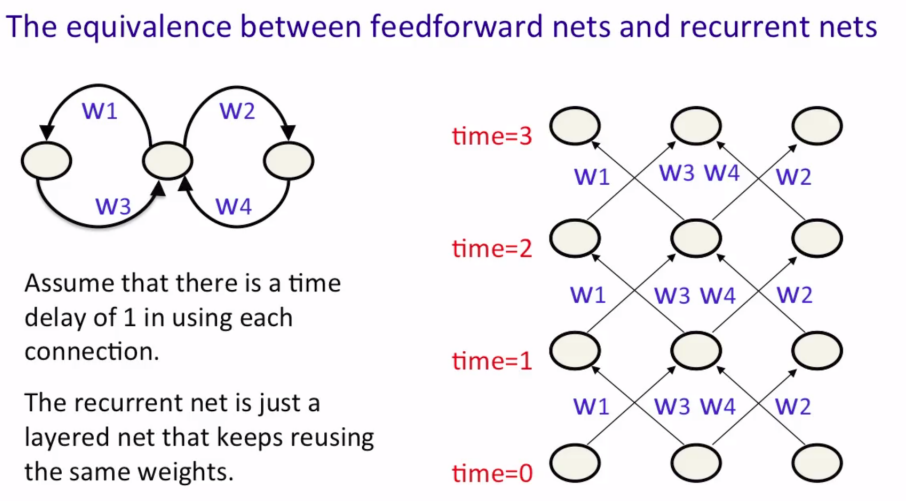


Figure 5.10: RNN and feedforward network equivalence.

- **Providing input to RNN:**
  - specify the initial states of **all the units** (so input layer has all inputs there),
  - specify the initial states of a **subset of the units** (one or just a few units have input),
  - specify the states of **the same subset of the units at every time step** - this is the most natural way to model sequence data.
- **Providing output target to RNN:**
  - specify desired final activities of **all the units**,
  - specify desired activities of **all units for the last few steps**,
  - specify desired activity of a **subset of the units**.

  An example: Suppose we're training an RNN on a sequence of numbers. After it has seen all numbers in the sequence, we want it to tell us the sum of the numbers of the sequence. To provide **input**, we should specify the state of one unit (say unit number 1), at every time step. We should specify a **target for one unit**

(say unit number 2), **only at the final time step**. There's 1 input value at every time step, namely the next number in the sequence. There 1 output value, and only at the last time step: that's where the model is expected to produce the sum of the numbers in the sequence.

- RNN are more biologically realistic. They have very complicated dynamics, and this can make them very difficult to train.
- RNN for modeling sequences - they are a very natural way to model sequence data. They are equivalent to very deep nets with 1 hidden layer per time slice. Except that they use the same weights at every time slice and they get input at every time slice.
- For example, they can generate text by **predicting the probability distribution for the next character** and then **sampling a character from this distribution.**
- To run **backpropagation on RNN**, we need to define a **loss function**:
  - **Element-wise loss** (this is for a single word): for a given word in the training sequence we determine classifier output vs ground truth, with for example logistic regression loss (=cross entropy loss).
  - **Sequence-wise loss** (for all the elements / words in the sequence): sum of element-wise loss functions.
- **Echo State Network (ESN)**
  - This is a RNN with sparsely connected (most of the weights are set to zero) hidden layer (typically 1% connectivity). The connectivity and weights of hidden neurons are fixed and randomly assigned. The weights of output neurons can be learned so that the network can (re)produce specific temporal patterns. The main interest of this network is that although its behavior is non-linear, the only weights that are modified during training are for the synapses that connect the hidden neurons to output neurons. Thus, the error function is quadratic with respect to the parameter vector and can be differentiated easily to a linear system.
  - These use a clever trick to make it much easier to learn a RNN. However, to get these networks to be good at complicated tasks, you need a very big hidden state - many more hidden units for a given task than a RNN that learns weights.
  - Hidden-to-output connections can be learned easily, as opposed to the hidden-to-hidden connections which aren't learned at all in an Echo State Network. It is very important to initialize weights sensibly, mostly hidden-to-hidden weights.
  - However, they cannot do well on high-dimensional data.

  - ESNs don't do backpropagation through time - the hidden-to-hidden connections are not learned. Also, without learning, there is no overfitting possible.

- **Hopfield networks** - similar to RNN (but without hidden units), there exist symmetrically connected networks. Connections between units are symmetrical (they have the same weight in both directions). These networks are much easier to analyze. They are also more restricted in what they can do, because they obey an energy function. For example, they cannot model cycles.
- **Boltzmann Machines = stochastic Hopfield networks.** They are good at modeling binary data.

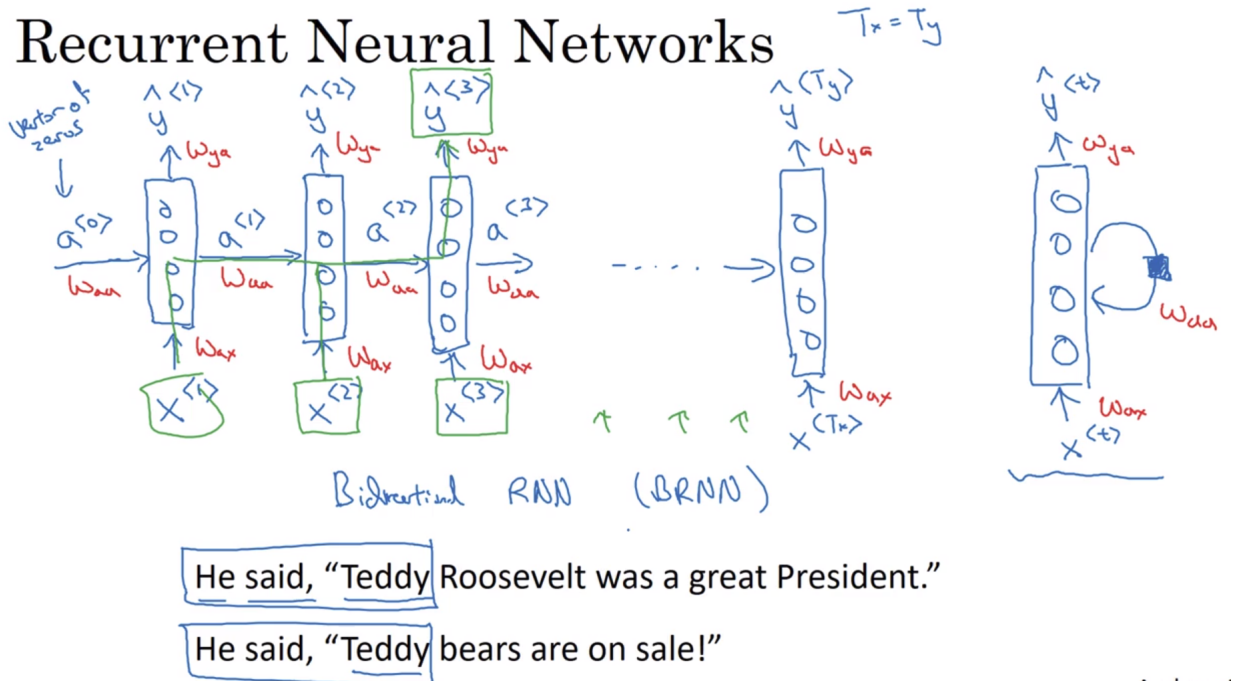


Figure 5.11: This figure describes a simple (unidirectional) RNN architecture. You are reading an input sentence from left ($x^{<1>}$) to right ($x^{<T_x>}$), the first word $x^{<1>}$ will be feeded it into a neural network layer. So RNN also scans through the data from left to right as well. When it then goes on to read the second word $x^{<2>}$ in the sentence, instead of just predicting $y^{<2>}$ using only $x^{<2>}$ it also gets to input some information from the fist time step. So in particular, the activation value from the first step is passed on to time step two. And for completness, activation value on the very beginning, $a^{<0>}$,is usually a vector of zeroes (but some researchers initializes this vector randomly). Sometimes, we can encounter a diagram like one on the right side. Sometimes people draw a loop like that, that the layer feeds back to the cell. Sometimes, they draw a shaded box to denote a time delay of one step. However it is much harder to interpret. RNN shares the parameters in each step (red color, they are the same). One disadvantage of this RNN is, that it only uses information which is earlier in the sequence to make a prediction. For that 2 examples, it would be good to know that Teddy is a President, or that it is something about bears on sale, but this model cannot access this information. Looking on the first 3 words, we cannot tell a difference. This problem is addressed to Bidirectional RNN (BRNN).

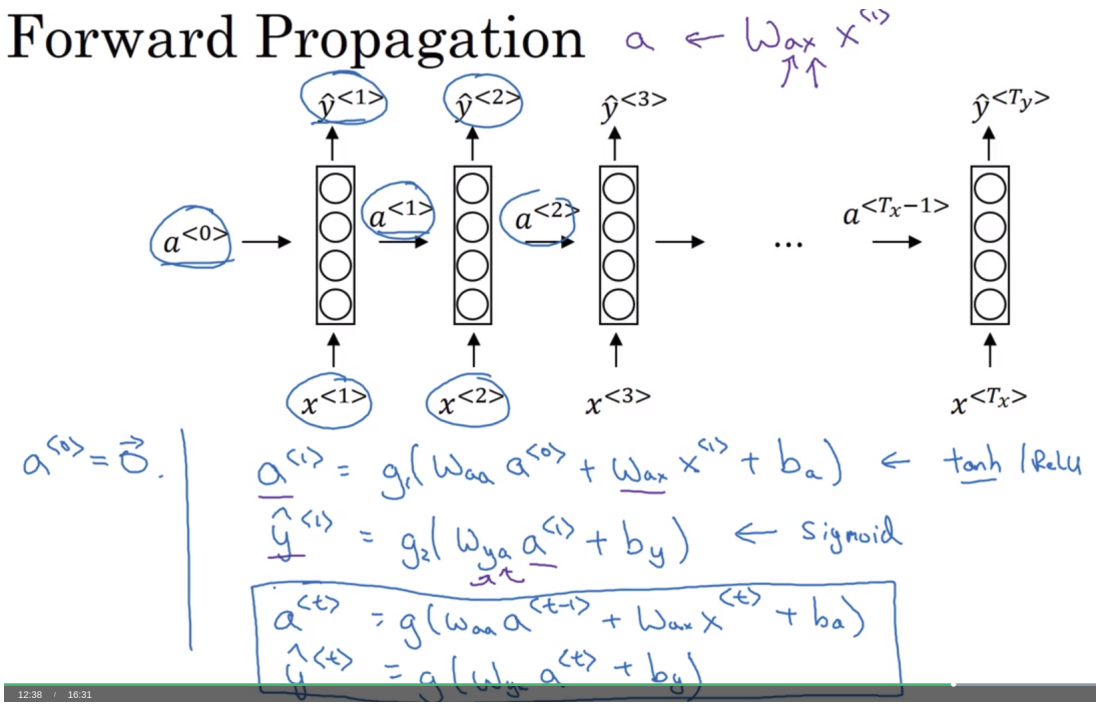


Figure 5.12: Forward propagation of RNNs. $g(..)$ is an activation function. w_{ax} means that this will be multiplied by x quantity and it will compute a quantity. Activation function for output $\hat{y}$is usually sigmoid or softmax (depends on a problem).

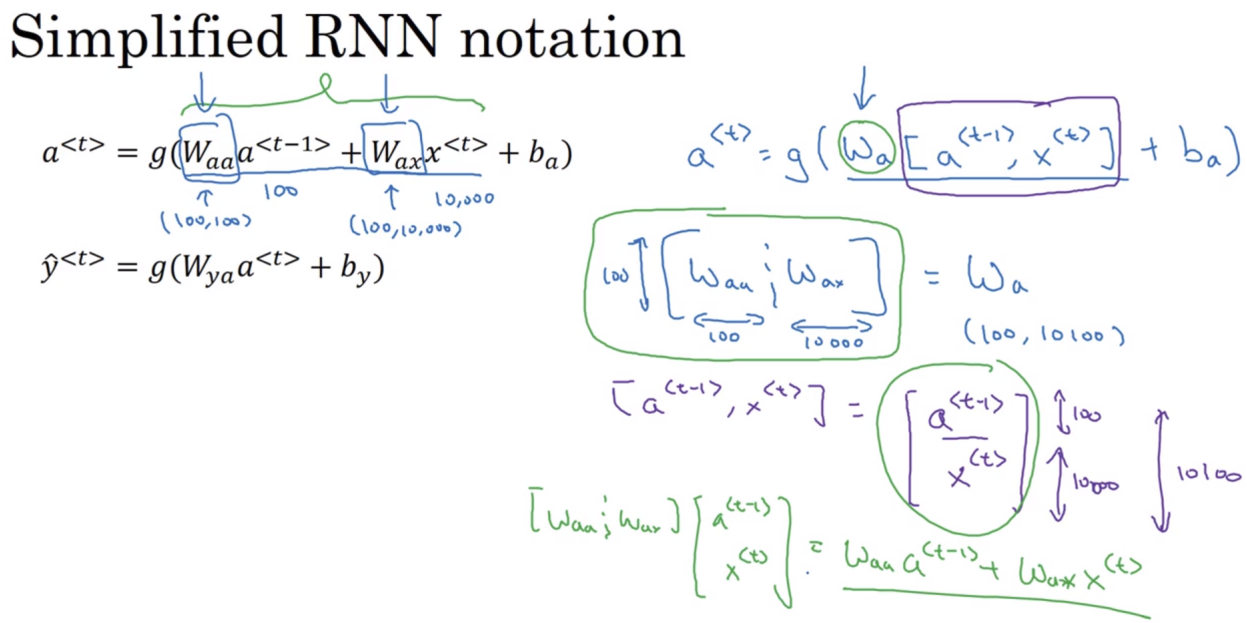


Figure 5.13: A simplified computationo of forward pass in RNN. rather than carrying around two parameter matrices, W_{aa} and W_{ax}, we can compress them into just one parameter matrix W_a.

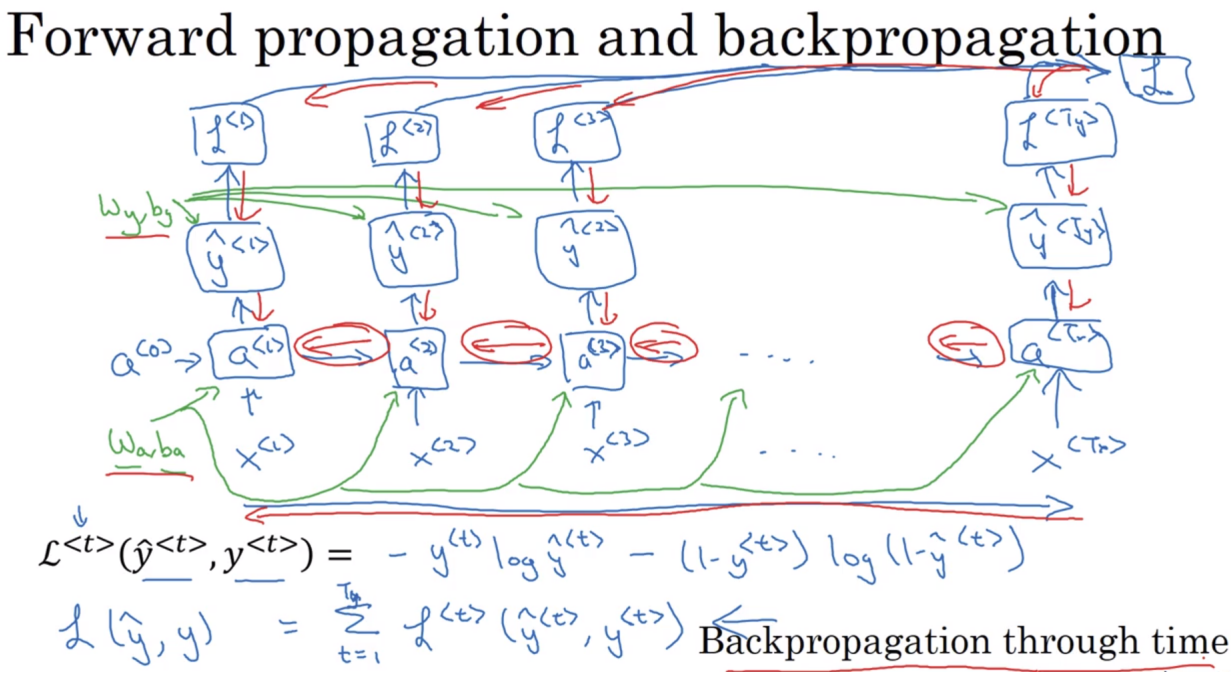


Figure 5.14: Forward and backward pass of a simple (unidirectional) RNN where the length of input and output is the same.

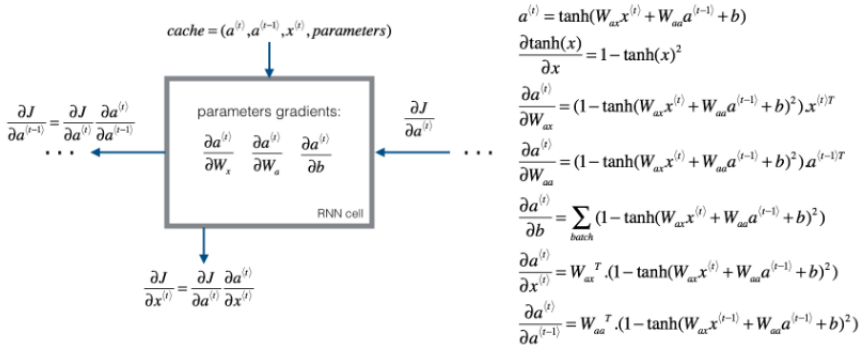


Figure 5.15: Backward pass for the basic RNN-cell. Just like in a fully-connected neural network, the derivative of the cost function J backpropagates through the RNN by following the chain-rule from calculas. The chain-rule is also used to calculate $(\frac{\partial J}{\partial W_{ax}}, \frac{\partial J}{\partial W_{aa}}, \frac{\partial J}{\partial b})$ to update the parameters (W_{ax}, W_{aa}, b_a).

- Types of RNN architectures:
  - **Many-to-many**, as been in all the previous figures, where input and output had the same length. This architecture is called many-to-many, because

there are many inputs and many outputs. There can be many-to-many with different lengths of input and output. This architecture is widely used for machine translation. In this second case, RNN will first read the input sequence entirely, and only then will generate the output (this is basically an encoder/decoder).

- **Many-to-one**, where there is just 1 output $\hat{y}$. This architecture can be used for example for sentiment classification, because there is no need to have output on each word - only 1 output on the whole sentence.
- **One-to-one** architecture is less interesting. It is basically standard NN.
- **One-to-many** architecture can be used for a music generation. There is just 1 input, integer, for example a genre of music you want or the first note of the music you want, and the architecture will output notes. The input would be on the very right side of RNN diagram only, so each step will be computed by using the previous activation (no input except the first one, but sometimes there is one at each step, and this is synthesized output from the previous step as can be seen in the next figure below, so $x^{<t>} = y^{<t-1>}$).

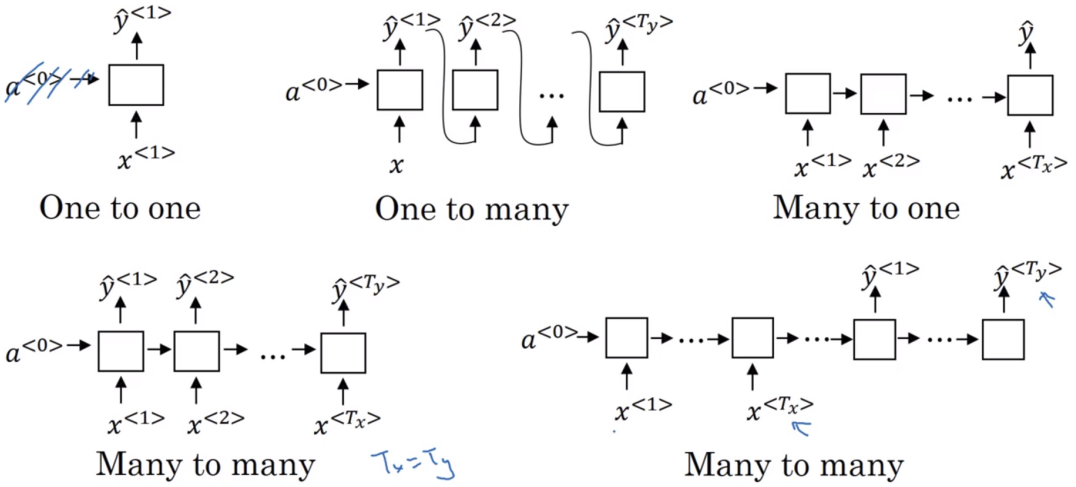


Figure 5.16: Different types of RNN architectures (except Attention RNN architecture, which is a bit different).

- **Attention model**
  - Another very significant type of RNN architecture (Bahdanau et al, from 2014). It is basically a modification of many-to-many architecture, where there is encoder and decoder as well.
  - It is very useful when there is a very long sequence on the input. Bleu score is quite good for shorter sequences for traditional man-to-many

RNNs, but as a sequence goes bigger and bigger (30 or more characters), Bleu score decreases.

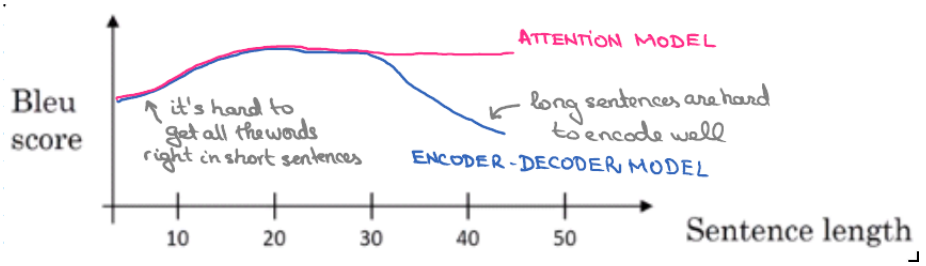


Figure 5.17: In tradicional encoder-decored RNN, the performance decays for longer sentences.

* This model has been applied in machine translation as well as to image captioning.
* Attention model uses **Bidirectional RNN** (classical RNN / GRU / LSTM - the last is most often used), with so called **attention weights** which are output of such BRNN and input to another RNN, that finally generates output (for example translation). Attention weights are basically information about how much attention should be given to a particular word from the input (or only a part of an input sentence while it's generating a translation, just like humans might - because humans do not translate in a way that they will remember the whole sentence and then we will start translating; instead, it will break the sentence up into subsentences, and translate those party by part and build up the translated sentence). This allows us on every timestep to look only maybe within a local window of the input sentence to pay attention to when generating a specific word (for example, translated to English). This method does not push the ability of the network to encode (memorize) long sequences beyond it's optimum.
* **Visualization of attention weights** can be useful. You can find out that attention weights to the corresponding input and output words will tend to be high. Thus, suggesting that generating a specific word to output is usually paying attention to the correct words from the input. All this thanks to learning using backpropagation with an attention model.
* The network learns where to "pay attention" by learning the values $e^{<t,t'>}$, which are computed using a small neural network: we can't replace $s^{<t-1>}$ with $s^{<t>}$ as an input to this neural network. This is because $s^{<t>}$ depends on $\alpha^{<t,t'>}$ which in turn depends on $e^{<t,t'>}$; so at the time we need to evaluate this network, we haven't computed $s^{<t>}$ yet.

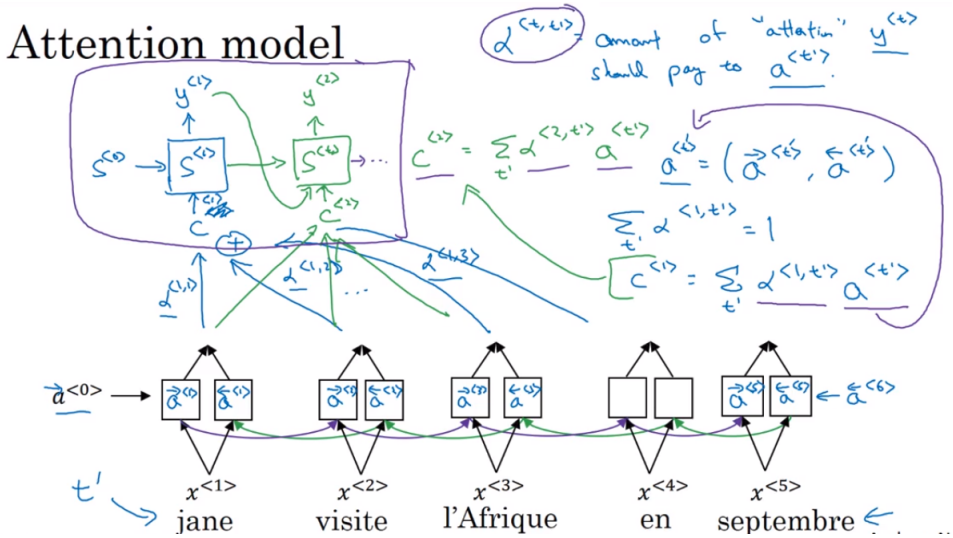


Figure 5.18: An intuition behind Attention model on French to English machine translation problem. Here, bidirectional RNN (or GRU or LSTM - more common) is used. In each time step i, there are features computed from the forward and from the backward occurrence ($\overrightarrow{a}^{<i>}$and $\overleftarrow{a}^{<i>}$). These 2 will be represented as a tuple $a^{<t>} = (\overrightarrow{a}^{<i>}, \overleftarrow{a}^{<i>})$. Or $a^{<t'>}$for words from Frech sentence. Then, on top of this bidirectional RNN, there is a one-directional RNN with state s to generate the translation. The first step will generate $y^{<1>}$and this will have as input some context $c^{<1>}$and this will depend on attention parameters $\alpha^{<1,1>}$, $\alpha^{<1,2>}$,... and these parameters tell us how much attention = how much context would depend on the features we are getting (the activations) from the different time steps. Context is then defined as a sum on the features from the different time steps weighted by attention weights (these weights must satisfy that they are non-negative, and sum to 1) - $\sum_{t'} \alpha^{<1,t'>} = 1$ and for context, $c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t'>}$.

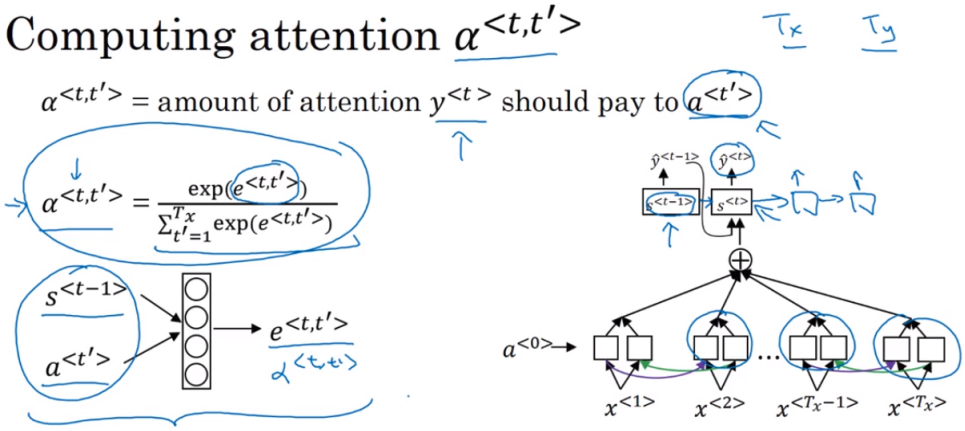


Figure 5.19: Computation of attention weights. After computation of $e^{<t,t'>}$(by a small ANN on the left, usually with 1 hidden layer), we are computing basically a softmax. This algorithm runs in quadratic cost which is a disadvantage. However, if an input sequence is not so extremely long, it is acceptable. There are research works on reducing this computational problem. BTW, similar architecture can be applied on other problems as well, for instance image captioning (look at the picture, and pay attention only to parts of the picture at a time while the algorithm is writing a caption for the picture).

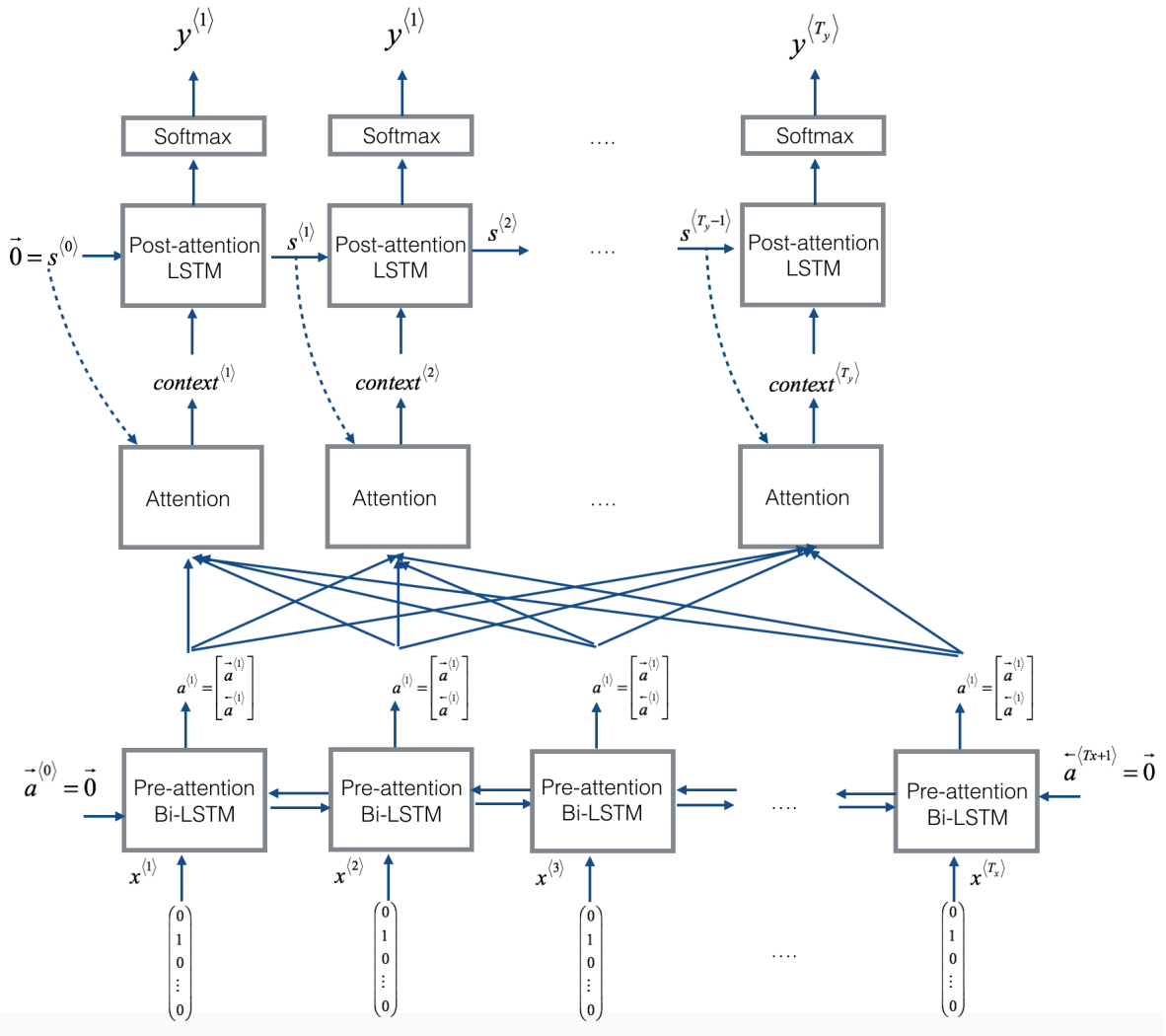


Figure 5.20: Attention model schema. There are 2 separate LSTMs in this model. The one at the bottom of the picture is a bidirectional LSTM and comes before the attention mechanism, we will call it pre-attention Bi-LSTM. The LSTM at the top of the diagram comes after the attention mechanism, so we will call it the post-attention LSTM. The pre-attention Bi-LSTM goes through T_x time steps; the post-attention LSTM goes through T_y time steps. The post-attention LSTM passes $s^{\langle t \rangle}, c^{\langle t \rangle}$ from one time step to the next. In 2 figures above, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{\langle t \rangle}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{\langle t \rangle}$ and the hidden cell state $c^{\langle t \rangle}$. We use $a^{\langle t \rangle} = [\overrightarrow{a}^{\langle t \rangle}; \overleftarrow{a}^{\langle t \rangle}]$ to represent the concatenation of the activations of both the forward-direction and backward-directions of the pre-attention Bi-LSTM.

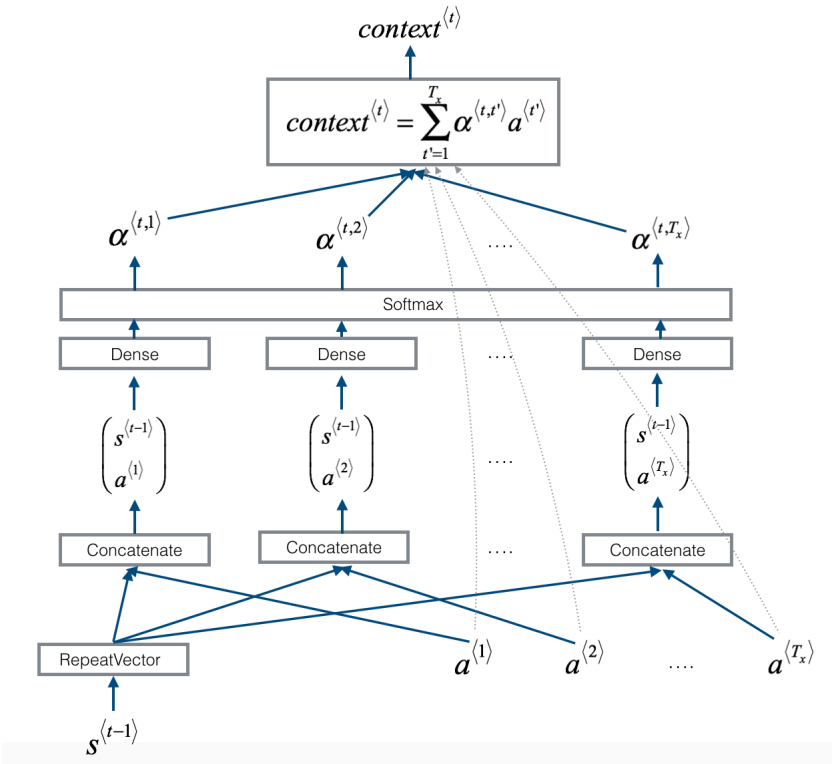


Figure 5.21: This diagram shows what one "attention" step does to calculate the attention variables $\alpha^{\langle t,t' \rangle}$, which are used to compute the context variable $context^{\langle t \rangle}$ for each timestep in the output ($t = 1, \ldots, T_y$). The model uses a **RepeatVector** node to copy $s^{\langle t-1 \rangle}$'s value T_x times, and then **Concatenation** to concatenate $s^{\langle t-1 \rangle}$ and $a^{\langle t \rangle}$ to compute $e^{\langle t,t' \rangle}$, which is then passed through a softmax to compute $\alpha^{\langle t,t' \rangle}$.

- **Vanishing gradients** is one of the problems with a basic RNN algorithm (actually, with deep neural nets in general!). Neurons in second hidden layer learn faster than neurons in the first layer, and so on - neurons in deeper layers learn much faster (the first vs the latest hidden layer can learn 100 times slower). We have here an important observation: in at least some deep neural networks, the gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. Sometimes, there is exploding gradient problem, where gradient in deep nets gets much larger in earlier layers. So gradient in deep nets is very unstable, and it tends to explode or vanish in earlier layers.

  Exploding gradient problem - for example, let's choose all the weights to the network to be large, say $w_1 = w_2 = w_3 = w_4 = 100$. Second, choose biases so that terms $\sigma'(z_j)$ are not too small. That is pretty easy to do, just ensure that each neuron has $z_j = 0$ (so that $\sigma'(z_j) = 0.25$ (we are using sigmoid neurons

again - however, in sigmoid nets, there is mostly vanishing gradients problem). So, for instance, $z_1 = w_1 a_0 + b_1 = 0$. We can achieve this setting by $b_1 = -100 a_0$. Using the same idea for other biases, when we do this, we see that all terms $w_j \sigma^{'}(z_j) = 100x0.25 = 25$. With these choices we got exploding gradient.

Unstable gradient problem - fundamental problem here is not much the vanishing or exploding gradient problem. It's that the gradient in early layers is the product of terms from all the later layers. When there are many layers, that is a really unstable situation. The only way all layers can learn at close to the same speed is if all those products of terms come close to balancing out. By default, this is unlikely to happen simply by chance.

Human language can have very long-term dependencies, where it worked at this much earlier can affect what needs to come much later in the sentence. But it turns out the basics RNN it's not very good at capturing very long-term dependencies - an example of 2 sentences:

- The cat ..... was ...
- The cats ..... were ...

  It might be difficult to get a neural network to realize that it needs to memorize the just see a singular noun or a plural noun, so that later on in the sequence that can generate either was or were, depending on whether it was singular or plural. And notice that in English, this stuff in the middle could be arbitrarily long. **RNN forgets.** In very deep RNN, the gradient (after forward propagation) would have a very hard time to propagate back to effect computations which were done much earlier.

  The basic RNN model has many local influences, meaning that the output $\hat{y}^{<3>}$ is mainly influenced by values close to $\hat{y}^{<3>}$. In other words, an element is influenced much more by close neighbors in the sequence than it is influenced by distant neighbors. The error cannot backpropagate easily.

  Vanishing gradients can be mitigated using Hessian Free Optimization - it uses optimizer that can detect directions with a tiny gradient but even smaller curvature. Or, proper initialization of initial weights. Or this with momentum. Or using LSTM for example...

- **Language modeling**
  - It is one of the most basic and important tasks in natural language processing. Building a language model can be done with RNN as can be seen in the next figure. Then it can be used for generating a text which is similar to training corpus (see corpus in the 1.4) and this is called **sequence generation.** Sequence model models a chance of any particular sequence of words as follows, and so what we like to do is to sample from this distribution to generate novel sequences of words. Sampling from trained RNN is different than training RNN on some corpus.

- There can be **word-level language model** or **character-level language model**, based on your vocabulary. Character-level language model has an advantage that there can't be an unknown token, because we are working with the whole alphabet or all possible symbols (in ideal case). But the main disadvantage is that you end up with much longer sequences. Character-level models are not as good as word language models at capturing long range dependencies between how the earlier parts of the sentence also affect the later part of the sentence. And character-level models are more computationally expensive for training. But in some special cases (for example when you need to deal with unknown words a lot) with this increase of computational power these days, some people are using also character-level language models.
- To summarize, in a language model we try to predict the next step based on the knowledge of all prior steps. At time step t, RNN is estimating $P(y^{<t>} \mid y^{<1>}, y^{<2>}, \ldots, y^{<t-1>})$.

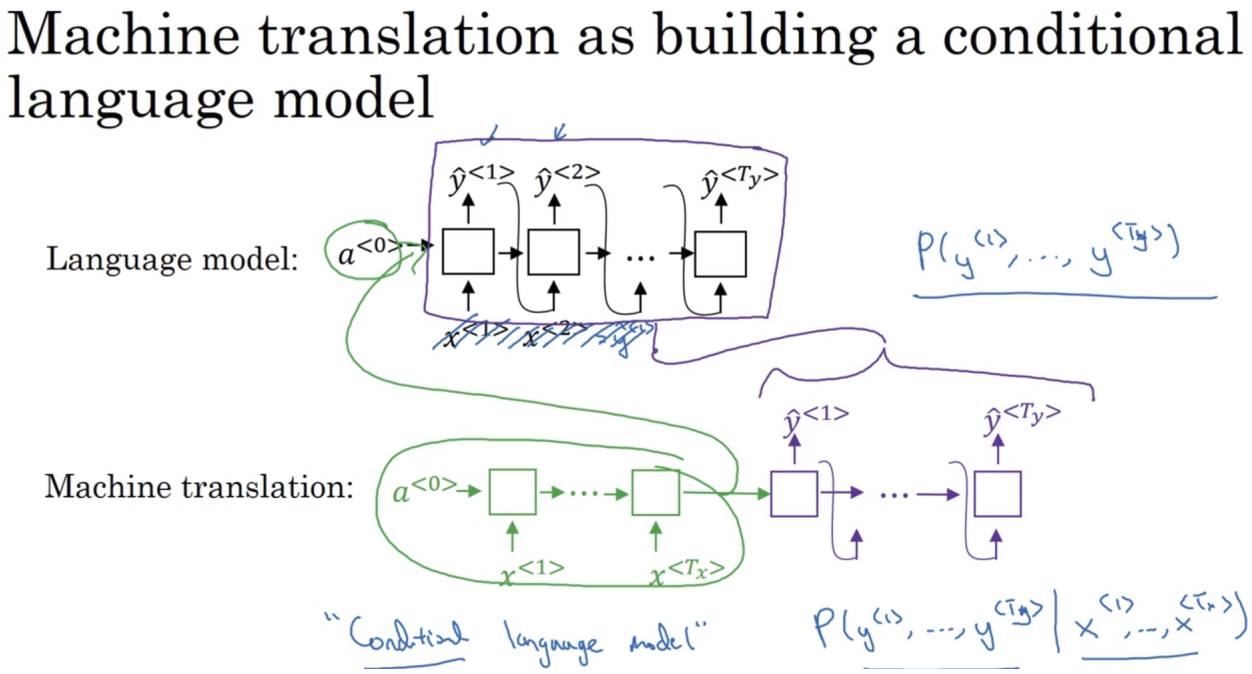


Figure 5.22: Machine translation building a "conditional" language model. It models, for example, probabilities of English words (translation) which depends (conditions) on some input - French words sentence. Green part is basically encoder network, that figured out some representation for the input sequence. One major difference between this and the earlier language modeling problems is rather than wanting to generate a sentence at random, it is needed to try to find the most likely English sentence, (translation). But the set of all English sentences of a certain length is too large to exhaustively enumerate. Therefore a search algorithm is needed - more precise Beam Search, see the next figure.

– For picking the most likely sentence, there is **Beam Search** algorithm. Greed search is very exhaustive and pick only 1 word and move on, Beam search can consider multiple alternatives. There is a parameter B (beam width, for example 3), that determines a number of alternatives. If $B = 1$, then this is a greedy search algorithm (but this usually finds much worse output sentence). Bigger B means usually better output (you consider a lot of possibilities), but it is more computationally expensive. In comparison to BFS or DFS (Breadth/Depth First Search) algorithms, in Beam search it is not guaranteed that the algorithm will find exact maximum for $arg\,max_y P(y|x)$. BFS and DFS are exact search algorithms. In production, beam parameter B is usually 10 or 100, and in research it can be even 1000 or 3000. This is application and domain dependent. Beam search is an approximate search algorithm, which means that it can sometimes output the wrong result. Usually, during / after

error analysis, beam width parameter should be increased if the fraction of beam search errors is larger than RNN errors.

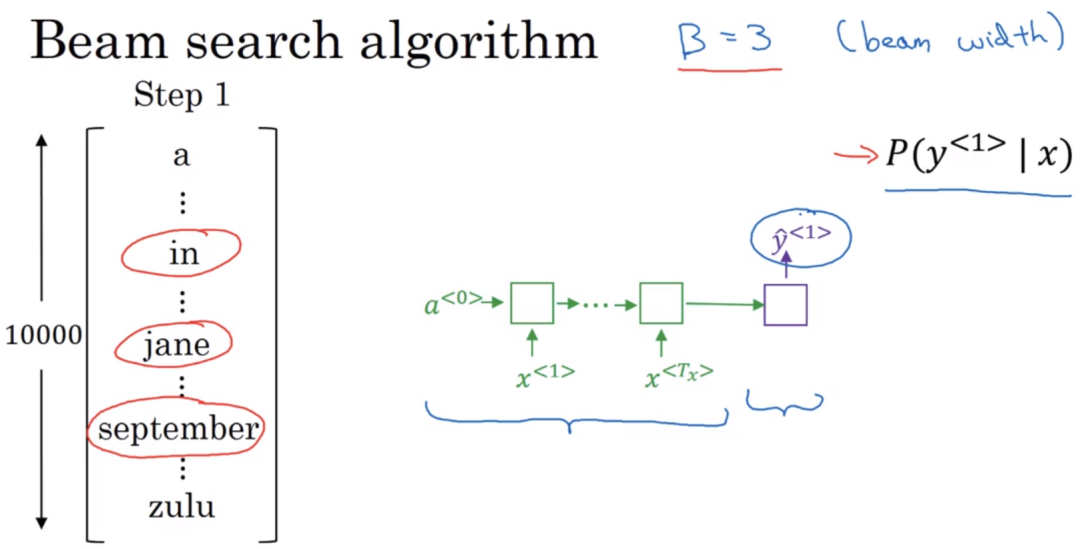


Figure 5.23: The first step of Beam search algorithm for picking the best and the most likely English translation. On the left, there is a dictionary of 10k words (all lowercase, but it is just a detail). We can see a NN fragment which is used by this step of Beam search algorithm (encoding is green, purple is encoding). This NN is trying to evaluate the probability of the first word (output) given the input sequence (in French for instance). Greedy search would pick the most probable one and move on, but Beam search with $B = 3$ will pick the best 3 candidates (in this example words 'in', 'jane', 'september'). So, the input sequence is run through encoding network, and then decoding network has softmax output over all 10k possibilities, then from these the algorithm will keep in memory only the top 3. The algorithm continues on the next figure.

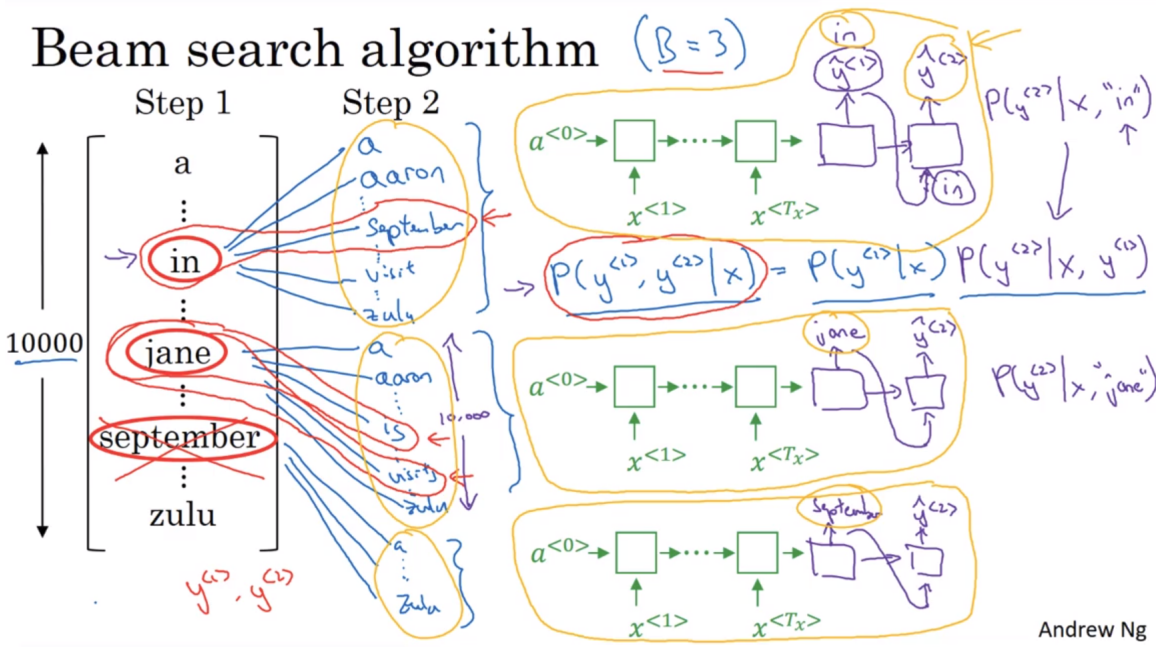


Figure 5.24: The second step of Beam search algorithm. The algorithm will for each 3 words consider what should be the second word (for each of these 3 words, 10k possibilities, so together 30k possibilities and again only the top 3 are saved). There is again another fragment of NN, with encoder and decoder parts. So, the second word on output is computed not just from the whole French input x, but also from the previously outputted word (this is what NN fragment from the figure is doing). But we are not searching only for just the second word, but for the pair <first word, second word>, that is most probable (actually 3 such pairs!), $P(y^{<1>}, y^{<2>}|x)$. So the algorithm will remember 3 such pairs, not NN - we are using a fragments = multiple instances. In this example, we found that 'september' from the previous step is not very probable (worse than top 3) with any pair, so this 'path' will be excluded. Because $B = 3$, in each step we need to instantiate 3 copies of NN (a different choices for the first word). But these 3 copies can be very efficiently used for evaluate all 30k choices for the second word. The algorithm continues on the next figure.

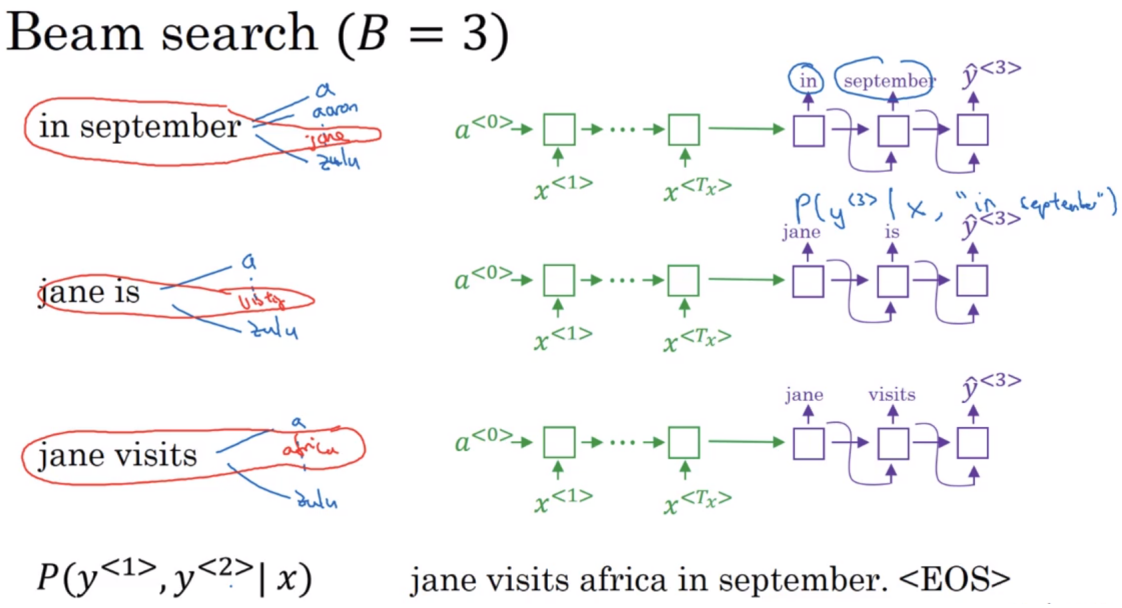


Figure 5.25: One more step of Beam search algorithm. After few more steps, hopefully the algorithm will outputs the correct translation (with <EOS> at the end).

Length normalization is a small change to the Beam search algorithm that can help to get much better results. By multiplying a lot of small numbers (close to 0), this will result in very very small number. This can be seen in $P(y^{<1>}, ..., y^{<T_y>}|x) = P(y^{<1>}|x) * P(y^{<2>}|x, y^{<1>}) * ... * P(y^{<T_y>}|x, y^{<1>}, ..., y^{<T_{y-1}>})$. Instead of doing $arg\,max_y \prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$, it is better to use logarithm (equivalent, should give the same result, but more numerically stable for computers = they cannot work precisely with floating point numbers): $arg\,max_y \boldsymbol{log} [\prod_{t=1}^{T_y} loP(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})]$ that results in $arg\,max_y \sum_{t=1}^{T_y} \boldsymbol{log} P(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})$.[3] If we are using a lot of words, it means a lot of multiplications with small numbers, so this will end up to have lower probability than outputting a small number of words. So this algorithm will tend to output a short sequences. This also holds true for logarithm modification, because they are negative numbers and more words we add, more negative (smaller) the result is. So the last modification is to normalize the result by a number of outputted words (translation) - multiplication by $\frac{1}{T_y}$. Or, more soft version normalizing with $\frac{1}{T_y^\alpha}$, where hyperparameter α can be 0.7 (if $\alpha = 1$, we are normalizing with the number of words, if $\alpha = 0$, then we are not normalizing at all, and in practice we want something in between). This normalization significantly reduces the penalty of outputting long sentences and because of hyperparameter α, this is a kind

[3] Because log(a.b)=log(a)+log(b) and log(z) is strictly monotonically increasing function.

of soft approach.

- **Word embeddings**
  - One of key ideas in NLP, it is a way of representing words, so that the algorithm understands analogy between men and women, or king and queen. With these ideas of word embeddings, it is possible to build a neural network and be successful with relatively small training set on some NLP task.
  - **Word representation can be** (for example, but there are many many algorithms and modifications):
    - **A vocabulary** (dictionary) can be used with **one-hot vector**. But there is no context between words with this representation - "apple" and "orange" are totally different one-hot vectors, and their inner product results in vector of zeros. Also, this representation treats each word as an independent entity. This does not allow algorithms to establish relationships between words, and it is not easy for algorithms to learn to generalize across words. Any inner product (dot product) between any two words is 0. Also the distance between any two words is the same for all pairs in the vocabulary.
    - **Featurized representation** - word embedding, see the next figures. The dimension of word vectors is usually smaller than the size of the vocabulary. Most common sizes for word vectors ranges between 50 and 400. We cannot guarantee that the individual components of the embeddings are interpretable.

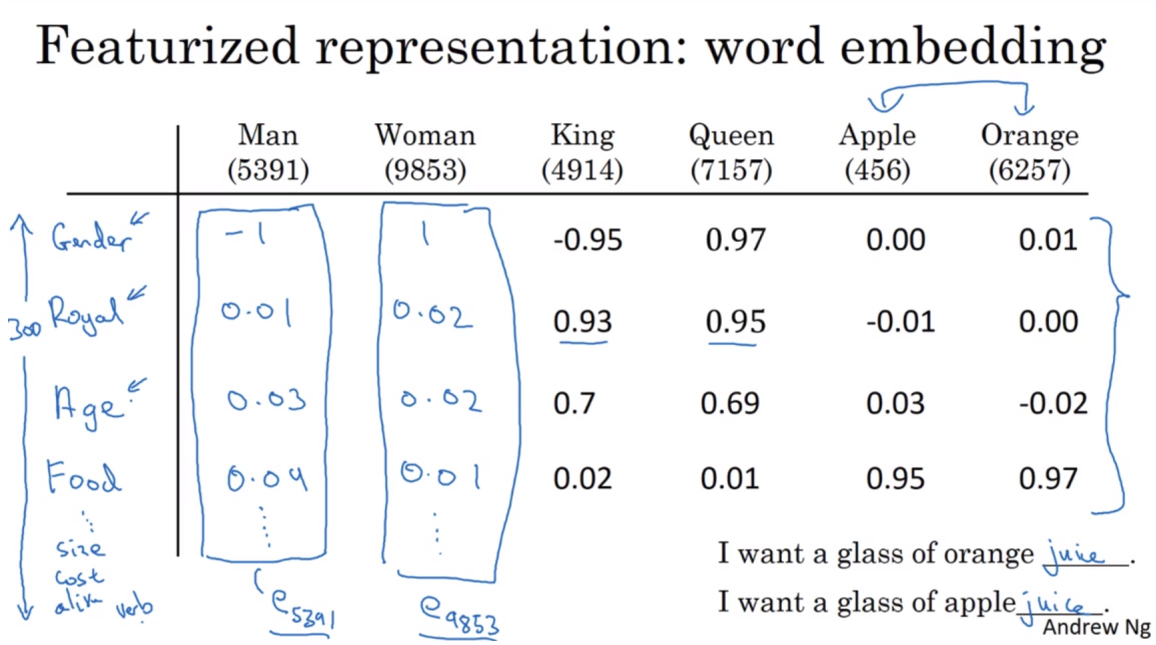


Figure 5.26: Word embeddings via featurized representation example.

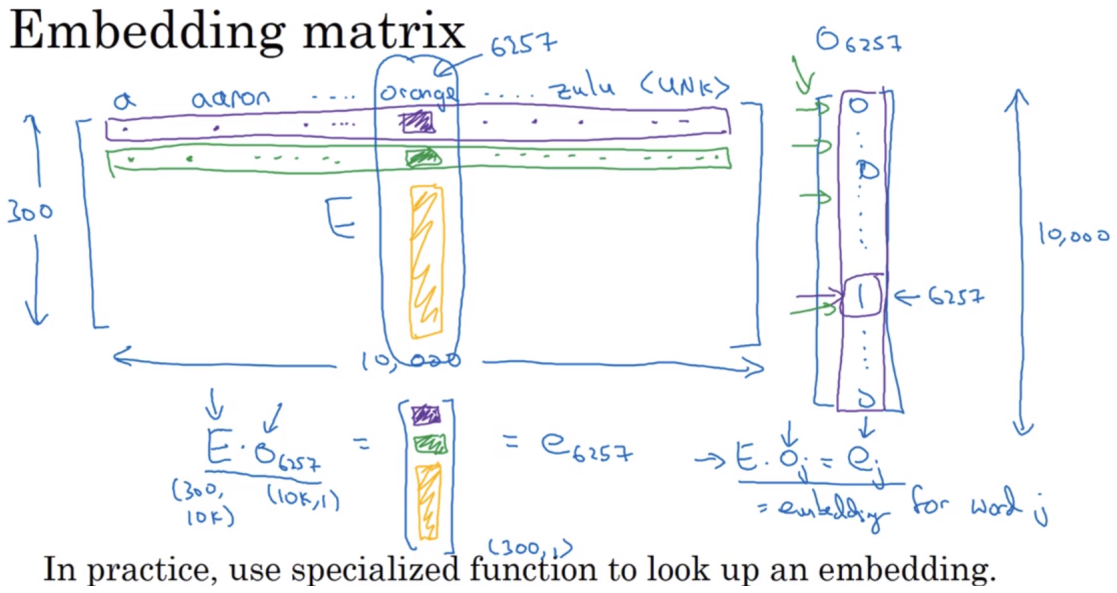


Figure 5.27: Embedding matrix multiplied on some specific one-hot vector, for example for word 'orange' results in embedding vector containing features for word 'orange' which is $(300, 1)$ dimensional. E is initialized randomly and using gradient descent, and learn all parameters using this $(300, 10\,000)$ matrix. However, in practice, it is not efficient to multiply matrix E by one-hot vector that is mostly a bunch of zeros. In practise, we use specialized function that lookups for a specific column in matrix E rather than do the whole multiplication of one-hot with the whole matrix E.

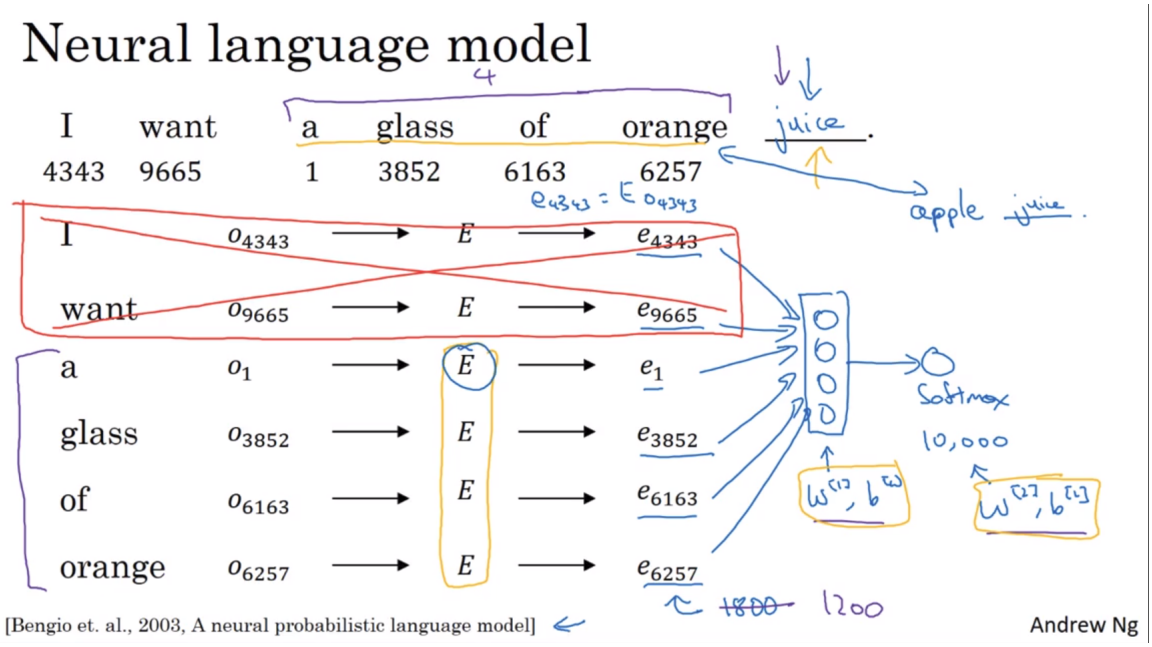


Figure 5.28: Learning word embeddings (from 2003). By the time, newer and simpler algorithms were developed with similar performance than very complex ones. Word embeddings are computed from E multiplied by one-hot vector for a given word. If we have 10,000 words in our dictionary, and 6 words from our sentence, then we have 6 one-hot vectors as well as 6 word embedding vectors. We put these 6 word embeddings vectors to NN, then to softmax (which has 10,000 outputs), **for prediction of the next word in a sentence**. What's actually more commonly done is to have a fixed historical window. So for example, you might decide that you always want to predict the next word given say the previous 4 words, where 4 here is a hyperparameter of the algorithm. Then it is possible to deal with arbitrary long sequences, because we are always working with the fixed input sizes. However, more simpler approach is to use a context of just **nearby 1 word** (instead of 4), this is called **Skip-gram model**. The idea is that we need just 1 word to determine the context (very simple context though). **Skip grams are used for learning embeddings (also the last and the next few words, or the last 1 word), and for building a language model, it is better to use the last few words.**

- **word2vec** is one of the most popular algorithm to build word embeddings (Mikolov et al, from 2013). Previously (the last figure) there was so called skip-gram model. It is an algorithm for building word embeddings, to be more particular skip-gram is only one version of word2vec, which works well in practice. We come up to with a few pairs of **context -> target** to create our **supervised learning**

problem. From a given sentence, we randomly pick a word (put it to context), and the randomly pick a word within some window - which can be before or after the picked context word (and put it to target). Skip-gram model is taking as input one word (like "orange") and then trying to predict some words skipping a few words from the left or the right side. To predict what comes little bit before or little bit after the context words. Now, it turns out there are a couple of problems with using this algorithm. And the **primary problem is computational speed**. In particular, for the softmax model, every time you want to evaluate this probability, you need to carry out a sum over all 10k words in your vocabulary. And maybe 10k isn't too bad, but if you're using a vocabulary of size 100k or a 1M, it gets really slow to sum up over this denominator every single time. There are few solutions to this, for example using a hierarchical softmax classifier.[4] In the original word2vec paper the authors actually had 2 versions of this Word2Vec model. The skip-gram was one, and the other one is called the **CBow** (**the continuous backwards model**), which takes the surrounding contexts from middle word, and uses the surrounding words to try to predict the middle word. In the word2vec algorithm, you estimate $P(t \mid c)$, where t is the target word and c is a context word. t and c are chosen from the training set to be nearby words.

[4]The algorithm builds a binary tree for a vocabulary, and eventually you would go down in tree to classify exactly what word it is, so there is no need to SUM over all 10k words - vocab size, in order to make a single classification; this scales as instead of linear. Each node has a softmax classifier. In practice, the hierarchical softmax classifier doesn't use a perfectly balanced tree or a perfectly symmetric tree, with equal number of words on the left and right sides of each branch. In practise, the more common words are at the top of the tree, while less common words are deeper.

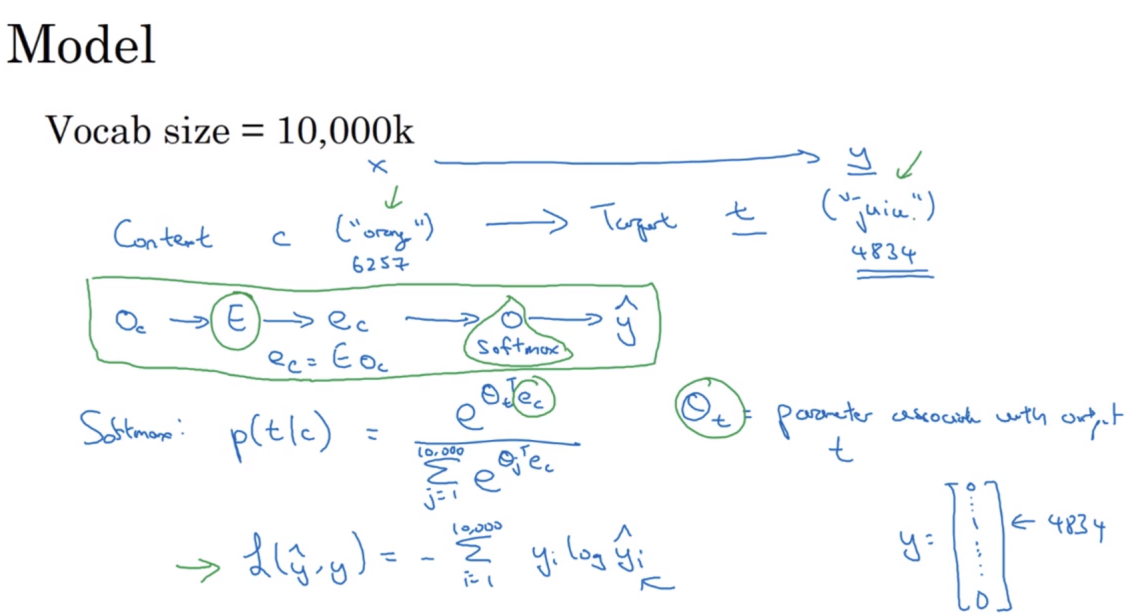


Figure 5.29: Skip-gram model for learning word embeddings. The green rectangle is basically a simple neural network.

Let's have an example. Let our dictionary contain 10k words. In word embedding learning, our goal is to build a model which we can use to convert a one-hot encoding of a word into a word embedding. Consider a sentence: "I almost finished reading the book on machine learning." Now consider the same sentence from which we have removed one word, say "book". Our sentence becomes: "I almost finished reading the . on machine learning." Now let's kept only the three words before and after this skipped word: "finished reading the . on machine learning" Looking at this 7-word window around the . if I ask you to guess what this skipped word stands for, you would probably say: "book", "article", or "paper". That is how the context words let you predict the word they surround. It's also how the machine can learn that words "book," "paper," and "article" have a similar meaning: because they share similar contexts in multiple texts. It turns out that it works the other way around too: a word can predict the context that surrounds it. The piece "finished reading the · on machine learning" is called a skip-gram with window size 7 $(3+1+3)$. By using the documents available on the web, we can easily create hundreds of millions of skip-grams. Let's denote a skip-gram with windows size 5 (see its schema on figure below) like this: $[x_{-2}, x_{-1}, x, x_{+1}, x_{+2}]$, where x is a word corresponding to skipped word (.), and for example x_{+1} is "on" and x_{-2} is "reading". You can see now why the learning of this kind is called self-supervised: the labeled examples get extracted from the unlabeled data such as text.

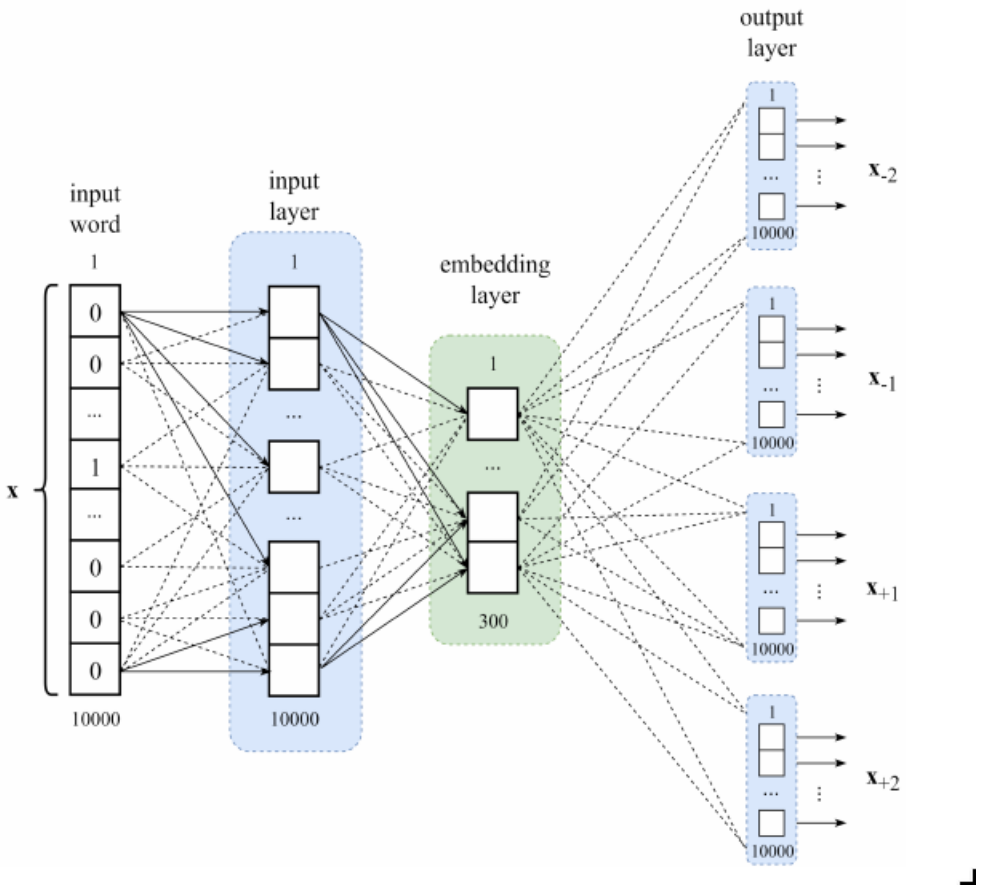


Figure 5.30: Schema of skip-gram model with window size 5 and embedding layer of 300 units using dataset that consists of 10k samples. It is a fully-connected network, like multi-layer perceptron. The input word is the one denoted as . in the skip-gram example above. The NN has to learn to predict the context words of the skip-gram given the central word. The activation function used in the output layer is softmax, and the cost function is negative log-likelihood. Because of the large number of parameters in the word2vec models, two techniques are used to make the computation more efficient: **hierarchical softmax** (this variant uses a binary tree) and **negative sampling** (the idea is only to update a random sample of all outputs per iteration of gradient descent).

* **Negative sampling** (Mikolov et al, from 2013). A new learning problem must be defined - given a pair of 2 words, we are going to predict if this is a context-target pair. This algorithm will generate for a given word multiple rows - one word from the sentence (within some window, for instance +-10 words) with target=1, and the other word randomly

chosen from a dictionary; the algorithm will put on this row target=0. To summarize, the way we generated this dataset is that the algorithm will pick a context word and then a target word from a given sentence (this is the first row, it is positive example). And then, for k times (from original paper, authors recommended for smaller dataset to be between 5 and 20, and for larger dataset they recommended to be between 2 and 5) algorithm will take the same context word and pick random words from a dictionary and label all those as zero. Those will be negative examples. It's okay if just by chance, one of those words we picked at random from the dictionary happens to appear in the window. It is called negative sampling because for each positive example we choose k negative examples. Instead of having 10k classes and softmax layer, we use a layer with 10k binary classification nodes (easier training), and in each iteration we are only going to train $k + 1$ of these nodes. So we will model this as a logistic regression: $P(y = 1|c, t) = \sigma(\theta_t^T e_c)$, where c is context word, t is target word, θ_t^T is parameter for each target word, $e_c$is parameter (the embedding vector) for each context word.

Positive samples can be obtained by the standard skip-gram method, where we sample the input sentence randomly for a context word and then we sample in close proximity (using a window of +-5 words) to the target for another word (which we label as target).

Negative samples can be obtained using the same steps, except for the target word we choose a random word from the vocabulary (not related with actual target).

- Then, there is a supervised learning problem. There is a pair context-word on the input previously obtained), and has to predict target label.

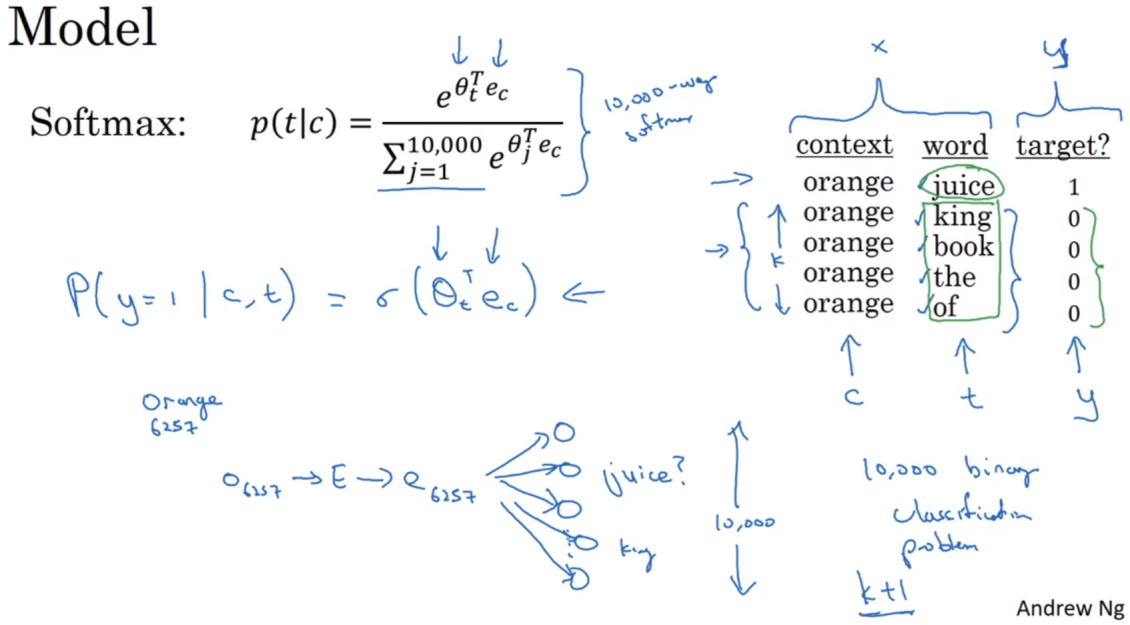


Figure 5.31: Negative sampling in word embeddings.$e_c$is an embedding vector or context word, and t means possible target. In this problem, we will find a logistic regression model. We have $k:1$ ratio of negative to positive examples. NN for this problem is build in the following way: on the input, there is a word 'orange'. One-hot vector is created with usage of our dictionary. Next, the embedding vector is created from this word, and then we have n (size of our dictionary) possible logistic regression models. One such model is a binary classifier corresponding to whether a given context word coresponds to target 'juice' or not, and similarly for all such classifiers. So, word2vec was very computationally expensive, and giant softmax (in this example of 10,000 classes) was replaced by many binary classifiers (in this example 10,000 of them) each of which is quite cheap to compute. And on every iteration, we're only going to train $k+1$ of them (in this example 5) - k negative examples and 1 positive example. And this is why the computation cost of this algorithm is much lower because we are updating $k+1$ let's just say units, ($k+1$ binary classification problems).

- **GloVe word vectors** (Global vectors for word representation), is another way of computing word embeddings (Pennington et al, from 2014). It is even simpler algorithm than Negative sampling. It is not used so much; only sometimes, because of its simplicity. The authors defined $X_{ij} = X_{ji}$ which is a number of times i (possible target word) appears in context of j (context word). And vice versa also works because it is symmetric relation. This algorithms do not sample word pairs by picking words that appeared in close proximity to each other,

as word2vec uses. Instead, counting of these sampled pairs is explicit. It iterates through the training set and counts how many times target word appears in proximity of context word.

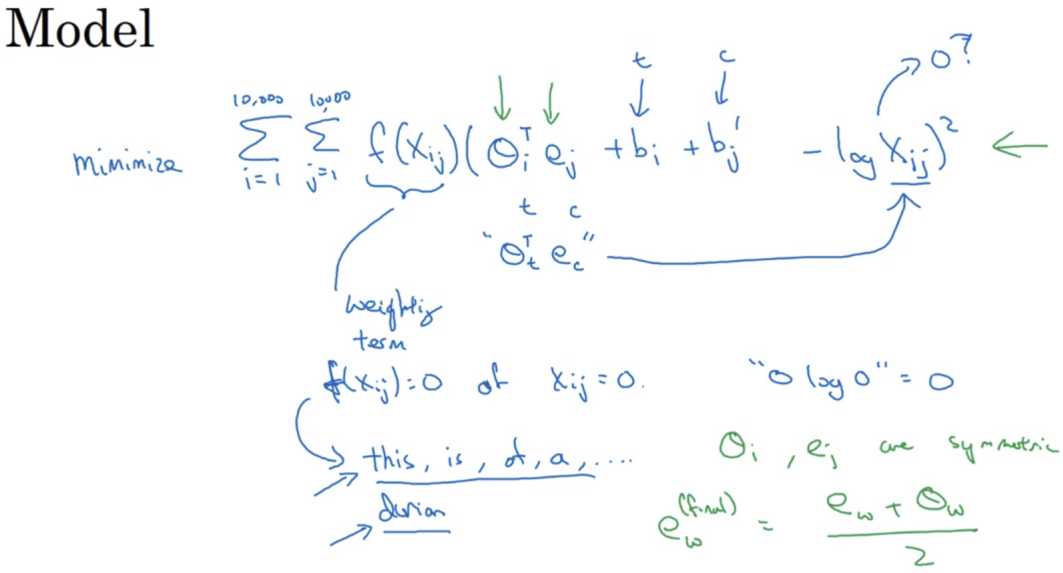


Figure 5.32: A model of GloVe algorithm.

– One of the most fascinating properties of word embeddings is that they can also help with **analogy reasoning**. An interesting property of these word embeddings (vectors) is that we can subtract them, for example $e_{man} - e_{woman} = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ if our embeddings has 4 dimensions. And let's say that the first dimension was *gender*, so this means that the most difference between *man* and *woman* is *gender*. Then we can perform: $e_{man} - e_{woman} \approx e_{king} - e_w$ and find word e_w as follows: $argmax_w\, sim(e_w, e_{king} - e_{man} + e_{woman})$ and hopefully find word "queen".
– **Similarity** (for instance from the previous example) between 2 word embeddings (vectors), can be calculated as **cosine similarity** (one of the most used one), which is its multiplication (inner product) divided by the Euclidean lengths:

$$sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2} \tag{5.3}$$

– **Debiasing word vectors**

* Example for intuition: if we have a vector g which roughly represents a gender (as a result of $e_{man} - e_{woman}$ for example, or using more of such vectors, like $e_{boy} - e_{girl}$, and computation of average from these results). Now, words "John" or "Ronaldo" will be closer to this vector g as negative values, and words "Marie" or "Katy" will be closer to this vector g and will be positive values. Now, similarity between words "literature" and g vector is a positive number, and similarity between "engineer" and g are negative values. We don't want that literature is somehow more similar to female gender, and engineer is more similar to male gender. There is a way of neutralizing such bias (in this example, for non-gender specific words).
* If you're using for example a 50-dimensional word embedding, the 50 dimensional space can be split into 2 parts: the **bias-direction** g (a vector that represents "gender" for instance - as been in the example above), and the remaining 49 dimensions, which we'll call $g_{\perp}$. In linear algebra, we say that the 49 dimensional $g_{\perp}$ is perpendicular ("othogonal") to g, meaning it is at 90 degrees to g. The neutralization step takes a vector such as $e_{receptionist}$ and zeros out the component in the direction of g, giving us $e^{debiased}_{receptionist}$.
* Given an input embedding e, the following formulas are used to compute $e^{debiased}$,where $e^{bias\,component}$ (there is a L2 / Frobenius form powered by 2 in the denominator) is the projection of e onto the direction g (so e is positively correlated with the bias axis g):

$$e^{bias\,component} = \frac{e.g}{||g||_2^2} * g \tag{5.4}$$

$$e^{debiased} = e - e^{bias\,component} \tag{5.5}$$

* **Equalization** is another thing that needs to be done, so that debiasing algorithm can be applied to pairs like "actress" and "actor". So, equalization is applied to pairs of words that you might want to have differ only through, for example, the gender property.
* As a concrete example, suppose that "actress" is closer to "babysit" than "actor." By applying neutralizing to "babysit" we can reduce the gender-stereotype associated with babysitting. But this still does not guarantee that "actor" and "actress" are equidistant from "babysit." The equalization algorithm takes care of this. So the key idea behind equalization is to make sure that a particular pair of words are equidistant from the 49-dimensional $g_{\perp}$.

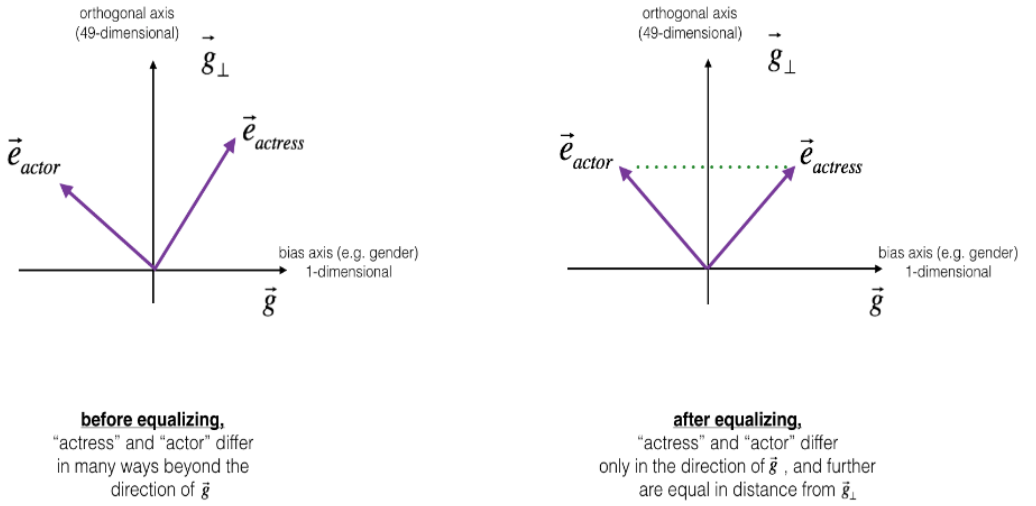


Figure 5.33: An intuition about equalization for debiasing of word embeddings.

$$\mu = \frac{e_{w1} + e_{w2}}{2}$$

$$\mu_B = \frac{\mu \cdot \text{bias\_axis}}{\|\text{bias\_axis}\|_2^2} * \text{bias\_axis}$$

$$\mu_\perp = \mu - \mu_B$$

$$e_{w1B} = \frac{e_{w1} \cdot \text{bias\_axis}}{\|\text{bias\_axis}\|_2^2} * \text{bias\_axis}$$

$$e_{w2B} = \frac{e_{w2} \cdot \text{bias\_axis}}{\|\text{bias\_axis}\|_2^2} * \text{bias\_axis}$$

$$e_{w1B}^{corrected} = \sqrt{|1 - \|\mu_\perp\|_2^2|} * \frac{e_{\text{w1B}} - \mu_B}{|(e_{w1} - \mu_\perp) - \mu_B)|}$$

$$e_{w2B}^{corrected} = \sqrt{|1 - \|\mu_\perp\|_2^2|} * \frac{e_{\text{w2B}} - \mu_B}{|(e_{w2} - \mu_\perp) - \mu_B)|}$$

$$e_1 = e_{w1B}^{corrected} + \mu_\perp$$
$$e_2 = e_{w2B}^{corrected} + \mu_\perp$$

Figure 5.34: Equalization formulas for debiasing of word embeddings. So as first, mean is computed (1). Then the projections of the mean over the bias axis (2) and then orthogonal axis (3). Then projections of embedding words over the bias axis (4 and 5). After that, adjust the bias part of the last resulting projections (6 and 7) which will be debiased by equalizing resulting adjusted vectors to the sum of their corrected projections (8 and 9).

- There exist a multiple different **basic blocks** (units):
  - **A simple RNN unit**

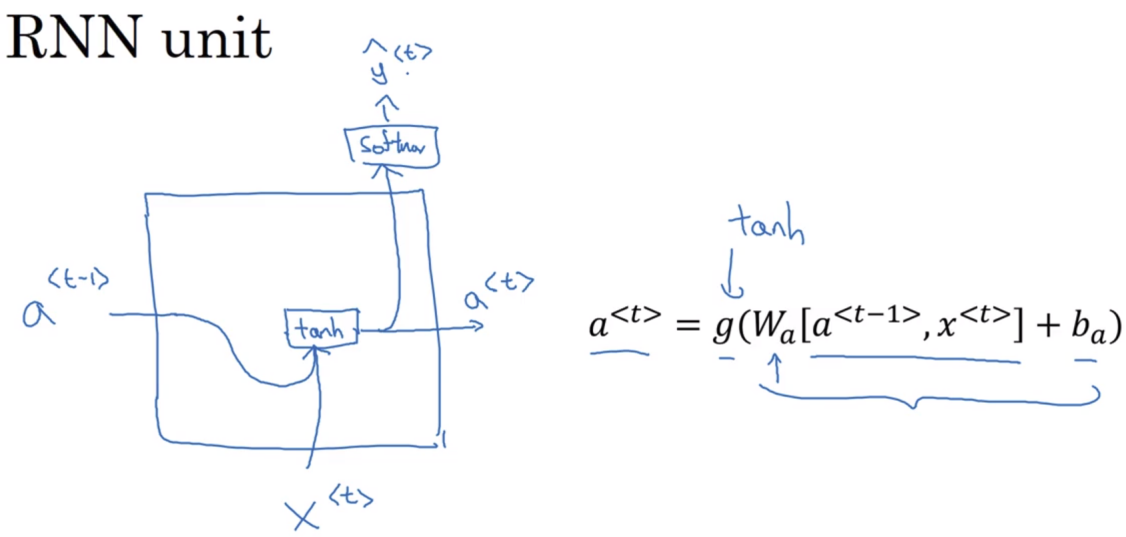


Figure 5.35: A simple RNN unit example.

– **Gated Recurrent Unit (GRU)**

  * A solution (Cho et al, and Chunk et al, both papers from 2014) to prevent gradient vanishing and it is **much better with capturing long range connections** (as well as LSTM is).
  * **There is a memory cell** $c^{<t>} = a^{<t>}$ (which will help to remember to capture longer-range connections) and a **gate unit** Γ_u which is a value between 0 and 1 calculated as $\tilde{c}^{<t>} = tanh(w_c[c, x^{<t>}] + b_c)$ but with sigmoid (which is mostly very close to 0 or very close to 1) instead of tanh activation function. 'u' Γ_u in stands for an update rule. The actual value of $c^{<t>}$is influenced by Γ_u- see for yourself, what is the result when it is 0 or 1 (well, not exactly, just approximately, it is result of sigmoid):

$$c^{<t>} = \Gamma_u.\tilde{c}^{<t>} + (1 - \Gamma_u).c^{<t-1>} \tag{5.6}$$

    so when Γ_u is 1, then $c^{<t>} = \tilde{c}^{<t>}$which is updating, or when Γ_u is 0, then $c^{<t>} = c^{<t-1>}$so do not update $c^{<t>}$, keep remembering the old value.
  * Value of $c^{<t>}$is maintained if the update is not performed, and therefore gradient vanishing problem is not present here. At each time step (layer), we will consider (actually, update gate will determine this) whether to update the memory cell unit value with a new value, or not.

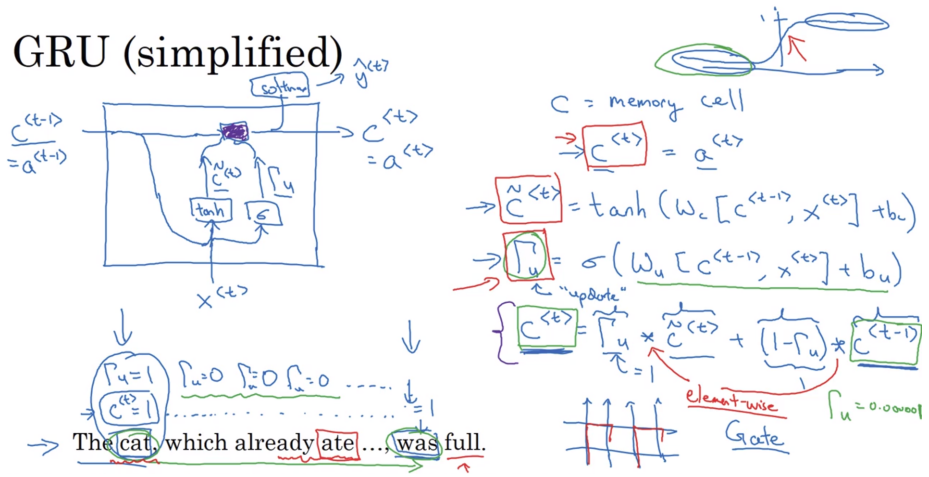


Figure 5.36: A simplified GRU unit in RNN. $c^{<t>}$, Γ_u, and $\tilde{c}^{<t>}$ all have the same dimension. "Full version" of GRU changes $\tilde{c}^{<t>}$ a bit, it is computed as $\tilde{c}^{<t>} = tanh(w_c[\Gamma_r.c^{<t-1>}, x^{<t>}] + b_c)$, where Γ_r is relevance gate (how relevant is $c^{<t-1>}$ to computing the next update candidate $\tilde{c}^{<t>}$) and is calculated similarly as Γ_u: $\Gamma_r = sigmoid(w_r[c^{<t-1>}, x^{<t>}] + b_r)$. Both $c^{<t>}$ and Γ_u are calculated as before.

– **Long Short-term Memory (LSTM) unit**

  * Slightly more powerful and more general version of GRU (Hochreiter and Schmidhuber, from 1997).
  * The idea is, you want to make that short term memory lasts for a long time. This is done by creating special modules that are designed to allow information to be gated in, and then information to be gated out when needed. And in the intermediate period, the gate is closed, so the stuff that arrives in the intermediate period doesn't interfere with the remembered state.
  * A memory cell $c^{<t>} \neq a^{<t>}$ and there is no relevance gate Γ_r. Also, instead of one update gate Γ_u, there are 2 gates: update gate Γ_u and forget gate Γ_f. There is also output gate Γ_o which is used as follows: $a^{<t>} = \Gamma_o.c^{<t>}$ (element-wise multiplication). All gate values are computed using $a^{<t-1>}$ and $x^{<t>}$, but they have different weights and biases. That's all.
    · **Forget gate** - for the sake of this illustration, lets assume we are reading words in a piece of text, and want use an LSTM to keep track of grammatical structures, such as whether the subject is singular or plural. If the subject changes from a singular word to a plural word, we need to find a way to get rid of our previously stored memory value of the singular/plural state.

- **Update gate** - we forget that the subject being discussed is singular, we need to find a way to update it to reflect that the new subject is now plural.
- **Updating the cell** - to update the new subject we need to create a new vector of numbers that we can add to our previous cell state (this is $\tilde{c}^{<t>}$).
- **Output gate** - to decide which outputs we will use.

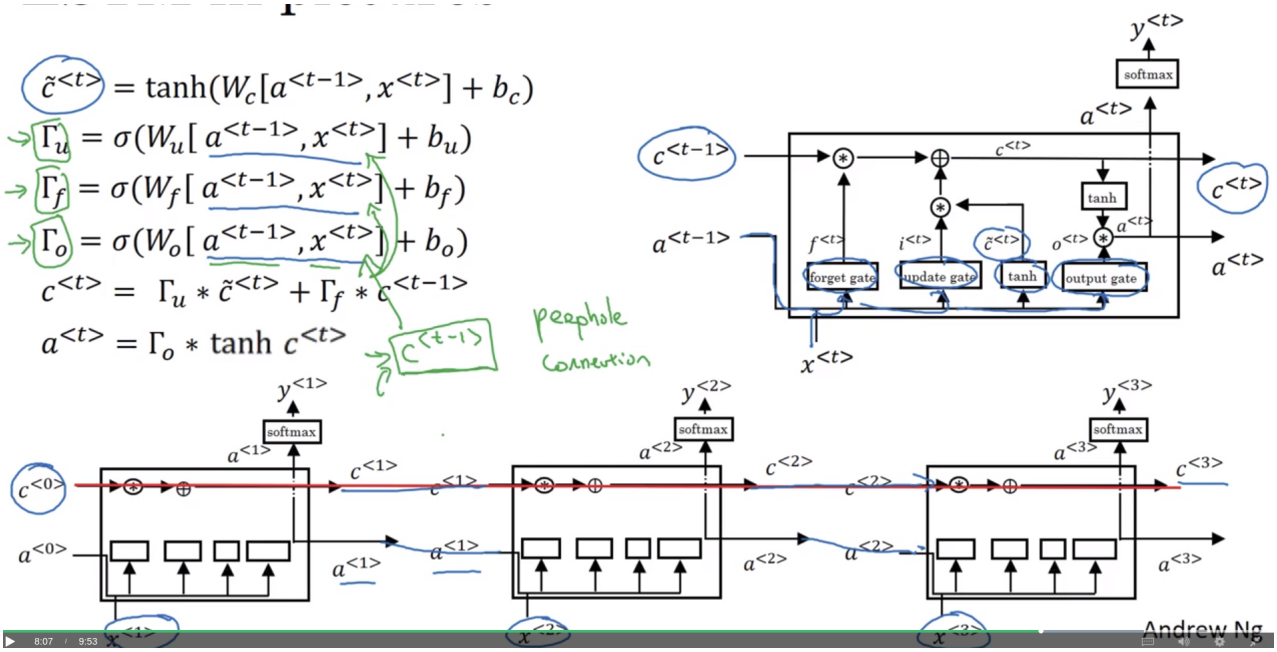


Figure 5.37: LSTM unit with calculation of each part. Math formulas are easier and more understandable than diagrams, but here are both. There exist many variations to LSTM, one of them is so called **peephole connection** as detailed in the figure.

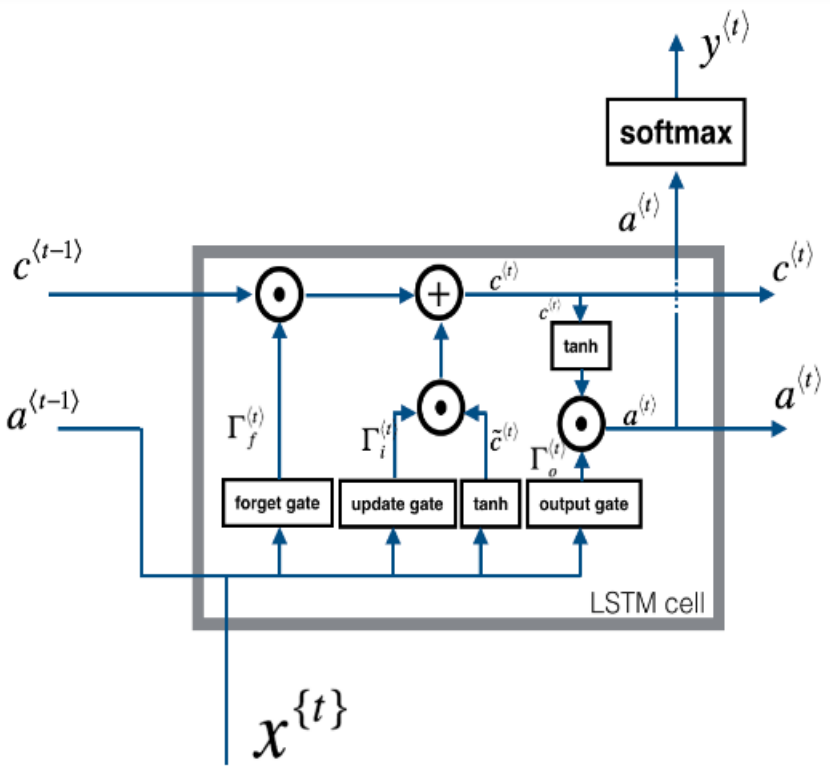


Figure 5.38: LSTM-cell. This tracks and updates a "cell state" or memory variable $c^{\langle t \rangle}$ at every time-step, which can be different from $a^{\langle t \rangle}$.

  * LSTM were historically much earlier, and GRU was probably created as a simplification to LSTM. Neither of them is superior to the other one - which one to use depends on an application. However, since GRU is simpler, it is easier to build much bigger model with it = so it also runs computationally much faster, but LSTM is more powerful and flexible.

– **Bidirectional RNNs**

  * These models let you at a point in time to take an information from both - earlier and later in the sequence.
  * This consists of basically 2 activation layers, one is in forward direction and the other one is in backward direction (both are a part of the forward propagation! No gradients are computed here, and weights are not updated when computing activations here - they are called forward $\overrightarrow{a^{<t>}}$ and backward $\overleftarrow{a^{<t>}}$ activations) - so no feedback or loops are there - BRNN is still **acyclic graph**!
  * After computing both forward and backward activations, we can output a prediction as follows:

$\hat{y}^{<t>} = g(W_y[\overrightarrow{a^{<t>}}, \overleftarrow{a^{<t>}}] + b_y)$

* Activation block can be standard RNN, GRU, or LSTM units.
* A disadvantage of BRNNs is that we need to have the entire sequence available before we can compute a predicted output. So, for example, for (real-time) speech recognition problem, we would need to wait for the person to stop talking before we can process the data.

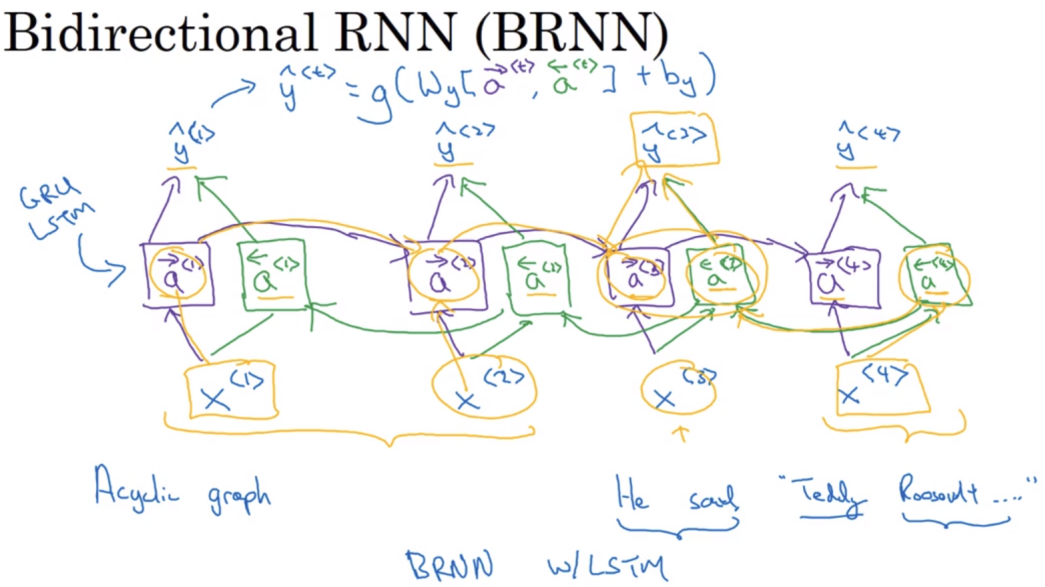


Figure 5.39: Bidirectional RNN architecture. A simple block can be either a simple RNN block, GRU, or LSTM (this one is often used).

- Standard ANNs may have over hundred hidden layers. RNNs use far less layers, even 3 hidden layers is considered as deep. Because of the time dimension, these networks can already get quite big even if you have just a small number of layers. And you don't usually see these stacked up to be like 100 layers. One thing that can be seen sometimes is that recurrent layers are stacked on top of each other. Or, after a few RNN layers, and then a bunch of layers not connected horizontally.
- **Sequence to Sequence architectures**
  - Sequence-to-sequence learning (seq2seq learning) is a generalization of the sequence labeling problem. Input list of a feature vectors (for example words) can have different length than list of labels (may be also words, for example in machine translation). Many seq2seq learning problems are currently (2019) best solved by neural networks. These architectures all have 2 parts: an encoder an a decoder.

- These models are basically **many-to-many RNNs** where the input and **output sequences have different lengths**. Inputs are processed by RNN called **encoder**. RNN then offers a vector that represents the input sentence (to encoder which can be RNN or CNN or some other architecture - its role is to read the input and to generate some sort of state that can be seen as a numerical representation of the meaning of the input the machine can work with - usually some vector / matrix of real numbers - this vector / matrix is called **embedding** of the input). After that, a **decoder network** follows. This one takes as input the encoding output by the encoder network (so, its input is an embedding), and then can be trained to output the translation one word at a time until eventually it outputs say, the end of sequence or end the sentence token upon which the decoder stops = so decoder is capable of generating a sequence of outputs, starting with some start of sequence input feature vector (usually all zeroes) $x^{(0)}$ to produce the first output $y^{(1)}$, update its state by combining the embedding and this input $x^{(0)}$, and then uses generated output $y^{(1)}$ as its next input $x^{(1)}$. Dimensionality of $y^{(t)}$ can be the same as dimensionality of $x^{(t)}$, however it is not strictly necessary.
- Both encoder and decoder are trained simultaneously using the training data. The errors at the decoder output are propagated to the encoder via backpropagation.
- They are useful for everything from machine translation to speech recognition. Given enough for example pairs of English and French sentences, translation will work decently well. This also works well on image captioning (input image -> output sentence) - CNN (encoder) + RNN (decoder).
- More accurate predictions can be obtained using an architecture with **attention**. Attention mechanism is implemented by an additional set of parameters that **combine some information from the encoder (**in RNNs, this information is the list of state vectors of the last recurrent layer from all encoder time steps) **and the current state of the decoder to generate the label.** That allows for even better retention of long-term dependencies than provided by gated units and bidirectional RNN.

5.7 Convolutional Neural Networks

- They are normal ANN, with some hierarchical structure, and are very deep usually - even 10 or 20 hidden layers. The layers have fixed structure of connections. Instead of matrices of weights, here are convolutional cores (some floating point numbers, but they are basically still normal weights).
- It can be said, that a CNN is a special kind of FFNN (feed-forward neural network) that significantly reduces the number of parameters in a deep neural network with many units without losing too much in the quality of the model.
- Original idea is from 1970s, but the seminal paper establishing modern subject of convolutional networks was in 1998 (Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Hattner.
- They use 3 basic ideas:
  - **local receptive fields**: we make connections in small, localized regions of the input image (because CNNs are mostly used in image classification problems) - instead of a vertical line of neurons, where each input pixel was connected to every hidden neuron as it was done in traditional feed-forward neural networks. So, that region in the input image is called the local receptive field for the hidden neuron - a little window on the input pixels. Each connection learns a weight, and a hidden neuron learns an overall bias as well. We are performing sliding of the local receptive field across the entire input image, for each local receptive field there is a different hidden neuron in the first hidden layer.
  - **shared weights**: as been said above, each hidden neuron has a bias and n weights connected to its local receptive field (if, for example, we are taking 5x5 pixels as our region, then it would mean that $n = 25$). The same weights and bias are used for each hidden neuron. This means, that all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image. Which makes sense. Suppose, that weights and bias are such that the hidden neuron can pick out, say, a vertical edge in a particular local receptive field. This ability is also likely to be useful at other places in the image. So it is useful to apply the same feature detector (feature - in this meaning, it is some kind of input pattern that will cause the neuron to activate) everywhere in the image. CNNs are well adapted to the translation invariance of images. The shared weights and bias are often said to define a **kernel** or **filter** (=feature map). There can be, and mostly are, in one convolutional layer a several different feature maps. So, for example, if we have 3 feature maps, each has 5x5 shared weights and a single shared bias, the result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image. The advantage of sharing

weights and biases is also that it greatly reduces the number of parameters involved in a CNN.

- **pooling (layers)**: these are usually used immediately after convolutional layers. They simplify the information in the output from the convolutional layer. As mentioned above, convolutional layer usually involves more than a single feature map. WE apply pooling (for example max-pooling) to each feature map separately. We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. This also helps to reduce the number of parameters needed in later layers. L2 pooling is another technique, where we take the square root of the sum of the squares of activations in some region. The details are different, but the intuition is similar to max-pooling - it is also a way of condensing information from the convolutional layer. Choosing which pooling to use is another hyper-parameter of the network. After convolutional and pooling layers (more such ones, as a cascade), there can be eventually for example a fully-connected layer - so from max-pooled layer, there is a connection to every one of output neurons.

- We can have the output (like what is there) of some image, but also on a particular pixel!

- There are:
  - **Convolutional layers**, shortly **Conv** (good thing is to use relatively small convolutional cores - M. Hradis).
    - They are randomly initialized, and ANN will learn them automatically. For decades, CV researchers had hand-crafted filters like this, but their results were not so good as ones from CNNs.
    - In fact, using convolutional layers greatly reduces the number of parameters in those layers, making the learning problem much easier. If you use ReLU as the activation function, you can even speed up training (in comparisonm with sigmoid activation functions, the speedup can be 3-5 times).
    - **There are two main advantages of convolutional layers over just using fully connected layers - parameter sharing** (less parameters to learn - so it also reduces overfitting - a feature detector, such as vertical edge detector, that is useful in 1 part of the image is probably useful in another part of the image; so it allows a feature detector to be used in multiple locations throughout the whole input image/volume) and **sparsity of connections** (in each layer, each output value depends only on a small number of inputs - activations from a previous layer).

    - **Dropout applied on these layers** - it is possible, but in fact there's no need: the convolutional layers have considerable inbuilt resistance to overfitting. The reason is that the shared weights mean that convolutional filters are forced to learn from across the entire image. This makes them less likely to pick up on local idiosyncrasies (=features/peculiarities) in the training data, so **there is less need to apply other regularizers**, such as dropout.
  - **Pooling layers**, shortly **Pool** (for reducing size of the input and thus **speeding up the computation**). Hyperparameters of these layers are: filter size (usually equals to 2 or 3), stride (usually equals to 2), and a type of pooling layer - padding is rarely used. Otherwise, there are no parameters to learn. There are 2 types of pooling layers:
    - average - take an average value from a given region of an input.
    - max pooling - take a max number from a given region of an input. This kind of pooling layer is used more often than the previous one. However, in a very deep neural network, some people use average pooling layer instead.
  - **Fully connected layers**, shortly **FC.**

- Why traditional ANN are not enough? Because they would work with information which is not sequentially near by. It's like I have a bunch of separated pixels - it is much harder to say what is on image. Also, if you implement a system for speech recognition, the word 'hello' can be told in the beginning, end, in the middle, it does not matter.

- **Convolutional kernels** - dimensions (shape also depends - if I want to detect SPZ in cars, then horizontal shape is a good idea), step (stride), number of kernels, padding or the input space.

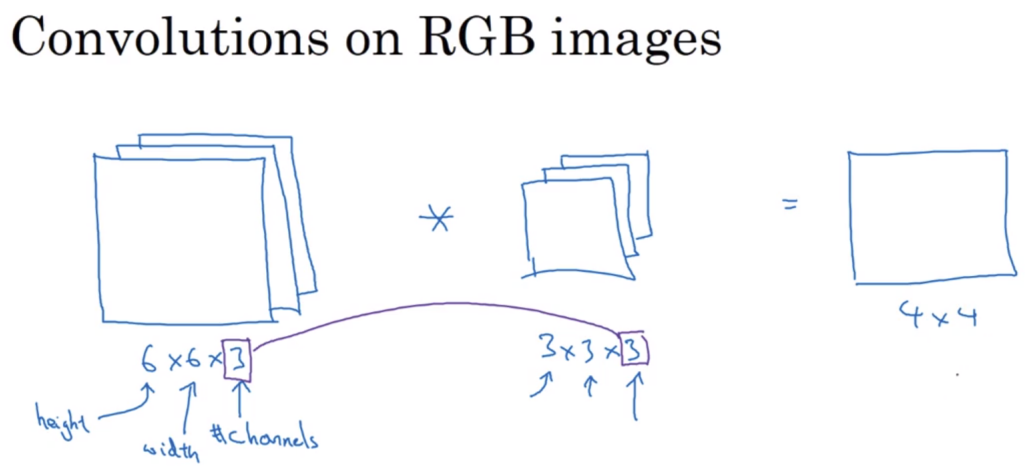


Figure 5.40: An example of convolution operation with 3x3x3 kernel on RGB image. The last dimension is called **channel** and must be the same in an input image and corresponding kernel. The result is then 4x4 dimensional (no channels).

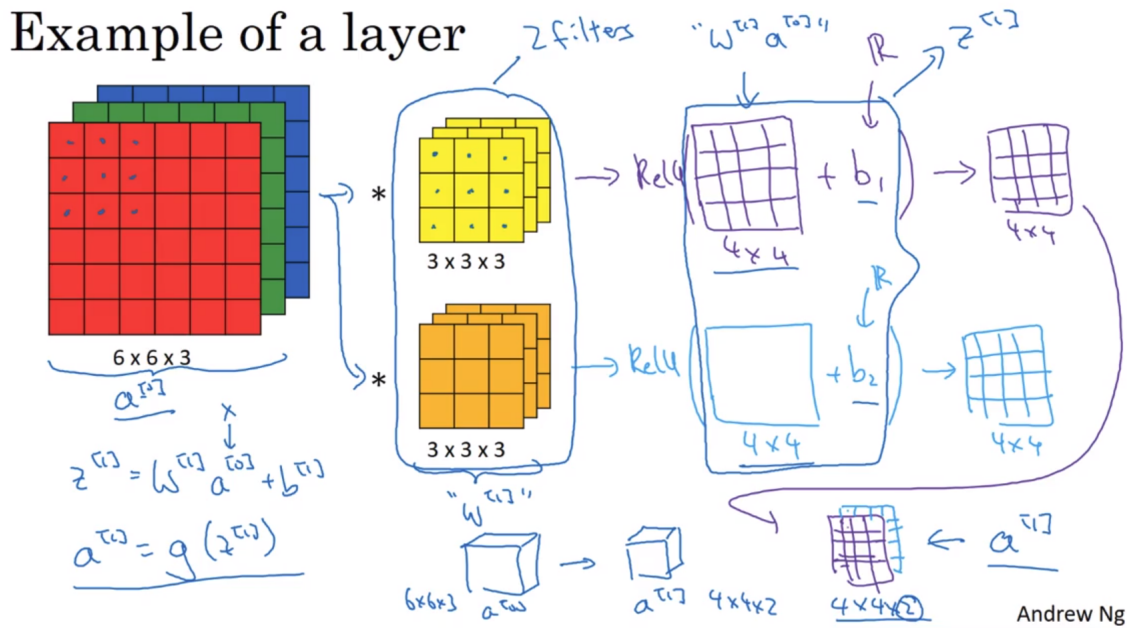


Figure 5.41: An example of a simple convolution layer with 2 filters. **We can use multiple filters** (stacking them), and then we would have 'channels' on the output (for example, if we would like to apply horoizontal and also vertical filters on an input).

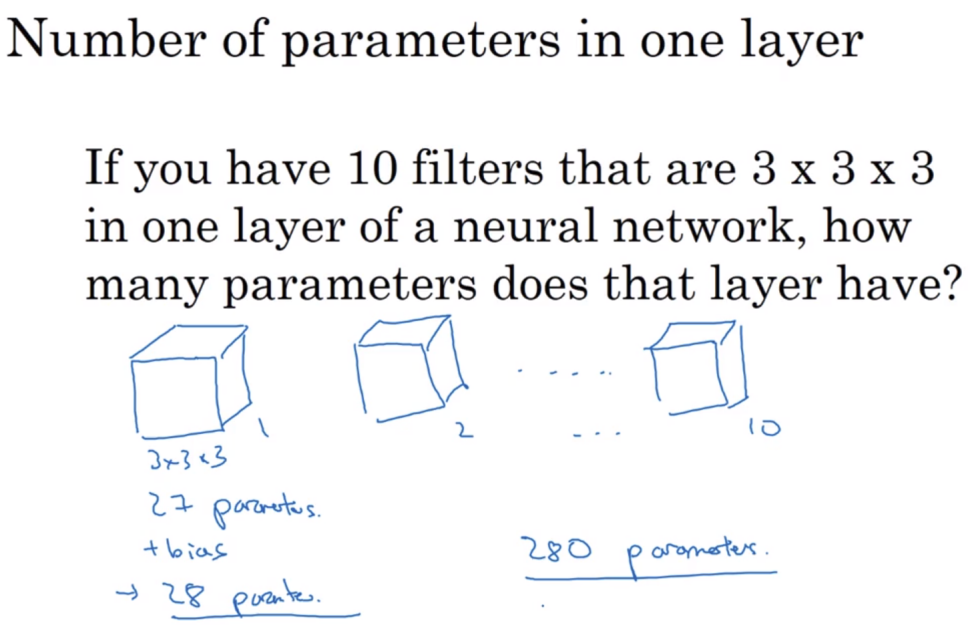


Figure 5.42: An example of parameters number in 1 layer consisting of 10 kernels. A nice thing about this is, that no matter how big the input image is (could be 1,000 x 1,000 or 5,000 x 5,000 pixels), but the number of parameters you have still remains fixed as 280. And you can use these ten filters to detect features, vertical edges, horizontal edges maybe other features anywhere even in a very, very large image is just a very small number of parameters.

- **Convolutional operation vs cross-correlation**: convolution is when we flip (by diagonal) a filter (and this operation is associative then, which can be useful for some signal processing problems), but in deep learning literature, the name **convolution** is used even without any flipping and we do not use cross-correlation (in deep learning we don't care about this operation being associative).
- **1x1 Convolutions**:
  - An idea from 2013 sometimes also called as Network in Network. Very influential even it is not so widely used, this idea has been inspirational also for Inception Network.
  - You basically multiply the whole input matrix by a number. The trick is, when more channels are used. Then it is multiplication by 1 number, by on each channel (the same position) and the ReLU (for instance) activation function is used.
  - Basically an idea is to have a fully connected neural network that applies to each channel and outputs with a dimension equal to the number of filters.
  - If you need to shrink width and height, you can use pooling layer. If you want to reduce number of channels (but also increase, or no change at all),

you can use 1x1 convolutions. For example, input is 28x28x192 and you want to have on output 28x28x32. Then if you use 1x1x192 convolution (32 such units and it needs to have the same number of channels as the input), it will make desirable output.

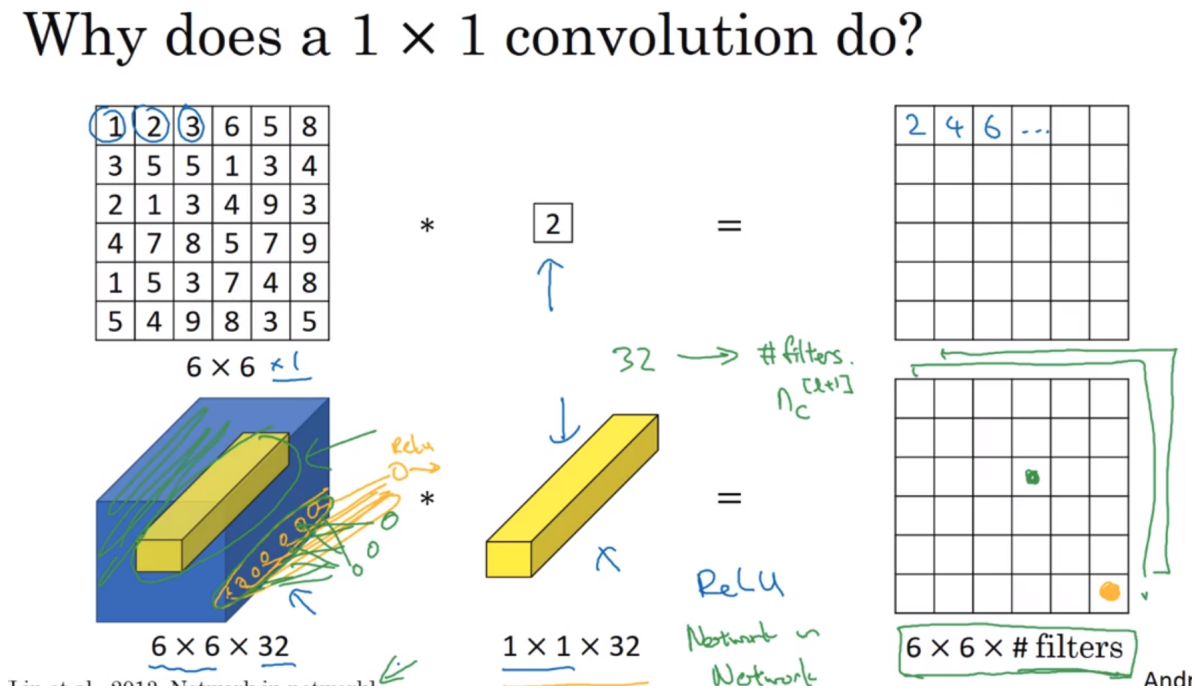


Figure 5.43: 1x1 Convolution on a simple 2d image and on an input with 32 channels.

- **Padding**
  - A modification that needs to be done to traditional kernel operation.
  - **Example**: if you take 6x6 matrix and kernel of size 3x3, after convolution operation, you get 4x4 matrix as a result. Because the number of positions for a convolutional filter (=kernel) is sort of 4x4 possible positions. (if you take $n\,x\,n$ image, and $f\,x\,f$ kernel, the output is $n-f+1\,x\,n-f+1$ (so called **valid convolution**). So every time you apply convolutional operator, an input (image) **shrinks**. If you have a very deep net, this is a bad thing because you don't want to shrink your image after each layer, resulting in shrinking an image too much. Padding help to prevent this shrinking. If you pad the whole image by 1 pixel in each of 4 directions (additional border of 1 pixel all around the edges), no shrinking will happen (for this case though).
  - How much padding to stop shrinking? $n+2p-f+1\,x\,n+2p-f+1$ (so called **same convolution** = input and output size are the same) where p is padding level, in the previous example it was $p=1$, so border of 1 pixel. So

$p = \frac{f-1}{2}$ (from the previous definition) is a way to make same convolution. f is almost always odd number (1 dimension of filter) - probably because you can have a "central" pixel with odd filter, and also this equation of p would need some modification.

  - You pad usually with zeros.
  - Padding is helpful with larger filters, because it allows them to better "scan" the boundaries of the image.

- **Strided convolutions** - steps for which convolution filter is moving can be higher than 1. For example $stride = 2$ to move always by 2 steps. But with this, you also **shrink** an input. $floor(\frac{n+2p-f}{S} + 1)\, x\, floor(\frac{n+2p-f}{S} + 1)$ where S is $stride$ and $floor$ is truncation of float number to get an integer.

- Image classification and relative problems:
  - Further and further it goes (next and next layer), the lower resolution the image has. And this is done mostly by pooling layer (max/mean operations). Max-pooling - from the nearby pixels, take only max value to the output.
  - In images it goes from high-res pixels, to fine features (edges, circles,....) to coarse features (noses, eyes, lips, ... on faces), then to the fully connected layers that can identify what is in the image[5].

- They are less prune to overfitting, than normal NN. This is because of convolutional cores, which see many input samples.

- They fit good to problems which have some structure, some signal - sound, images for instance (but not only to these families of problems of course). For example, if we would reorder (in consistent way = for every image, the same reorder) some input data, then the classical neural network does not see the difference. But in CNN, the network does not need to "learn" / know that the pixels are one after another.

- So there are multiple additional hyperparameters in ConvNet (convolutional neural network): stride, padding, filter size, how many filters to use (in a given layer), ... Usually, a trend is to lower down "resolution" of an input image, and increase number of channels.

[5] https://www.quora.com/Is-machine-learning-anything-more-than-an-automated-statistician

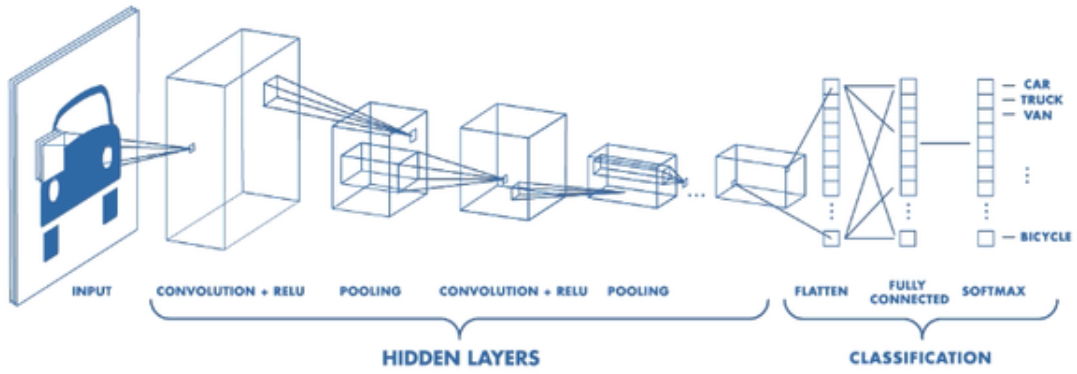


Figure 5.44: An example of Convolutional Neural Networks.

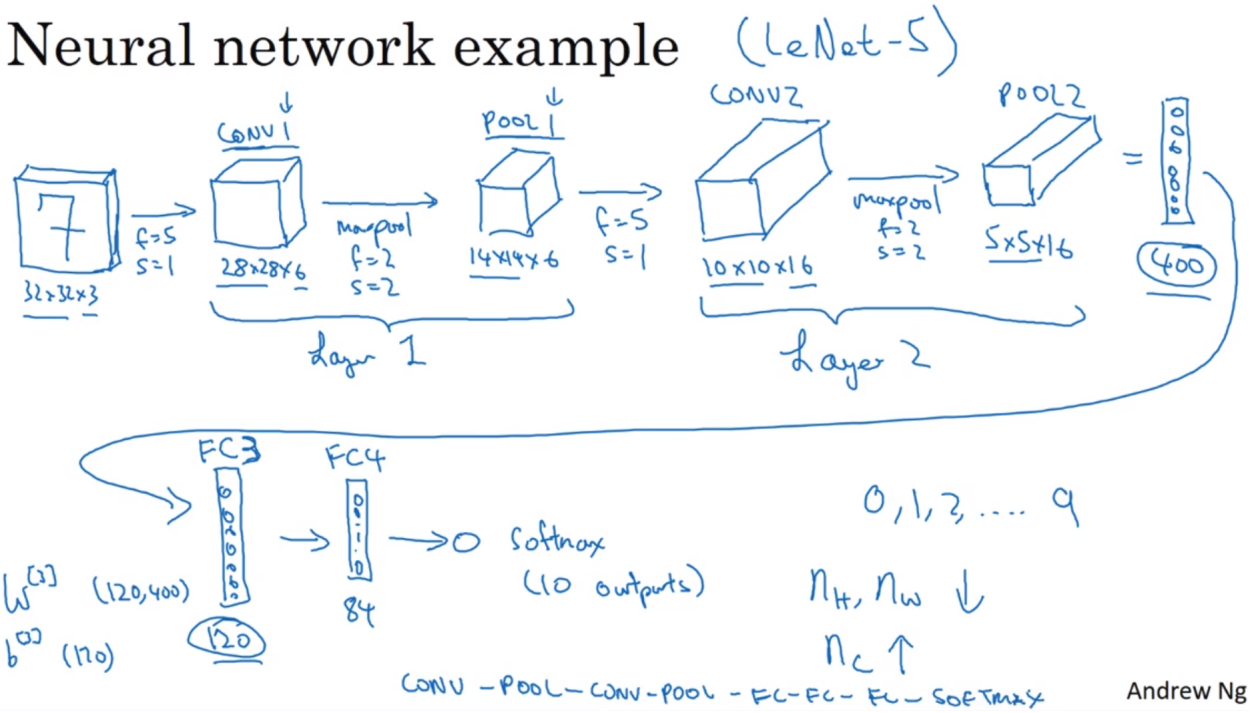


Figure 5.45: Another example of Convolutional Neural Networks. Here on this figure we can see a common pattern - convolutional layer, followed by pooling layer (max pooling usually), and this is repeated untill a few fully connected layers, and then the last layer, usually with softmax activation function.

Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Andrew Ng

Figure 5.46: A dimensions of each layer in CNN from the previous figure.

- **YOLO** (You Only Look Once, Redmon et all, from 2015) - a solution to convolutional implementation of sliding window (for locating accurately bounding boxes from an input image).

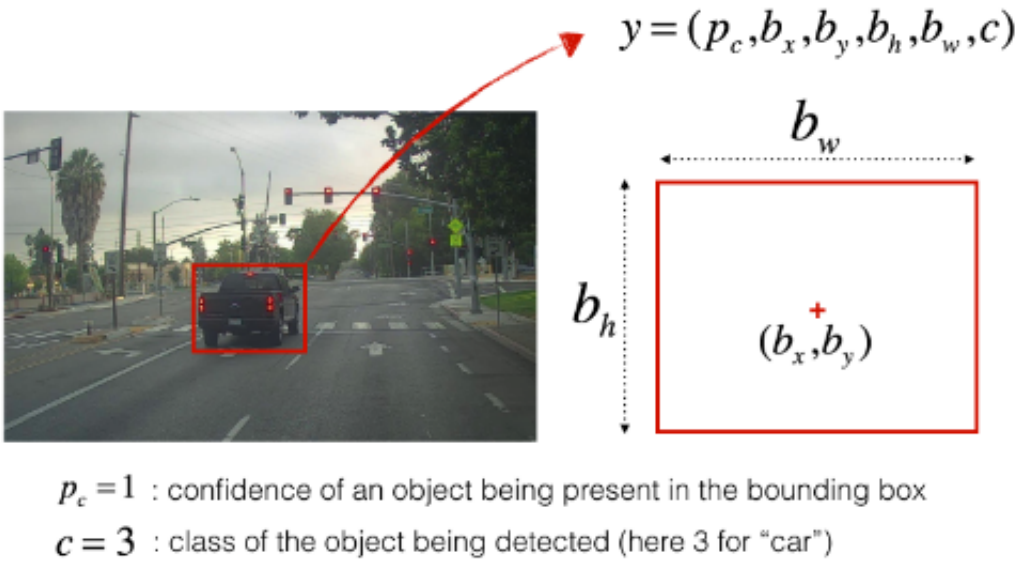


Figure 5.47: Definition of a bounding box. However, if you expand c into an 80-dimensional vector (if there are 80 classes), each bounding box is then represented by 85 numbers.

- Usage for **object detection. One of the most promising algorithm so far.** There is "**intersection over union**" (IoU) function for evaluation of object detection algorithm. This function is used for evaluating bounding boxes - if there is ground truth, and then predicted bounding box - how good is the predicted one? If it is slightly moved for example. So it computes the intersection over union of these two bounding boxes - $\frac{size\,of\,intersection}{size\,of\,union}$. A low CV task will judge that the predicted answer is correct if this value is greater equal to 0.5 (usually between 0.5 and 0.7). The higher IoU is, the more accurate prediction we have.
- Popular algorithm because it achieves high accuracy while also being able to run in real-time. This algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

- The **input** is a batch of images of shape (m, 608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

We will use 5 anchor boxes. So you can think of the YOLO architecture as the following: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

Lets look in greater detail at what this encoding represents.

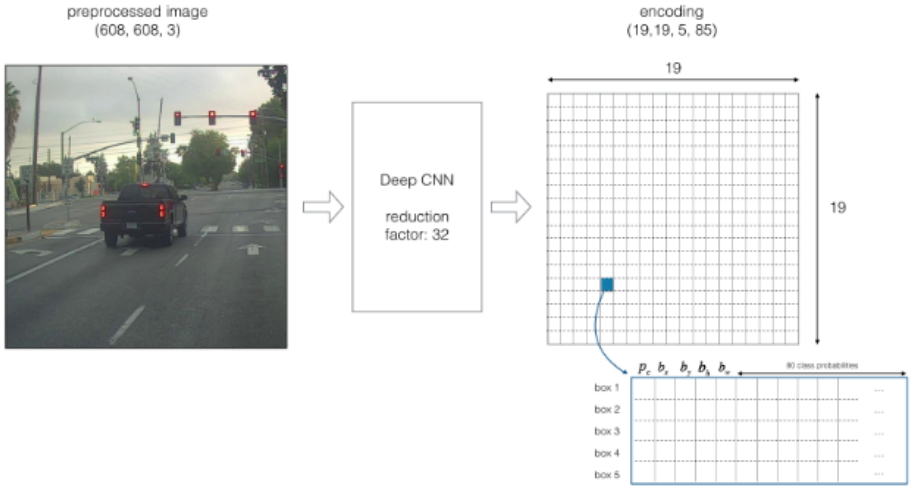


Figure 5.48: An example of Encoding architecture for YOLO. If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object. Since we are using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height. **For simplicity, we can flat the last two last dimensions** of the shape (19, 19, 5, 85) encoding. So the output of the Deep CNN is (19, 19, 425).

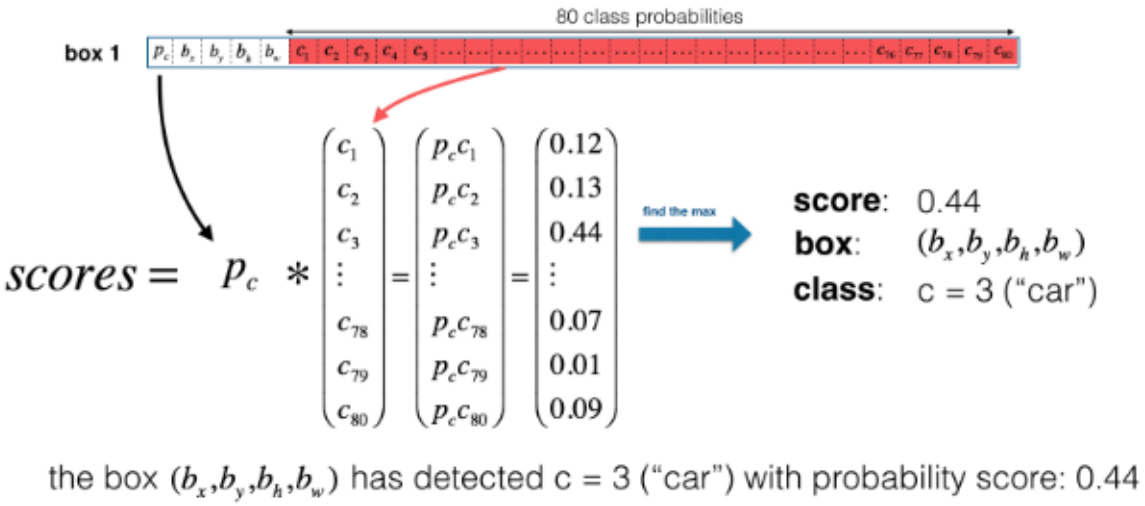


Figure 5.49: An example of finding a class detected by each box. This is a continuation of the previous figure. Now, for each box (of each cell) we will compute the following elementwise product and extract a probability that the box contains a certain class.

Here's one way to visualize what YOLO is predicting on an image:

- For each of the 19x19 grid cells, find the maximum of the probability scores (taking a max across both the 5 anchor boxes and across different classes).
- Color that grid cell according to what object that grid cell considers the most likely.

Doing this results in this picture:

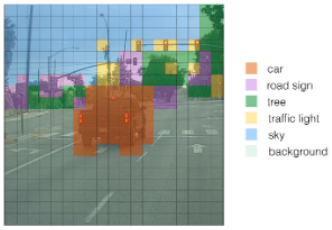


Figure 5 : Each of the 19x19 grid cells colored according to which class has the largest predicted probability in that cell.

Note that this visualization isn't a core part of the YOLO algorithm itself for making predictions; it's just a nice way of visualizing an intermediate result of the algorithm.

Figure 5.50: Visualization of results from YOLO algorithm. Another way to visualize YOLO's output is to plot the bounding boxes that it outputs. Doing that results in a very chaotic visualization = there would be too many boxes. You'd like to filter the algorithm's output down to a much smaller number of detected objects. To do so, you'll use **non-max suppression:** i) get rid of boxes with a low score (thresholding), ii) select only one box when several boxes overlap with each other and detect the same object.

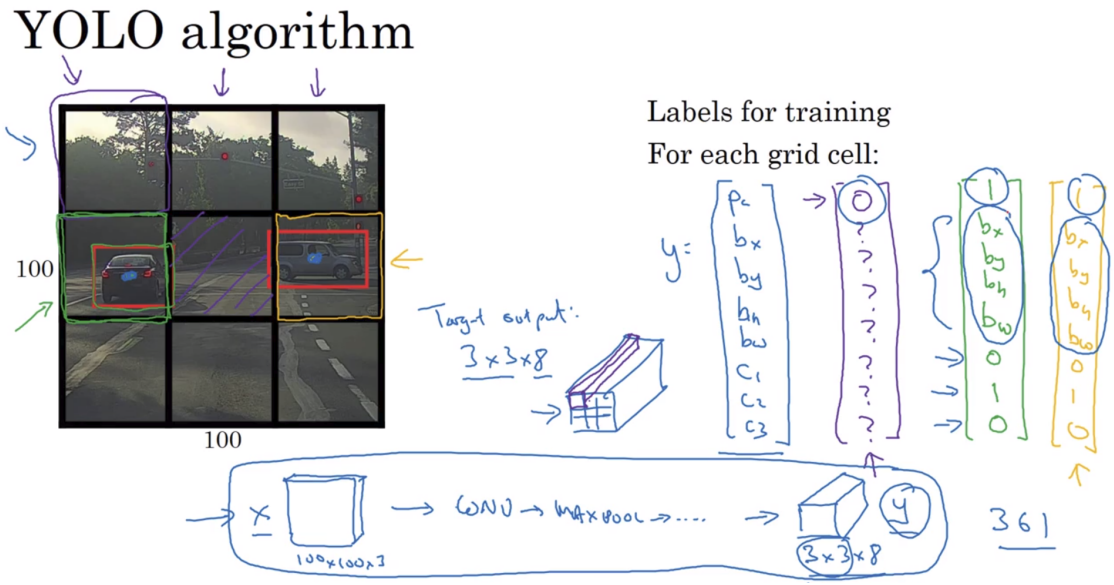


Figure 5.51: An example of input and output of YOLO algorithm using CNN. We can see that the output is basically a vector for each window and it consist of X and Y position of a centre of **found bounding box** (in values between 0 and 1), along with its height and width (these can be bigger than 1) = all these values are **relative to a given window**. If no object was detected in a given window, the first value in the vector is 0 and all other values are irelevant.

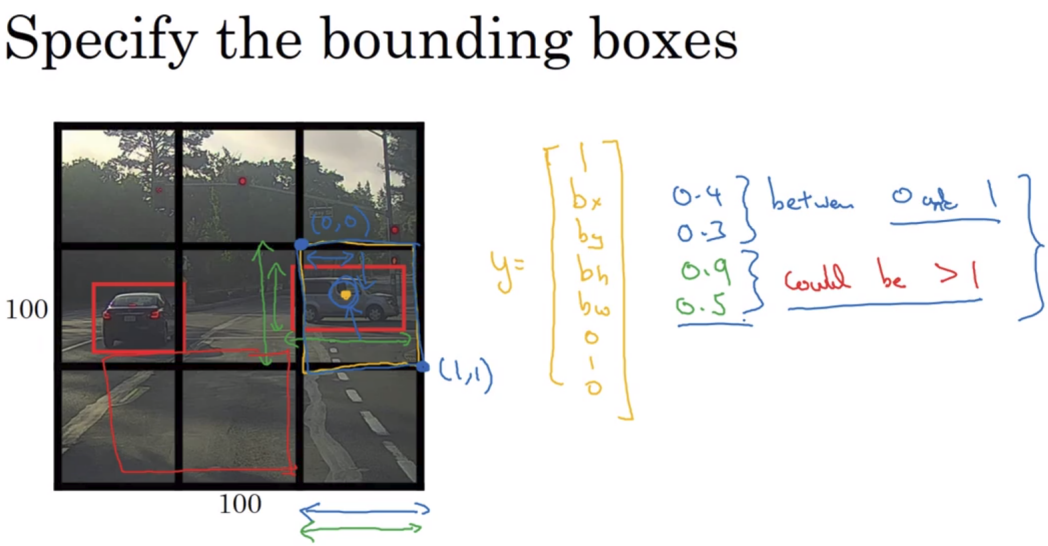


Figure 5.52: An example of found bounding boxes description in a vector. Further explanation was in the previous figure.

– An example of flow of YOLO:

1. Input image $(608, 608, 3)$
2. The input image goes through a CNN, resulting in a $(19, 19, 5, 85)$ dimensional output.
3. After flattening the last two dimensions, the output is a volume of shape $(19, 19, 425)$:
   a) Each cell in a 19x19 grid over the input image gives 425 numbers.
   b) $425 = 5$ x 85 because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes.
   c) $85 = 5 + 80$ where 5 is because $(p_c, b_x, b_y, b_h, b_w)$ has 5 numbers, and and 80 is the number of classes we'd like to detect.
4. You then select only few boxes based on:
   a) **Score-thresholding**: throw away boxes that have detected a class with a score less than the threshold.
   b) **Non-max suppression**: Compute the **Intersection over Union** and avoid selecting overlapping boxes (learn how to ensure the object detection algorithm detects each object only once). In general, this IoU measures a degree of overlap between two bounding boxes - how similar they are.
5. This gives you YOLO's final output.

- There can be a situation, when multiple detections (bounding boxes) will be found for the same object. This solves **non-max suppression** method - only the most probable classifications will be outputted, and all other will be suppressed. The algorithm starts by suppressing all bounding boxes below some threshold (0.6 for instance). Then, for all remaining boxes, pick one with the largest probability (save this as a prediction) and discard any remaining box with IoU bigger equal to 0.5 with such predicted box.

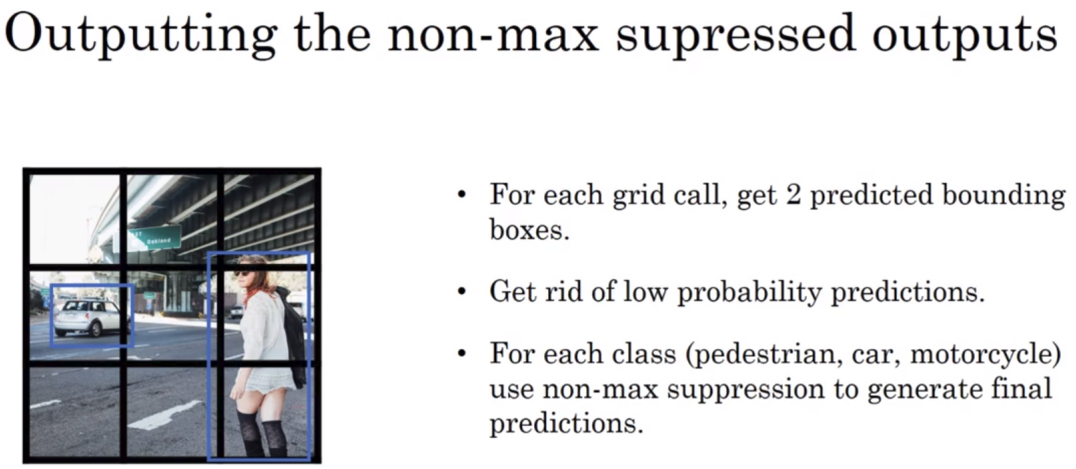


Figure 5.53: Non-max supressed outputs pseudo-algorithm.

  - There can be a situation, when multiple objects overlaps (partially) - solution is to use **anchor boxes** (predefined two shapes called "Anchor Boxes" and for each, associate multiple predictions - we use 5 or more anchor boxes in general; for example, one anchor box for pedestrian, another one for car; and they are associated with either the car or pedestrian based on how similar their shape is). As a much more advanced version (in comparison to manually set 5-10 anchor box shapes - that cover the type of objects we want to detect, which is also possible), is to use a K-means algorithm (this was suggested in YOLO paper) to group together two types of object shapes you want to detect. Then, use that to select a set of anchor boxes that are the most representative.
  - At training time, only one cell - the one containing the center/midpoint of an object - is responsible for detecting this object.

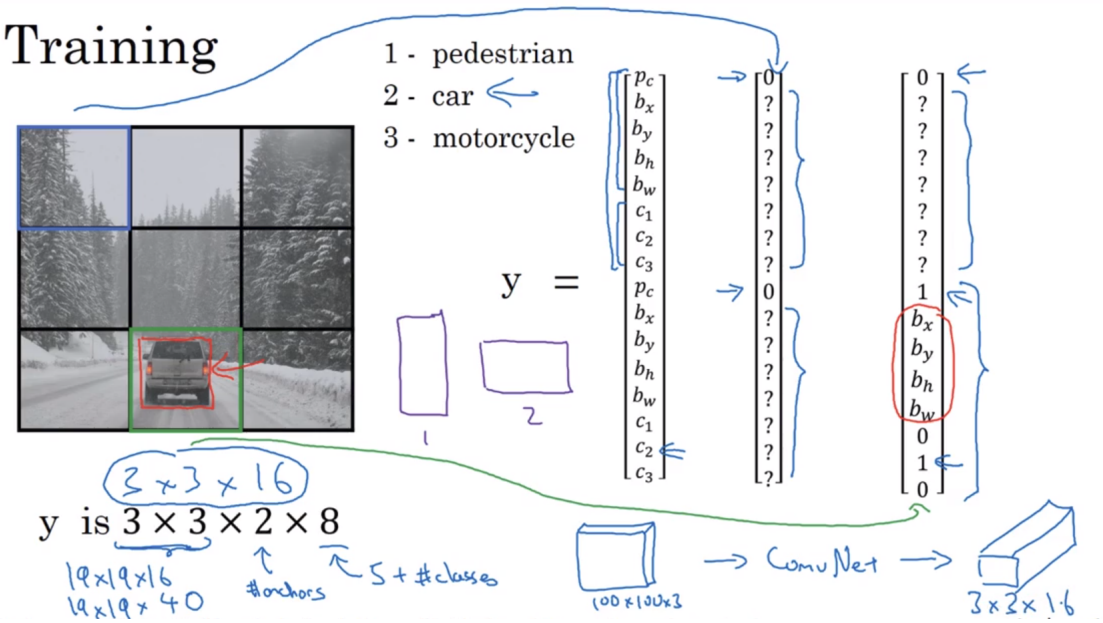


Figure 5.54: A brief training explanation with dimensions of input and anchors.

- **Region proposals**
  - Very influential in computer vision. Motivation - some traditional and popular CV algorithms tends to search a lot of regions where there is clearly no object. There is nothing interesting to classify.
  - **R-CNN** is a solution for such optimization (Girshik et al, from 2013). It tries to pick only a few regions that make sense to run a classifier later on. The way it searches for these regions is by using segmentation algorithm (it searches for a blobs and then places bounding boxes around them). So, in a nutshell, propose regions, classify them, one at a time; output label + bounding box (this bounding box is a new one,determined by running the classifier on the proposed region bounding box and finding an object). It turned out that R-CNN is still slow.
  - **Fast R-CNN** is an improvement to R-CNN (Girshik, from 2015) that uses convolution implementation of sliding windows to classify all the proposed regions. However, it also turned out to be quite slow.
  - **Faster R-CNN** is another improvement to region proposals (Ren et al, from 2016). This one uses the whole convolutional network to propose regions instead of segmentation algorithms (Region Proposal Network). However, Faster R-CNN implementations are usually still slower than YOLO algorithm. In the previous Fast R-CNN and R-CNN, region proposals are generated by selective search rather than using convolutional neural network. Basically:[6]

[6] https://www.quora.com/How-does-the-region-proposal-network-RPN-in-Faster-R-CNN-work

1. In the first step, the input image goes through a CNN which outputs a set of convolutional feature maps on the last convolutional layer.
2. Then a sliding window is run spatially on these feature maps. The size of sliding window is nxn (for example 3x3). For each sliding window, a set of 9 anchors (9 by default and these are region boxes, only those will be proposed that the most likely contain some objects) are generated (because we have 3x3 sliding window size) which all have the same center, but with 3 different aspect ratios and 3 different scales (in order to accommodate different types of objects, elongated objects like buses, for example, cannot be properly represented by a square bounding box). All these coordinates are computed with respect to the original image. Furthermore, for each of these anchors, value $p^*$is computed which indicates how much these anchors overlap with the ground truth bounding boxes:

$$p^* = \begin{cases} 1 & if\ IoU > 0.7 \\ -1 & if\ IoU < 0.3 \\ 0 & otherwise \end{cases} \tag{5.7}$$

   where IoU is intersection over union of anchor and ground truth box.
3. Filter out anchors - keep only top anchors, non-maximum suppression algorithm). Finally, 3x3 spatial features extracted from these convolution feature maps are fed into a small network which has 2 tasks: a) classification, and b) regression. The output of regressor determines a predicted bounding box (x, y, w, h), and the output of classification subnet is a probability p indicating whether the predicted box contains an object (1) or it is from background (0 for no object). The loss function is defined over output of both subnets, with 2 terms and balancing factor λ.

Some popular CNN architectures

This section provides a list and a brief description of some well-known architectures of CNNs. All were designed for image recognition etc problems.

LeNet-5

- When this architecture was designed (LeCun et all, 1998), people used usually average pooling, and no padding (so always valid convolutions).
- This CNN was used for recognition of digits. However, the output layer does not have softmax (it has something different, not relevant these days).

and https://medium.com/@smallfishbigsea/faster-r-cnn-explained-864d4fb7e3f8

- It has about 60,000 parameters, so this CNN is relatively small. It is mostly historical matter. Let's say in about 2017, it is common to have 10M up to 100M of parameters.
- What is still trendy is, that as you go more deep, dimensions (height and width) are smaller, but number of channels increases.

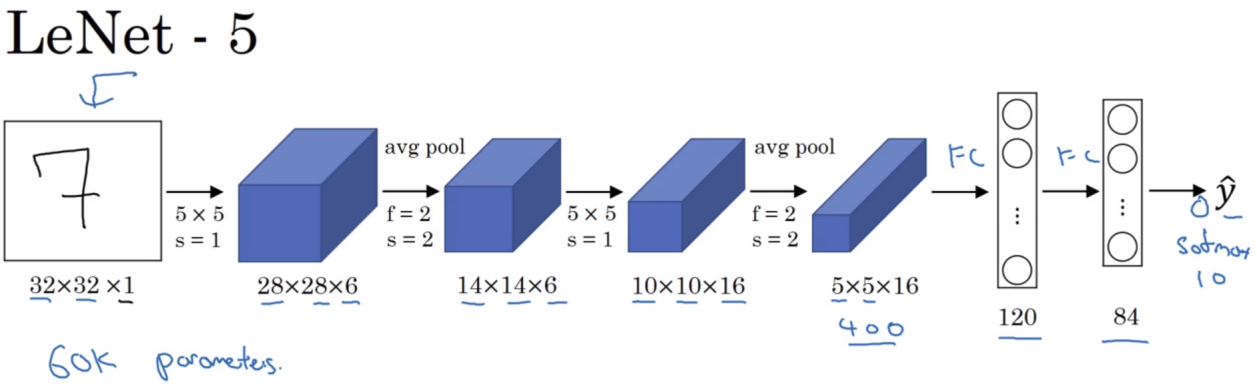


Figure 5.55: Architecture of LeNet-5 with dimensions of each layer.

AlexNet

- **This paper convinced a lot of researchers in CV to take a serious look at deep learning (Krizhevsky et al, 2012). This paper was very significant.**
- AlexNet has a lot of similarities with LeNet-5, but it is much bigger (and uses max-pooling as well as softmax).
- It has around 60M parameters.
- It uses ReLU activation functions.
- Here the author proposed LRN layer - **Local Response Normalization** layer, that is not used very much in present. It basically goes through all the channels (in a filter) and normalize these numbers. However, many researchers found that this does not help that much.

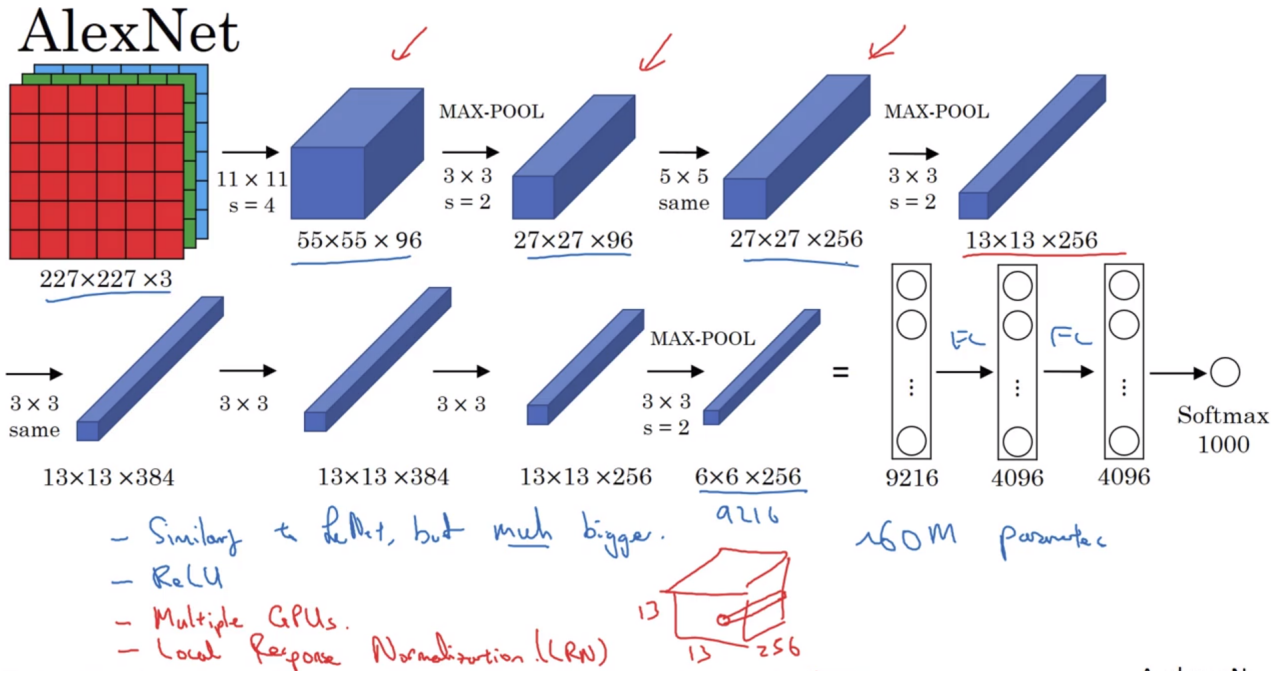


Figure 5.56: Architecture of AlexNet with dimensions of each layer.

VGG-16

- The authors (Simonyan and Zisserman, 2015) in comparison to the previous architecture, proposed much simpler network (=convolutions are simple, pooling and so on - no need to have so much hyperparameters). But it is very large - it is very deep network with about 138M of parameters. The network has 16 layers.
- Number of filters is always a double (64 to 128, 256 and then 512) and dimension of a layer goes down by factor 2.
- There is another version, VGG-19, that is even bigger version of this one. Both have similar performance.

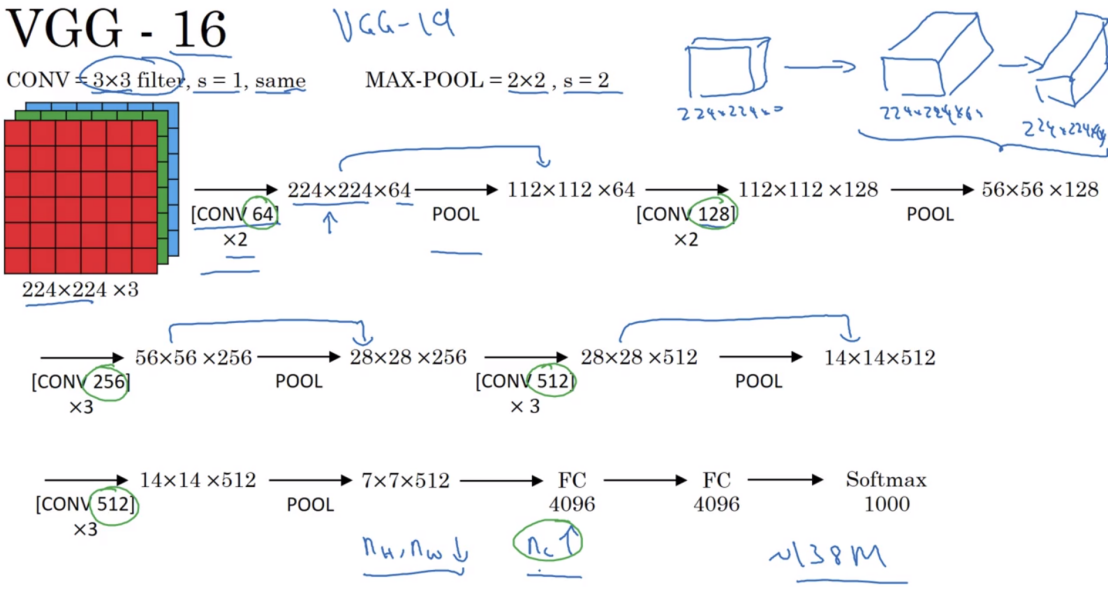


Figure 5.57: Architecture of VGG-16 with dimensions of each layer.

ResNet

- Also known as Residual Networks (He et al, 2015). Basically, you can make these neural networks deeper and deeper without really hurting your ability to at least get them to do well on the training set. And hopefully, doing well on the training set is usually a prerequisite to doing well on your hold out set, dev set, or test set.
- Very deep "plain" networks don't work in practice because they are hard to train due to vanishing gradients. The skip-connections help to address the Vanishing Gradient problem. They also make it easy for a ResNet block to learn an identity function.
- It uses **skip connections** which allows you to take the activation from one layer and suddenly feed it to another layer even much deeper in the neural network[7](if a dimension of such 2 layers don't fit, padding can be used). And using that, it is possible to train very, very deep networks, sometimes over 100 layers (which is one of the best advantage of this).
- **Residual block** - a basic block they are built from, they are using skip connections as can be seen in the next figure. Information is passed into a deeper block.

[7]When there is no skipping, just a sequence of layers and direct connection between each consecutive two, it is called 'plain network' - from original paper of ResNet.

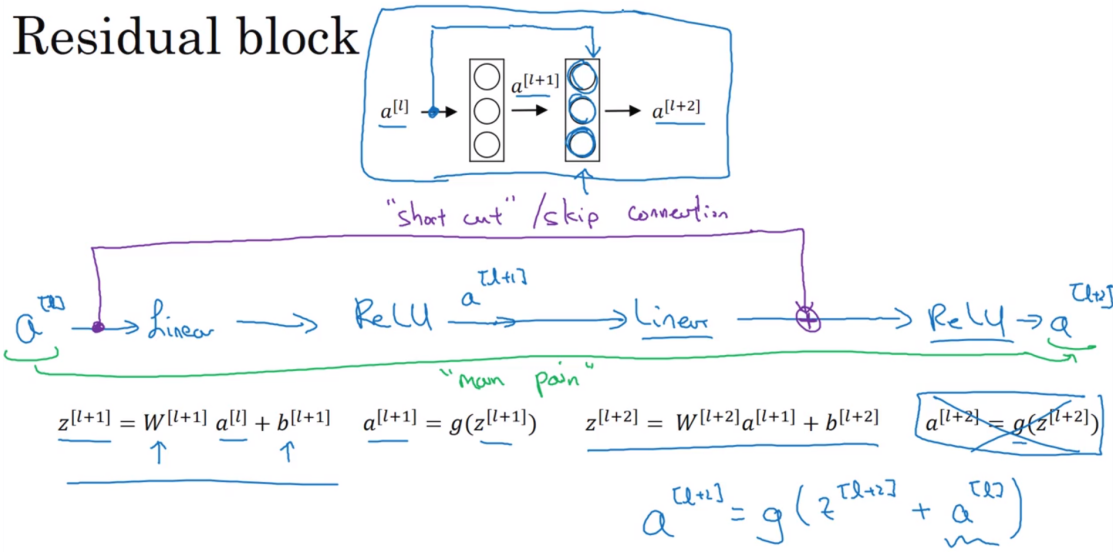


Figure 5.58: Residual block scheme in ResNet.

- There are 2 main types of blocks used in ResNet, very deep Residual Networks are built by stacking these blocks together.:
  - **The identity block** - standard block which corresponds to the case where the input activation (say $a^{[l]}$) has the same dimension as the output activation (say $a^{[l+2]}$).

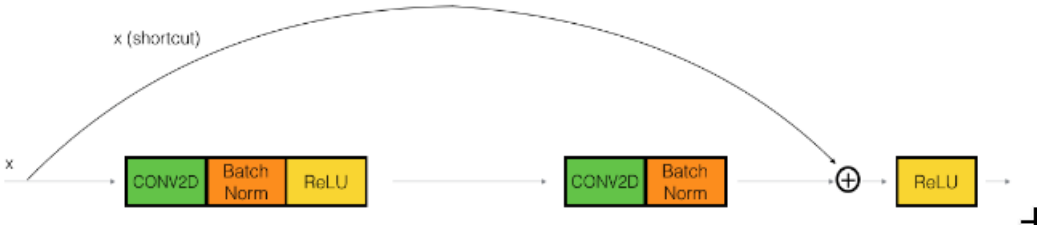


Figure 5.59: Identity block. Skip connection "skips over" 2 layers. The upper path is the "shortcut path." The lower path is the "main path." In this diagram, we have also made explicit the CONV2D and ReLU steps in each layer. To speed up training there is alos a BatchNorm step.

  - **The convolutional block** - you can use this type of block when the input and output dimensions don't match up. The difference with the identity block is that there is a CONV2D layer in the shortcut path.

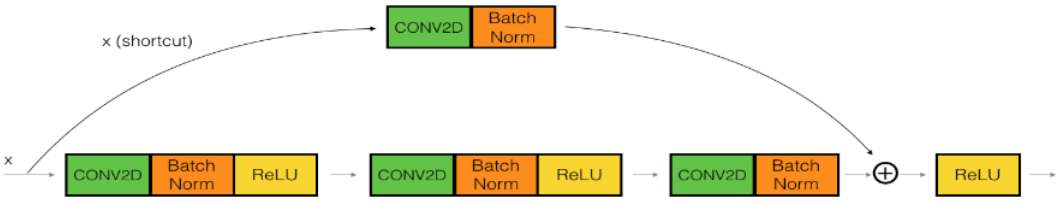


Figure 5.60: Convolutional block. The CONV2D layer in the shortcut path is used to resize the input xx to a different dimension, so that the dimensions match up in the final addition needed to add the shortcut value back to the main path. For example, to reduce the activation dimensions's height and width by a factor of 2, you can use a 1x1 convolution with a stride of 2. The CONV2D layer on the shortcut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.

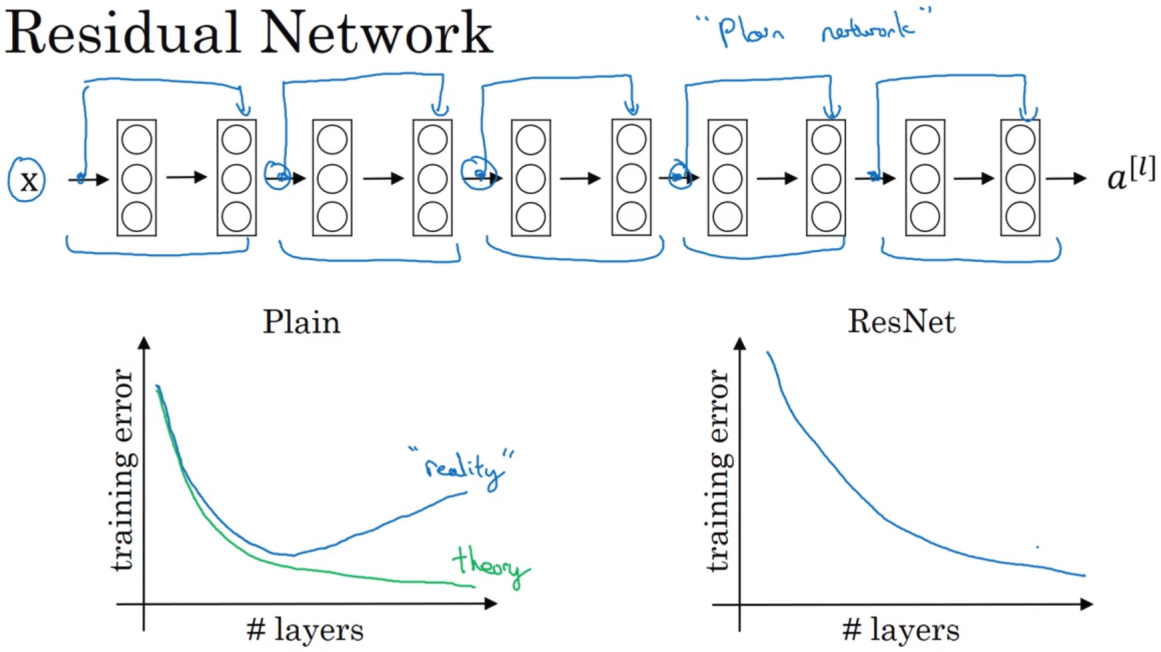


Figure 5.61: Residual network architecture and a comparison of learning plots of plain and ResNet. We can see, that plain NN with adding hidden layers the training error will tend to decrease after a while but then they'll tend to go back up (in theory, as you make ANN deeper, it should only do better and better on the training set. But in reality, having deeper and deeper plain network means that your optimization algorithm just has a much harder time training. So your training error gets worse if you pick a network that's too deep. But what happens with ResNet is that even as the number of layers gets deeper, you can have the performance of the training error kind of keep on going down. Even if we train a network with over a hundred layers.

- When we calculate $a^{[layer+2]} = g(z^{[layer+2]} + a^{[layer]}$, we are assuming that $z^{[layer+2]}$and $a^{[layer]}$ have the same dimensions. This imposes the use of "same convolution". If they does not have the same dimensions, then we will use a helping operation to make dimensions the same: $a^{[layer+2]} = g(z^{[layer+2]} + W_s a^{[layer]}$, where $a^{[layer+2]}$has 256 dimensions and $a^{[layer]}$has 128, then W_s will have 256x128 dimensions (multiplication) and this matrix can be either learned or fixed.

Inception Network

- Previous architectures - you had to pick what size of filter and what pooling layer (or without pooling) you want. Inception network (Szegedy et al, 2014) combines them all - the network learn what works best. It is very complicated ANN architecture, see the next figure.

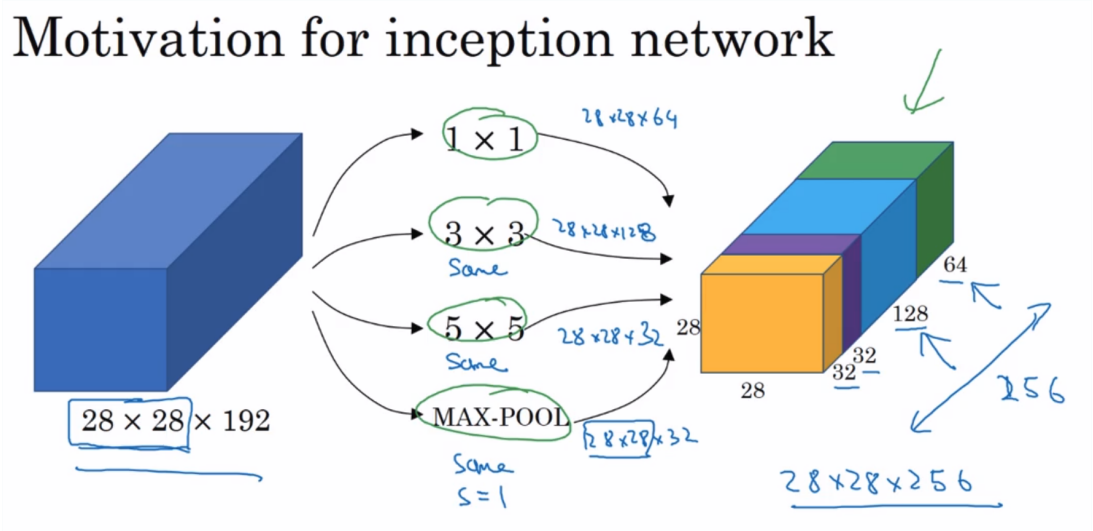


Figure 5.62: Inception Network motivation. We can have multiple (in this example 'same') convolutions (each with different size of filters) at the same time and stack the result along with max-pooling (with padding to have the same dimensions as an input). Results have different number of channels, but the same heights and widths. And then, a network will find out whatever the combinations and sizes are good to have.

- The problem of having such layer, is a very big computational cost. Therefore, it is possible to do an "optimization" - first, let's decrease a number of channels with 1x1 convolution (n of them, depending on the resulting channel dimension), which is sometimes called as **bottleneck layer** (and this won't hurt the performance of ANN very much). Then we can increase the size again, but the part after 1x1 convolution is the smaller one (therefore a bottleneck). So this 1x1 convolution is here only for reducing the computational cost.

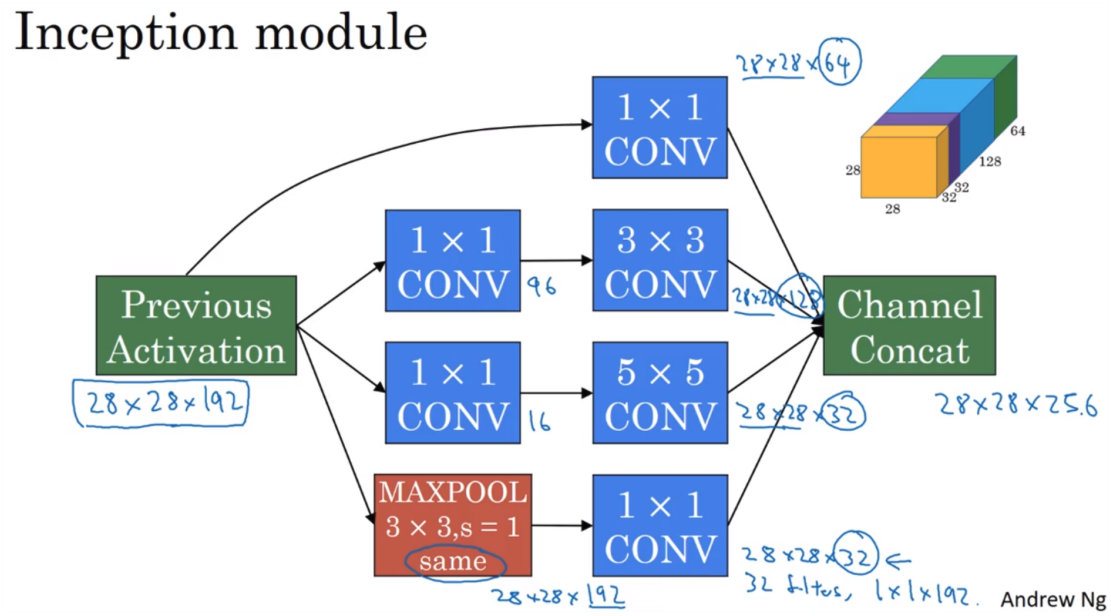


Figure 5.63: An example of Inception module. These 1x1 convolutions are there for reducing computational cost. There can be 10 or more of such modules in Inception network.

- Inception network consists of **inception modules** (as the previous figure), and there are also **side branches**. These side branches help to ensure that the features computed even in the hidden units, even at intermediate layers, are not too bad for predicting the output cause of a image. And this appears to have a regularizing effect on the inception network and helps prevent this network from overfitting.
- Making an inception network deeper (by stacking more inception blocks together) could hurt training set performance.
- Originally developed at Google, and called GoogleNet. But Inception network is named after the movie The Inception.
- There are newer versions of Inception network, for example there is a version that combines ResNet idea (skipping layers) and Inception network.

Siamese Network

- This is widely used in face recognition problem, **for one-shot encoding (for building a similarity function for calculation of a distance between 2 images if there is a match, see also Subsection 1.4)**. The idea is from Taigman et al, DeepFace, from 2014.

- Siamese network can be implemented as any kind of neural network (CNN, RNN, MLP). The network only takes one image as input at a time (so the size of the network is not doubled).
- This is (mostly) kind of CNN, that learns representation of input (image) in a vector of fixed size. And this resulting vector is sometimes said that it is **encoded input** (because this CNN is computing basically a function, that encodes its input). Then, when we have 2 images, so 2 vectors, from the same CNN, we can compute their **similarity** with $||f(x_1) - f(x_2)||^2$ - **L2 norm**, to determine a **distance between two vectors**. If the result is small, $x_1$and $x_2$is the same person.
- Running 2 different inputs through two identical convolutional neural nets, and comparing the outputs is called Siamese neural network architecture. Goal of training is that we want to train a neural net so that its encoded output can be used in a function that indicates when the 2 input pictures are of the same person.
- In Siamese networks, we take an input image of a person and find out the encodings of that image, then, we take the same network without performing any updates on weights or biases and input an image of a different person and again predict it's encodings. Now, we compare these two encodings to check whether there is a similarity between the two images. These two encodings act as a latent feature representation of the images. Images with the same person have similar features/encodings. Using this, we compare and tell if the two images have the same person or not.[8]
- Objective function they use is called **triplet Loss** (Schroff et al, FaceNet, from 2015).
  - It is a one way to learn the parameters of neural network that gives you a good encoding for your pictures of faces. Just apply gradient descent on the triplet loss function.
  - In Triplet Loss, you always look at 3 images: Anchor (A), Positive (P), and Negative (N) examples. So we want $||f(A) - f(P)||^2$ to be small, to be more particular: $||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2$.
  - We can rewrite the previous formula as $||f(A)-f(P)||^2-||f(A)-f(N)||^2 \leq 0$. To make sure that this is satisfied, both formulas can be set to 0, so $0-0 \leq 0$. So make sure, that our NN will not output always zeroes, or always the same values, we need a modification - we say, that this equation does not need to be lower equal to zero, but it needs to be quite a bit smaller than zero. We add (on the right side) a hyperparameter $-\alpha$ (also called a margin), so that our neural network will not find trivial solutions (zeroes or same values). If

[8] https://towardsdatascience.com/siamese-network-triplet-loss-b4ca82c1aec8

we move α to the left side, we have eventually:

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0 \tag{5.8}$$

- Triplet loss function is then defined on 3 images A, P, and N. We want the previous formula to be always lower than 0, so:

$$L(A, P, N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0) \tag{5.9}$$

- So, if you have a training set of 10k images of 1k people, you have to split them into 3 sets (A, N, and P). Traditional CNNs will not work, as each person has only 10 images, and CNN needs far more data to learn features required to sucessfully beat this task. So, you can notice that for 1 person, it is needed to have multiple images (at least 2: Anchor and Positive example) for training. After the training, you can apply a given model to one-shot learning problem, where you can have just a single example of a person you want to recognize.
- Choosing these triplets - cannot be chosen randomly, because $||f(A)-f(P)||^2+\alpha \leq ||f(A)-f(N)||^2$ is easily satisfied, because chances that A and P are the same person are much lower than A and N is different person, and ANN will not learn much from it. It is needed to chose triplets that are “hard” to train on (both distances are relatively close to each other).
- By using Siamese networks, it is possible to deal with Face recognition problem as with binary classification problem (see the next figure). This is an alternative to using triplet loss function.
- The triplet loss function tries to "push" the encodings of two images of the same person (Anchor and Positive) closer together, while "pulling" the encodings of two images of different persons (Anchor, Negative) further apart.
- The triplet loss is an effective loss function for training a neural network to learn an encoding of a face image. The same encoding can be used for verification and recognition. Measuring distances between two images' encodings allows you to determine whether they are pictures of the same person.

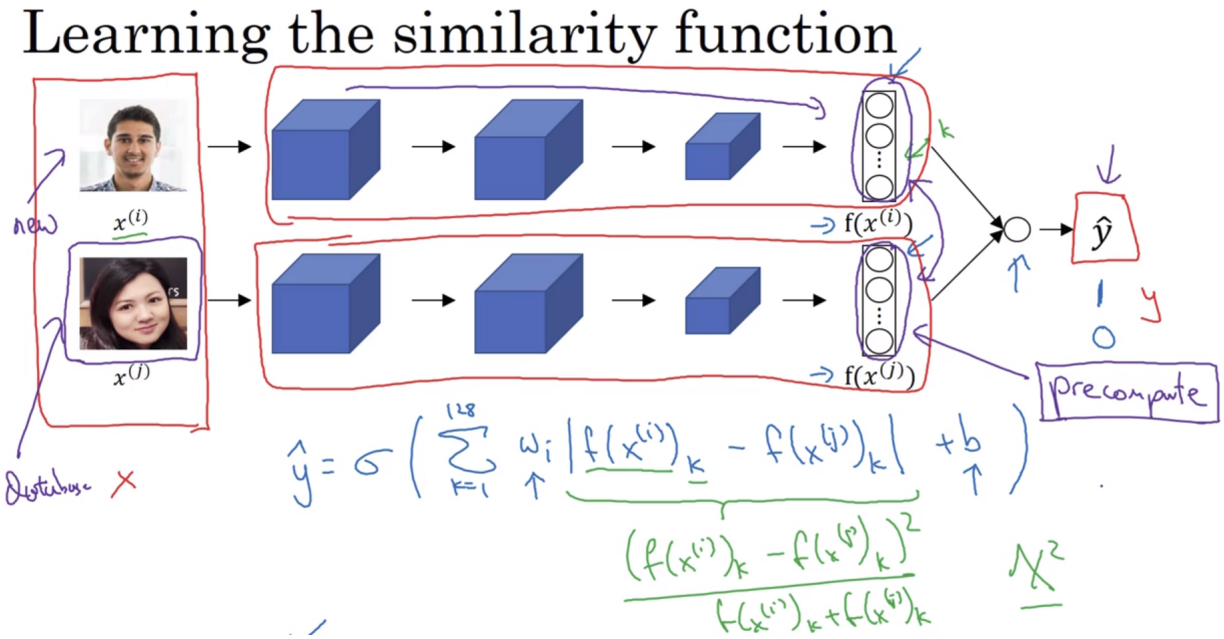


Figure 5.64: Learning a similarity between 2 images by using Siamese Network as a binary classification task (an alternative to triplet loss). Both CNNs are computing vectors $f(x^{(i)})$ and $f(x^{(j)})$ (each is maybe 128 dimensional or even higher) and both networks have the same (or really tight) parameters. The output $\hat{y}$ can be a sigmoid function that takes an element-wise difference between these 2 encodings (results are basically 'features'). It may have also weights and bias, as a traditional Logistic regression, so that the model will train these. Green text is an alternative formula sometimes called chi-squared. Purple vector is basically a computational trick, when we precompute that vector from data in our database. When a new person will come on the input, the upper ConvNet will compute its encoding (vector) and then this one is compared to precomputed encoding. That information will be used for making a prediction.

  - So, we need multiple pictures of the same person to train the network. But, after training, we don't need to have multiple images of the same person to test the face recognition classifier.
  - To build an SNN, we first decide on the architecture of our neural network. Given an example, to calculate the average triplet loss, we apply consecutively, the model to anchor, positive, and then to negative image, and then we compute the loss for that example. We repeat this for all triplets in the batch and then compute the cost. Geradient descent with backpropagation will propagate the cost through the network to update its parameters.

- There is an alternative to triplet loss - binary classification. Train a Siamese net-

work so that we got just 2 input images. Each is going through the different, but identical convolutional network, that computes output vector (image encodings). Such encoding output from both are feed to logistic regression node, which outputs 1 if the input image is the same person, or 0 if these people are different. Computational trick - precompute the encodings for all current entries in the database; just new entries will be computed (their encoding) at test time. So we don't even need to store images. This precomputing trick can be used for triple loss classifiers as well.

5.8 Stacked Auto-Encoders*

5.9 Generative Adversarial Networks

- This is a class of neural networks used in unsupervised learning. They are implemented as a system of two neural networks contesting with each other.
- The most popular application of GANs is to learn to generate photographs that look authentic to humans.
- They consist of two parts:
  - **the discriminator** is a CNN that is trained to recognize images. This networks takes as input two images: one "real" from some collection of images, and the image generated by the generator network. It has to learn to recognize which one of the two images was generated by the generator. It gets penalized if it fails to recognize which one of the two images is fake.
  - **the generator** is an inverse network that take a random seed and uses it to generate images. This networks takes a random input (typically a Gaussian noise) and learns to generate an image as a matrix of pixels. This network gets a negative loss if the discriminator network recognizes the fake image.

  The discriminator evaluates the output of the generator and sends signals to the generator on how to improve, and the generator in turn sends signals to the discriminator to improve its accuracy as well, going back and forth in a zero-sum game till they both converge to best quality.

5.10 Reinforcement Learning

- RL solves a very specific kind of problem where the decision making is sequential.
- Usually, there is an agent acting in an unknown environment. Each action brings a reward and moves the agent to another state of the environment (usually as a result of some random process with unknown properties). The goal of the agent is to optimize its long-term reward.
- RL algorithms, such as **Q-learning**, are used in learning to play video games, robotic navigation and coordination, inventory and supply chain management, optimization of complex electric power systems, or learning financial trading strategies.

6 Instance-based Learning

- **These algorithms are based on similarity of new case** (new, previously unseen sample) **with known** (already seen) **records**.
- **These methods are simple, but they can approximate complex function with qualitative or quantitative output value.**
- **These are lazy-learning algorithms, as they do not create any model** and delay the induction or generalization process until classification is performed. Calculation for prediction of output is performed in a moment when a new sample is given to a model. Therefore, these methods can approximate well relations which are inside space of training data. On the other hand, prediction of data outside of training data can work not so well.
- **Lazy-learning algorithms require less computation time during the training phase than eager-learning algorithms** (such as decision trees or neural nets) **but more computation time during the classification process.**
- Decision problem with instances or examples of training data that are deemed important or required to the model. Such methods typically build up a database of example data and compare new data to the database using a **similarity measure** in order to find the best match and make a prediction.
- **Pros**
  - There are normally no parameters to tune, the system is normally hard coded with priors in form of fixed weights or some algorithms like tree search based algorithms. No need to know anything about model. Such system normally does what is known as lazy learning by absorbing the training data instances and using those data instances for inference.
  - Incremental character, there is (almost) no generalization, we can add new data and immediately work with them. We can even add a new class. This is also considered as disadvantage.
  - Lazy learning.
  - Prediction of both qualitative and quantitative values.
  - Records are usually saved in k-d trees, so finding out a neighbor may have complexity $log_2 N$, where N is a number of saved instances.

- **Cons**[1]
  - They are computationally expensive since they save all training instances = the more data we have (saved, "training" data), the slower prediction time the algorithm will have. This is the major disadvantage.
  - These methods are usually sensitive to skewed class, because class with smaller amount of samples can be outvoted.
  - These methods are very sensitive to attributes, which are not relevant.
  - Gained knowledge is not understandable and (easily) interpretable by human.
  - The performance depends on the choice of the similarity function to compute distance between two instances.
  - There is no simple or natural way to work with nominal valued attributes or missing attributes.
  - They do not tell us much about how the data is structured.
- **IBL methods** - (mostly) supervised iterative algorithms (however, k-NN is basically unsupervised), they describe a way how to save only relevant items.
  - **IB1** - saves all training samples.
  - **IB2** - saves only incorrectly categorized samples. Very sensitive to noise and lower memory requirements.
  - **IB3** - extended by statistical tests which can detect noise. New samples can be accepted, rejected or monitored. On classification, only the accepted samples are included.
  - **IB4** - extended by possibility of give some weights to the input attributes. So this method does not suppose that each attribute has same relevance. This is suitable if there is a big amount of attributes.

6.1 k-Nearest Neighbors

- **Supervised learning algorithm** for **classification** and **regression**.
- Non-parametric method - it does not make any underlying assumptions about the distribution of data.
- Contrary to other learning algorithms, that allow discarding the training data after the model is built, kNN keeps all training examples in memory. Once a new, previously unseen example comes in, the algorithm finds k training examples closest to it and returns the majority label (classification), or the average label (regression).

[1] http://cs.uccs.edu/~jkalita/work/cs586/2013/InstanceBasedLearning.pdf

- The output: classification - class membership, regression - average of the values of its k nearest neighbors.
- Sensitive to the local structure of data.
- **For continuous variable - Euclidean distance, for discrete - Hamming distance.** Another popular distance function is the **negative cosine similarity** (and there are also other ones, such as Chebychev distance, or Mahalanobis distance):

$$s(x, y) = \frac{\sum_{j=1}^{D} x^{(j)} y^{(j)}}{\sqrt{\sum_{j=1}^{D} (x^{(j)})^2} \sqrt{\sum_{j=1}^{D} (y^{(j)})^2}} \tag{6.1}$$

  - If the angle between two vectors is 0 degrees, then two vectors point to the same direction, and cosine similarity is equal to 1. If the vectors are orthogonal, the cosine similarity is 0. If the vectors are in opposite directions, then it is -1.
- Hard decision or soft score - hard is voting, soft is based on some threshold value.
- Before use, it needs data to be normalized and all irrelevant and redundant attributes to be removed.
- k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification.
- It is not robust to missing values, KNN requires complete records to do their work.
- **k-NN needs:**
  - sufficient amount of training data.
  - normalization of input values.
  - removal of non-relevant or redundant values.

Algorithm 6.1 K-nearest neighbors algorithm

k-NN (dataset, sample) {

1. Go through each item in my dataset, and calculate Euclidean distance from that data item to my specific sample. Sort samples and select only the first K items.
2. Classify the sample as the majority class between K samples in the dataset having minimum distance to the sample.

}

6.2 Self-Organizing Map

- Unsupervised learning, for dimensionality reduction. It also belongs to clustering (see Section 8.5).
- Also known as Self-organizing feature map (SOFM) or Kohonen map.
- It is based on neural network, where each neuron has vector of weights and a position in map.
- Self-organizing maps differ from other artificial neural networks as they apply competitive learning as opposed to error-correction learning (such as backpropagation with gradient descent).
- Good for visualization of n-dimensional space in 2D or 3D.
- **Learning:**
  - Neurons compete between each other, which one will be the closest to test sample from dataset (distance of sample and each vectors of weights).
  - Winner (or his closes neighbors) modifies its weights (make a step closer to test sample) by learning rate (which is later lower and lower).
  - Neighbor function - determines the surroundings of winner, in which the weights will be modified. It is smaller and smaller with the time.
  - The end of learning - the algorithm reached a maximum number of iterations, or the changes in weight vectors are very little.
  - The algorithm requires sufficient amount of training data.

6.3 Learning Vector Quantization*

6.4 Locally Weighted Learning*

7 Decision Trees

- A decision tree is an acyclic graph that can be used to make decisions. In each branching node of the graph, a specific feature j of the feature vector is examined. If the value of the feature is below a specific threshold, then the left branch is followed, otherwise the right branch is followed. As the leaf node is reached, the decision is made about the class to which the example belongs. A decision tree can be learned from data.
- They are basically a hierarchical non-linear models. They primary use is for classification of qualitative values based on input attributes.
- They main advantage is (except non-linearity) also a very good interpretability, flexibility, and existence of algorithms which are able to create those trees automatically.
- Each node in a decision tree represents a feature in an instance to be classified, and each branch represents a value that the node can assume. Instances are classified starting at the root node and sorted based on their feature values.
- The feature that best divides the training data would be the root node of the tree. There are numerous methods for finding the feature that best divides the training data such as information gain and gini index.
- They construct a model of decisions made based on actual values of attributes in the data. Decisions fork in tree structures until a prediction decision is made for a given record.
- Decision trees are trained on data and can be used for both classification and regression. They are built by recursively partitioning of the input space to distinct non-overlapping regions (nodes in the tree). Leaves of the tree contain the final decision about the predicted class (in case of classification). The purpose is to find such tree, that best separates data into given classes. Finding an optimal partition is an NP-complete problem.
- They can suffer from high variance: decision trees learned on different training sets generated by the same phenomenon are often very different, when in fact they should be the same.
- The problem of constructing optimal binary decision trees is an NP-complete problem and thus theoreticians have searched for efficient heuristics for constructing near-optimal decision trees.

- Decision trees tend to perform better when dealing with discrete or categorical features.
- Overfitting:
  - The most straightforward way of tackling overfitting is to pre-prune the decision tree by not allowing it to grow to its full size. There is no single best pruning method.
- **Pseudo-algorithm**
  1. Do in a loop (there is usually some termination condition, which determines when the algorithm stops and it is very important since it influences overfitting):
     a) Get information about a node.
     b) Decide if a node will be split (step c), or if a node will be changed to a list (if so, determine its output value).
     c) Choose the best attribute (feature) for splitting.
     d) Split data into new nodes.
  2. Perform tree pruning (something like regularization, this technique reduces too complex, over-fitted trees).
- **Termination condition**

  All values can be set and they depends on a given problem and data:
  - Maximum number of records in a given node reached, no splitting is allowed on a given node (this can be set, for instance to10% as a minimum records in node for splitting).
  - Maximum depth of a tree reached, do not create more and more nodes (this can be set, for instance to5).
  - One class is has a given node majority of records (this can be set, for instance to90%).
- **Reduced-Error Pruning**
  - Splits data to training (2/3) and validation (1/3).
  - **Pseudo-algorithm**
    1. Train a tree on all training data (over-fit).
    2. Repeat on validation data, until a new tree is created.
       a) Try to substitute all nodes in tree with a list (according to validation data) and evaluate all new trees.
       b) Among all the new trees, pick the best one. If this tree is better than a tree chosen last time, then choose the new one, otherwise keep the old one as the best option.

- **Rule Post-Pruning**
  - Final model is more general.
  - **Pseudo-algorithm**
    1. Transform over-fitted tree to rules.
    2. For each rule, try to remove all combination of conditions, estimate accuracy, and save the best condition.
    3. Sort and use rules according to accuracy.

7.1 Iterative Dichotomiser 3

- Input quantitative / **output qualitative (nominal).**
- Historically, before ID3 there was CLS. Successor of ID3 was C4.5.
- Topology depends on attribute. It generates tree in deterministic way. There is no guarantee to generate optimal tree.
- The optimization criterion is in this case the average log-likelihood: $\frac{1}{n}\sum_{i=1...N}[y_i \, ln \, f_{ID3}(x_i) + (1-y_i) \, ln \, f_{ID3}(x_i))]$ where f_{ID3} is a decision tree. It may look similar to **logistic regression**, which **builds a parametric model** $f_{w,b}$ by **finding an optimal solution to the optimization criterion**. **ID3** algorithm **optimizes the optimization criterion approximately** by constructing a **non-parametric model** $f_{ID3}(x) = P(y = 1|x)$.
- It chooses attribute with the lowest **conditional entropy** (so the **highest information gain**) - the algorithm tries to chose such attributes, that increases information gain of a tree as much as possible (by lowering total entropy). **The problem with information gain is that it favors attributes with many possible values. C4.5 solves this by introducing gain ratio, which normalizes the information gain by the number of feature values (instead of entropy, there is Gini index).**
- Entropy-based split criterion makes sense - entropy reaches its minimum of 0 when all examples in S have the same label. On the other hand, the entropy is at its maximum of 1 when exactly one-half of examples in S is labeled with 1, making such a leaf useless for classification.
- **The algorithm does not guarantee, that final tree is the best possible one.**
- **Each list represents classification into a single specific class.**
- Missing attribute values - it uses value which is more occurring in a given node, or/and most occurring in a given class.

- Reduced-Error Pruning - new trees are created from validation data and the best one is always chosen instead of previous one.
- Rule Post-Pruning - new model is more general, it converts over-fitted tree to rules and according to accuracy it chooses only certain rules and removes other ones.
- The algorithm is as follows:
  1. Let S denote a set of labeled examples. In the beginning, the decision tree has only a start node that contains all training examples. Start with a constant model defined as $f^S_{ID3} = \frac{1}{|S|}\sum_{(x,y)\in S} y$. The prediction given by the above model would be the same for any input x.
  2. Then we search through all features $j = 1, ..., D$ and all thresholds t, and split the set S into two subsets: $S_- = \{(x, y)|(x, y) \in S, x^{(j)} < t\}$ and $S_+ = \{(x, y)|(x, y) \in S, x^{(j)} \geq t\}$. These two new subsets would go to the new leaf nodes, and we evaluate for all possible pairs (j, t) **how good the split** (more on this later) with pieces $S_-$and $S_+$is. Finally, we pick the best such values (j, t), split S into $S_-$and S_+, form two new leaf nodes, and continue recursively on $S_-$and S_+(or quit if no split produces a model that is sufficiently better than the current one).
     - How good the split is - in ID3, the goodness of a split is estimated by using the criterion called **entropy**. It is a measure of uncertainty about a random variable. It reaches a maximum when all values of the random variables are equiprobable, and reaches minimum when the random variable can have only 1 value. The entropy of a set of examples S is given by $H(S) = -f^S_{ID3} ln\, f^S_{ID3} - (1 - f^S_{ID3}) ln(1 - f^S_{ID3})$.
     - When we split a set of examples by a certain feature j and a threshold t, the entropy of a split, $H(S_-, S_+)$ is simply a weighted sum of two entropies: $H(S_-, S_+) = \frac{|S_-|}{|S|}H(S_-) + \frac{|S_+|}{|S|}H(S_+)$. So, in ID3, at each step at each leaf node, we find a split that minimizes the entropy given by this equation, or we stop at this leaf node.
     - The algorithm stops at a leaf node in any of the situations:
       * All examples in the leaf node are classified correctly by the one-piece model (equation $f^S_{ID3} = \frac{1}{|S|}\sum_{(x,y)\in S} y$).
       * We cannot find an attribute to split upon.
       * The split reduces the entropy less than some ϵ - the value for which as to be found experimentally. This is a hyperparameter.
       * The tree reaches some maximum depth d (also hyperparameter).
     - The decision to split the dataset on each iteration is local (doesn't depend on future splits), the algorithm doesn't guarantee an optimal solution. The model can be improved by using techniques like backtracking during

the search for the optimal decision tree at the cost of possibly taking longer to build a model.

7.2 C4.5 and C5.0

- C4.5 is an extension to ID3 - it also uses entropy and information gain. What is different than ID3 is, that input values can be quantitative (so in these, features can be both continuous and discrete), missing attribute values are better handled, and it solves overfitting problem by using a pruning technique.
- Pruning consists of going back through the tree once it's been created and removing branches that don't contribute significantly enough to the error reduction by replacing them with leaf nodes.
- C5.0 has even more improvements than C4.5 and is protected by license.
- Input quantitative+qualitative / **output qualitative (nominal).**
- Topology depends on attribute.
- In node, quantitative attribute is converted to nominal (binary threshold or intervals).
- State of node S is determined by:
  - $|S|$, number of items in node$S$,
  - $H(S)$, entropy in node$S$,
  - $\{A\}$, of attributesA_j which can be used for splitting.
- Evaluation of quality of split in node$A$:
  - $|A|$, enumeration of potential attributes of$A$.
  - $I(S, A)$,information gain during splitting by$A$.

$$I(S, A) = H(S) - H(S|A) \tag{7.1}$$

    * or, with missing values (where$S_0$is number of items with missing value/feature$A$):

$$I(S, A) = \frac{|S - S_0|}{S}(H(S) - H(S|A)) \tag{7.2}$$

  - $P(S|A)$, ratio gain during splitting by$A$.

$$P(S|A) = \sum_{i=1}^{|A|} \frac{|S_i|}{|S|} log_2(\frac{|S_i|}{|S|}) \tag{7.3}$$

* or, with missing values (where S_0 is number of items with missing value/feature A):

$$P(S|A) = -\frac{|S_0|}{|S|} log_2(\frac{|S_0|}{|S|}) - \sum_{i=1}^{|A|} \frac{|S_i|}{|S|} log_2(\frac{|S_i|}{|S|}) \tag{7.4}$$

– $I_p(S, A)$, ratio information gain (aka gain ratio).

$$I_p(S, A) = \frac{I(S, A)}{P(S, A)} \tag{7.5}$$

- The algorithm stops in some leaf node in one of the following cases:
  – All the examples in the leaf node belong to the same class.
  – None of the features provide any information gain.
  – Additional stopping criteria as hyperparameters - is a tree is already deep enough, or a number of examples in a given leaf node is below a certain threshold.
- Division on node according to attribute with the **biggest ratial information gain** $I_p(S, A)$, but a node must have **at least average information gain** $I(S, A)$.
- C4.5 algorithm is trying to look for a split that maximizes the information gain.
- There is a built-in overfitting prevention mechanism. Once the tree is built, C4.5 replaces some branches (subtrees) by leaf nodes. Doing that the algorithm reduces variance (but increases bias). One possible way to **decide whether to keep a subtree or replace it with a leaf** is to apply the tree to the examples from the **validation set** and measure the **error made in different leaves**. If the **weighted sum of errors** made in the leaves of some branch is higher than the error that would have been made should the tree stop one level earlier, then the branch is replaced by the leaf.

7.3 Classification and Regression Tree

- Also known as CART.
- Input quantitative+qualitative /**output quantitative+qualitative.**
- Binary topology.
- Division on node according to only 1 attribute - this can cause, that a tree is not able to approximate a simple relation.
- An advantage is that there is a possibility of class weighting, and we can also create and use cost matrix (penalization for miss-classification) for penalization function.

- The algorithm uses GINI index (as information gain) for splitting.

$$GINI(X) = 1 - \sum_{\forall x \in X} p(x)^2 \tag{7.6}$$

7.4 Chi-squared Automatic Interaction Detection*

7.5 Conditional Decision Trees*

7.6 M5*

8 Clustering

- Clustering is a problem of learning to assign a label to examples by leveraging an unlabeled dataset. So, because of this, deciding on whether the learned model is optimal is much more complicated than in supervised learning.
- A set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.
- Applications: market segmentation, social network analysis, organize computer clusters, analysis of astronomical data.
- Clustering can be hierarchical (simple algorithm, repetitive runs produces the same solution, output is dendogram) or non-hierarchical (potentially different solution after each run).
- These methods have ability to process even high-dimensional data.
- A good clustering method creates clusters with:
  - great similarity of objects inside class (**high intra-class similarity**), and
  - low similarity of objects between classes (**low inter-class similarity**).
- Most of the algorithms are so-called **hard clustering** (such as **k-means** and **DBSCAN**), in which each example can belong to only one cluster. But there exist algorithms that allow each example to be a member of several clusters with different membership score, such as **Gaussian mixture model** (GMM) and **HDBSCAN**.

8.1 Methods Based on Division

- These (**centroid-based**) methods create a "database" of N objects which belong to k classes, $k \leq N$. Parameter k must be specified.
- Each class must have at least 1 object and each object belongs to just 1 class.
- **Representative methods:**
  - **k-means** - method based on central point (each class is represented by 1 such point, which is picked randomly). Sensitive to noise, can not find clusters of different size and non-convex shape.
  - **k-medoids** - method based on representative object (each class is represented by 1 such object, which is the closest to center of a given class). Computationally more expensive than previous method, and is effective if we have less data.
  - **CLARANS** (Clustering Algorithm based on Randomized Search).
- **Pseudo-algorithm:**
  1. Pick k objects, which will represent each individual clusters. Other samples will be divided to classes according to similarities.
  2. Cycle what ends when nothing was moved: new cluster centroids will be estimated and objects will be moved to clusters, based on distanced from a middle of clusters.

K-Means

- First step is to pseudo-randomly pick a number of clusters k. For example $k = 2$, so we randomly pick 2 points (so called **cluster centroids**) - so each runtime produces different solution. Two, because I want to divide data into 2 parts. It is not always easy to determine a number of clusters - elbow method, or manually - for example via visualizations. It should be lower than the number of examples in dataset.
- Non-hierarchical clustering algorithm.
- Input to the algorithm: K (number of clusters), and training set.
- Iterative algorithm

  (random initialization at 0 step - randomly pick K training examples. Set $\mu_1, ..., \mu_K$ equal to these K examples)

  1. **Cluster assignment step** - assign one of the clusters to all data points. So it goes from all data samples, and all clusters, and chooses a cluster with the smallest distant (this is again, done for each datapoint).

   - see Equation 8.2 - minimize $J(...)$ with respect to $c^{(1)}, c^{(2)}, ..., c^{(n)}$and hold $\mu_1, \mu_2, ..., \mu_k$fixed.

2. **Move centroid step** - based on result, clusters are moved to a different location, calculated as means of positions of each individual points in a given cluster. If no movement was done, the algorithm ends.
   - see Equation 8.2 - minimize $J(...)$ with respect to $\mu_1, \mu_2, ..., \mu_k$.

- The algorithm can end up in local optima, if the initially picked up cluster centroids are totally wrong. Solution is to run it 50 or 1,000 times, calculate cost function and in the end, pick up just 1 with the best result (or the average result) = but only good if the number of clusters is relatively small (e.g. 2-10). If we have hundreds of clusters, it is not going to make a big difference.

- Less automatic technique for choosing the number of clusters is called **elbow method** and it is depicted on the next figure. Another techniques how to find k are **average silhouette method**, or **gap statistics** (this should be quite effective).

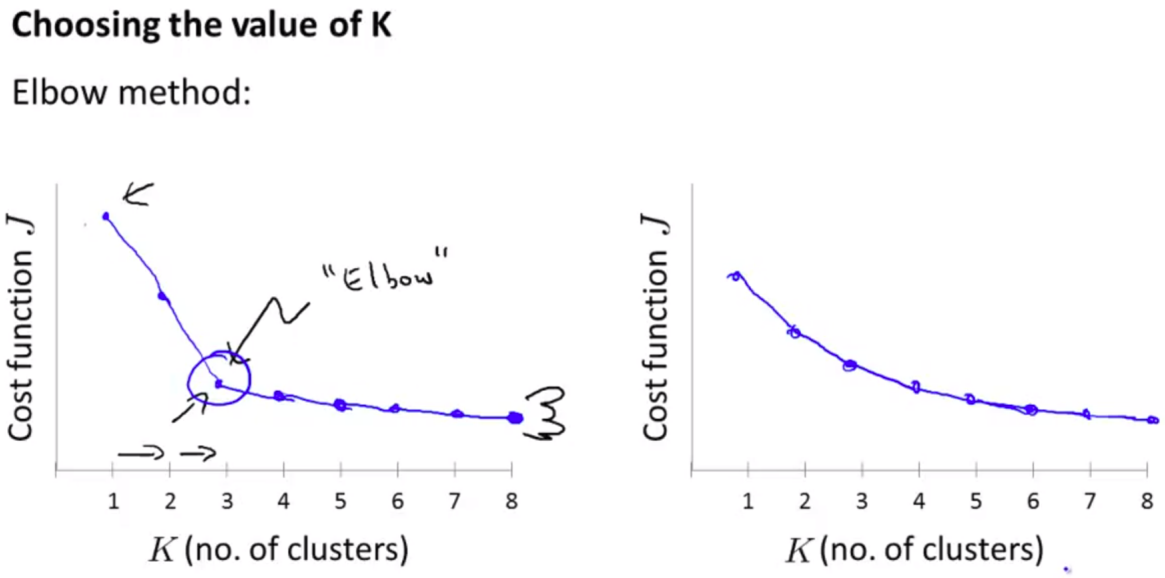


Figure 8.1: An example of elbow method used for determinign a good value of number of clusters. Right plot: until 3 ("elbow") it goes very rapidly, but then not so much, there is only a little difference. So maybe 3 clusters is a good idea. However , right plot: it turnes out that elbow method is not used so often because it can end up like this. The best way is to think about usecases / see data and to ask, for what purpose am I running K-means?

- **Optimization objective (basically cluster assignment step)**

$$J(c^{(1)}, ..., c^{(2)}, \mu_1, ..., \mu_K) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_c(i)||^2 \tag{8.1}$$

$$min_{c^{(1)},...,c^{(2)},\mu_1,...,\mu_K} \; J(c^{(1)}, ..., c^{(2)}, \mu_1, ..., \mu_K) \tag{8.2}$$

where:

  - $c^{(i)}$is an index of a cluster to which example $x^{(i)}$is currently assigned
  - $\mu_k$is cluster centroid k, and we have K clusters
  - $\mu_c(i)$ is cluster centroid to which example $x^{(i)}$ has been assigned

8.2 Hierarchical Clustering Methods

- System of non-empty subsets, and an intersection between each 2 subsets is either empty set, or one of these 2 subsets. This is different than in non-hierarchical clustering methods, in which different non-empty subsets have empty intersection (=each subset is different).
- No need to know a number of classes in advance.
- Distance between clusters - methods based on averaging, centroid, closest neighbor, the most distant neighbor, or medium-based.
- Less computationally expensive.
- No possibility to "fix" wrong decision. If we merge some classes, we will never split them (or vice versa). Solution - combination of hierarchical clustering and some other clustering methods.
- They can be
  - **Aglomerative** - (bottom-top approach) from each individual samples (=clusters) with consecutive merging until there is just 1 cluster.
  - **Division** - (top-bottom approach) on the very beginning there is just 1 cluster and then at the very end there is as many clusters as samples.
- **Representative methods:**
  - Chameleon
  - Diana
  - Agnes
  - BIRCH
  - ROCK

8.3 Density-based Clustering Methods

- Clusters are regions with sufficiently high density of objects. A cluster is increased until density of objects in its neighborhood does not exceed below some threshold.
- Clusters may have different shapes, which is advantage. On the other hand, it is needed to define parameter of density.
- **Representative methods:**
  - **DBSCAN** (Density-based Clustering Method Based on Connected Regions with Sufficiently High Density) and **HDBSCAN**
  - **DENCLUE** (Density-based Clustering)

DBSCAN

- Instead of guessing how many clusters you need, by using this method you define two hyperparameters: ϵ and n.
- You start by picking an example x from your dataset at random and assign it to cluster 1. Then you count how many examples have the distance from x that is less or equal to ϵ. If this quantity is greater or equal to n, then you put all these ϵ-neighbors to the same cluster 1.
- Then, you examining each member of cluster 1 and find their respective ϵ-neighbors. If some member of cluster 1 has n or more such neighbors you expand cluster 1 by putting those ϵ-neighbors to the cluster. You continue with expanding cluster 1 until there are no more examples to put in it.
- Then you pick from the dataset another example not belonging to any cluster and put it to cluster 2. You continue like this until all the examples either belong to some cluster, or are marked as outliers. So, an outlier is an example whose ϵ-neighborhood contains less than n examples.
- The advantage of this method is that it can build clusters that have an arbitrary shape, while centroid-based algorithms create clusters that have a shape of a hypersphere.
- The obvious challenge of DBSCAN is to find out a good values of the two hyperparameters DBSCAN is using.

HDBSCAN

- This algorithm keep the advantages of DBSCAN, but it removes the need to decide on the value of ϵ. The algorithm is capable of building clusters of varying density.

- Modern implementations of HDBSCAN are much slower than k-means, but for practical tasks, the qualities of HDBSCAN may out-weight its drawbacks. It is recommended to use HDBSCAN on your dataset first.

8.4 Grid-based Subspace Clustering Methods

- Space of objects is divided to finite number of cells.
- Short time of processing data, which is not dependent on the number of objects, but on the number of cells.
- **Representative methods:**
  - WaveCluster
  - Clique (Clustering in Quest)

8.5 Clustering Methods Based on Models

- Optimization of match between some mathematical model and dataset.
- These methods find out clusters which maximally corresponds to a given model.
- Data generation is based on probability function.
- **Representative methods:**
  - **Expectation-Maximization**
    - k number of probabilistic distribution functions, each function represents 1 cluster. This method is able to find out parameters of such distribution functions (so it's vector of parameters).
    - This method is similar to k-means. Initiation - randomly pick k objects, which will represents clusters, and then estimate parameters of distribution functions. Two steps: expectation step (determine probabilities of membership objects to each clusters), and maximization step (recalculate vectors of parameters based on probabilities from the previous step).
    - Fast convergence, but not always reaches global optima.
  - **SOM** (Self-organizing Map, also known as Kohonen Map), described in Section 6.2).
  - **COBWEB** - it uses classification tree.

8.6 Gaussian Mixture Model

- TODO - check GMM this can be considered as clustering method. Maybe name this chapter to unsupervised learning (with sections: density estimation, clustering, dimensionality reduction, outlier detection).
- Computing a GMM is very similar to doing model-based density estimation (see Section **??**). In GMM, instead of having just one multivariate normal distribution (MND), we have a weighted sum of several MNDs:

$$fx = \sum_{j=1}^{k} \phi_j f_{\mu_j, \Sigma_j} \tag{8.3}$$

where f_{μ_j,Σ_j} is a MND j, and ϕ_j is its weight in the sum. The values of parameters μ_j, Σ_j and ϕ for all $j = 1, ..., k$ are obtained using the **expectation maximization algorithm** (EM) to optimize the **maximum likelihood** criterion.

- Again, for simplicity, let's consider one-dimensional data. Also assume that there are two clusters, $k = 2$. In this case, we have two Gaussian distributions:
  $f(x|\mu_1, \sigma_1^2) = \frac{1}{\sqrt{2\pi\sigma_1^2}} exp - \frac{(x-\mu_1)^2}{2\sigma_1^2}$ and $f(x|\mu_2, \sigma_2^2) = \frac{1}{\sqrt{2\pi\sigma_2^2}} exp - \frac{(x-\mu_2)^2}{2\sigma_2^2}$
  where $f(x|\mu_1, \sigma_1^2)$ and $f(x|\mu_2, \sigma_2^2)$ are two pdf defining the likelihood of $X = x$.

- We use expectation maximization algorithm to estimate $\mu_1, \sigma_1^2, \mu_2, \sigma_2^2, \phi_1, \phi_2$. The last two parameters are useful for the density estimation and less useful for clustering.

- EM works like follows. In the beginning, we guess the initial values for $\mu_1, \sigma_1^2, \mu_2, \sigma_2^2$, and set $\phi_1 = \phi_2 = 0.5$ (in general, it is $\frac{1}{k}$for each $\phi_j, j \in 1, ..., k$. At each iteration of EM, the following 4 steps are executed iteratively until the values $\mu_j$and σ_j^2 don't change much (for example, the change is below some threshold ϵ):
  1. For all $i = 1, ..., N$, calculate the likelihood of each $x_i$using
     $f(x_i|\mu_1, \sigma_1^2) = \frac{1}{\sqrt{2\pi\sigma_1^2}} exp - \frac{(x_i-\mu_1)^2}{2\sigma_1^2}$ and $f(x_i|\mu_2, \sigma_2^2) = \frac{1}{\sqrt{2\pi\sigma_2^2}} exp - \frac{(x_i-\mu_2)^2}{2\sigma_2^2}$
  2. Using Bayes' Rule, for each example x_i, calculate the likelihood $b_i^{(j)}$ that the example belongs to cluster $j \in \{1, 2\}$ (in other words, the likelihood that the example was drawn from the Gaussian j):
     $b_i^{(j)} \leftarrow \frac{f(x_i|\mu_j, \sigma_j^2)\phi_j}{f(x_i|\mu_1, \sigma_1^2)\phi_1 + f(x_i|\mu_2, \sigma_2^2)\phi_2}$
     where the parameter $\phi_j$reflects how likely is that our Gaussian distribution j with parameters μ_j and σ_j^2may have produced our dataset. That is why in the beginning we set $\phi_1 = \phi_2 = 0.5$: we don't know how each of the two Gaussians are likely, and we reflect our ignorance by setting the likelihood of both to one half.
  3. Compute the new values of $\mu_j$and $\sigma_j^2, j \in \{1, 2\}$ as:
     $\mu_j \leftarrow \frac{\sum_{i=1}^N b_i^{(j)} x_i}{\sum_{i=1}^N b_i^{(j)}}$ and $\sigma_j^2 \leftarrow \frac{\sum_{i=1}^N b_i^{(j)} (x_i - \mu_j)^2}{\sum_{i=1}^N b_i^{(j)}}$
  4. Update $\phi_j, j \in \{1, 2\}$ as:
     $\phi_j \leftarrow \frac{1}{N} \sum_{i=1}^N b_i^{(j)}$

- EM algorithm is very similar to the k-means - start with random clusters, then iteratively update each cluster's parameters by averaging the data that is assigned to that cluster. The only difference in GMM is that the assignment of an example $x_i$to the cluster j is **soft**: $x_i$belongs to cluster $j$with probability $b_i^{(j)}$. This is why we calculate the new values for μ_j and σ_j^2(in step 3) not as average (used in k-means), but as a weighted average with weights $b_i^{(j)}$.

- Once we have learned the parameters μ_j and σ_j^2 for each cluster j, the membership score of example x in cluster j is given by $f(x|\mu_j, \sigma_j^2)$.

- The extension to n-dimensional data ($n > 1$) is straightforward. Instead of the variance σ^2, we now have the covariance matrix Σ that parametrizes the multinomial normal distribution.
- In contrary to k-means where clusters can only be circular, the clusters in GMM have a form of an ellipse that can have an arbitrary elongation and rotation. The values in the covariance matrix control those properties.
- There is no universally recognized method to choose the right k in GMM. It is recommended to first split the dataset into training and test set. Then you try different k and build a different model f_{tr}^k for each k on the training data. You pick the value of k that maximizes the likelihood of examples in the test set:

  $argmax_k \prod_{i=1}^{|N_{test}|} f_{tr}^k(x_i)$,

  where $|N_{test}|$ is the size of the test set.

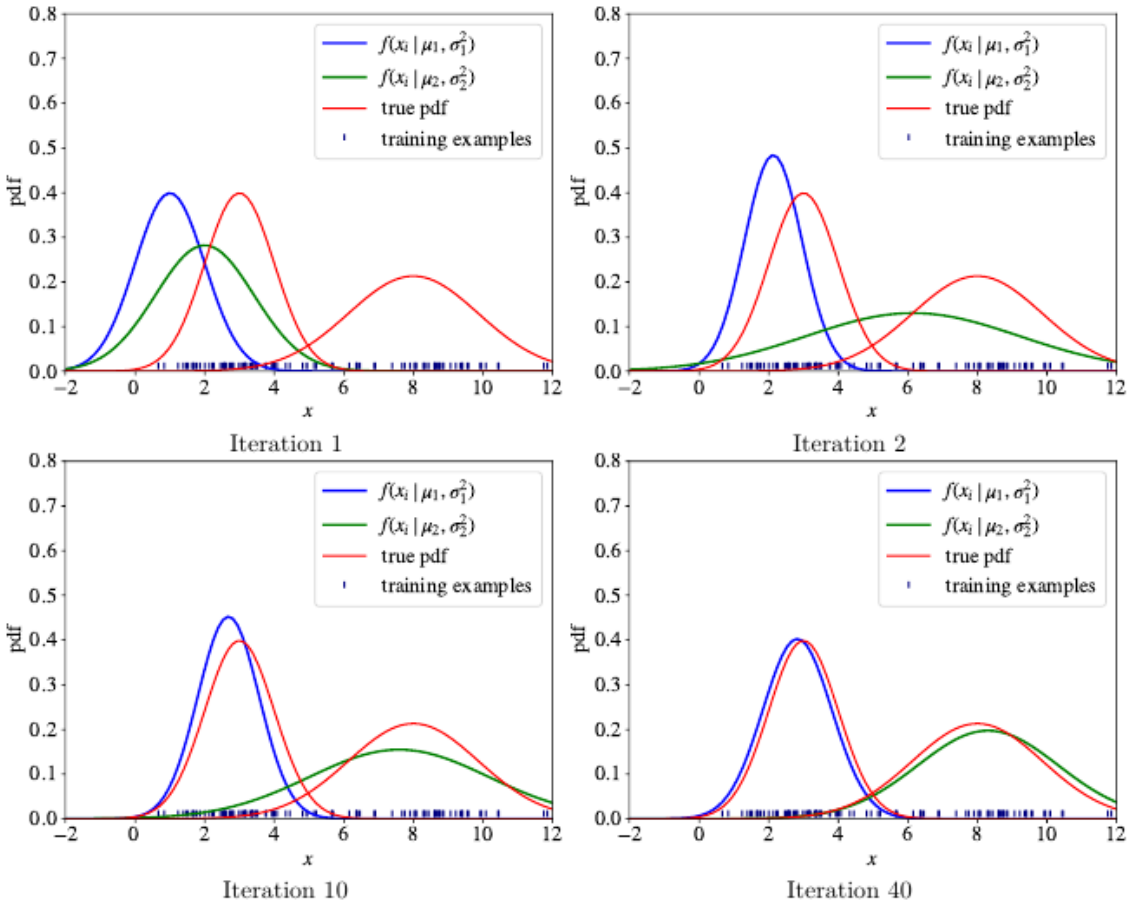


Figure 8.2: The progress of the Gaussian mixture model estimation using the EM algorithm for two clusters (k = 2).

9 Bayesian Methods

- Bayesian methods are those that explicitly apply Bayes' Theorem for problems such as classification and regression.
- Statistical learning algorithms.

Permutations vs Combinations

- **Permutations** - order matters. $\frac{n!}{(n-m)!}$ where n is a number of unique objects, and m is a number of unique attributes.
- **Combinations** - order does not matter. $\frac{n!}{(n-m)!m!}$ where n is a number of unique objects, and m is a number of unique attributes or groups. Shortly, $\begin{pmatrix} n \\ m \end{pmatrix}$. This is basically an approach "without replacement". How many teams of 5 people can be formed from 10 people? The answer is $\begin{pmatrix} 10 \\ 5 \end{pmatrix} = 252$.
- Replacement:
  - **Sampling with replacement** (**independent**), e.g. drawing a card and putting it back in the deck.
  - **Sampling without replacement,** e.g. drawing a card from a deck and not putting it back.
- With the options permutation, combination, with replacement, and without replacement, we have most of the probability situations that are likely to arise in a basic probability world.

Probabilities

- probability - the degree of belief in the truth or falsity of a statement
- probability distribution - collection of statements that are exclusive and exhaustive
- exclusive - given complete information, no more than one of the statements can be true
- exhaustive - given complete information, at least one of the statements must be true

- **A-priori (or prior) probability**: just "state" of a thing, without anything else included. Very simple one. It is given at the beginning; vs posterior - given from later analysis.
- This is called **Bayesian prior**. It may be worth noting that this probability can be interpreted as a marginal probability, summed over all possible data that you **could** see. **So, prior probability is an example of a marginal probability.**
  - $P(apple)$
- **Joint probability**: something AND something (and from this, we can calculate something OR something). **In other words, joint probabilities are the probabilities that two separate events from two separate probability distributions are both true.** If things are related to each other, this is not joint probabilities and we cannot do a multiplication of such probabilities.
  - $P(apple, heavy) = P(heavy, apple)$ - ordering does not matter
  - $P(x, y) = P(x) * P(y)$ ... this means, that joint distribution is equal to product distribution - probabilities are independent
  - previous can be re-written as: $P(apple, heavy) = P(apple|heavy) * P(heavy) = P(heavy|apple) * P(apple)$

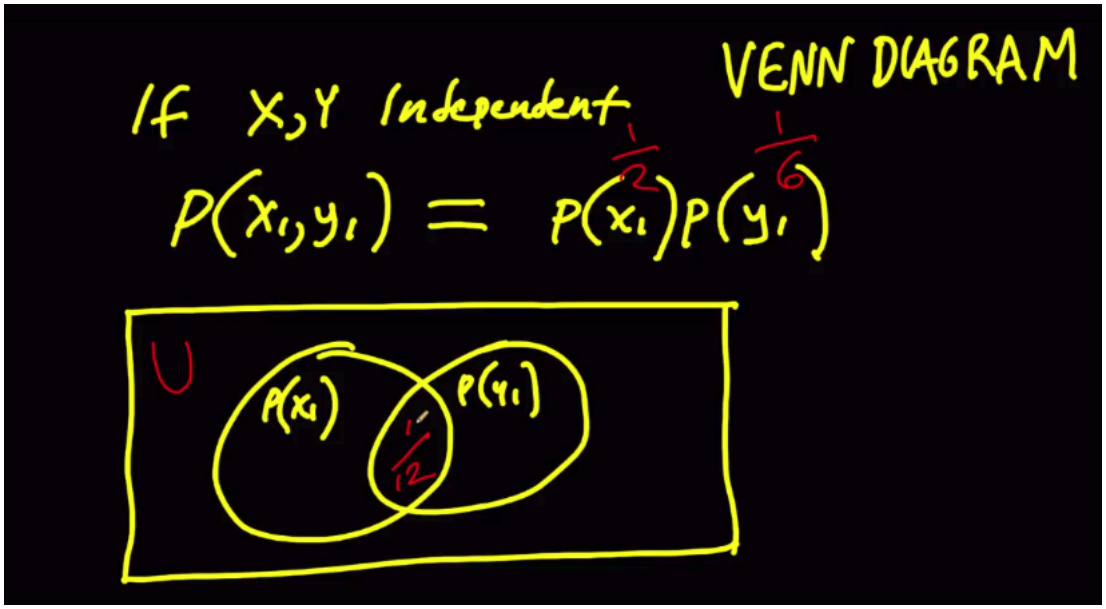


Figure 9.1: Joint probabilities explanation through Venn diagrams.

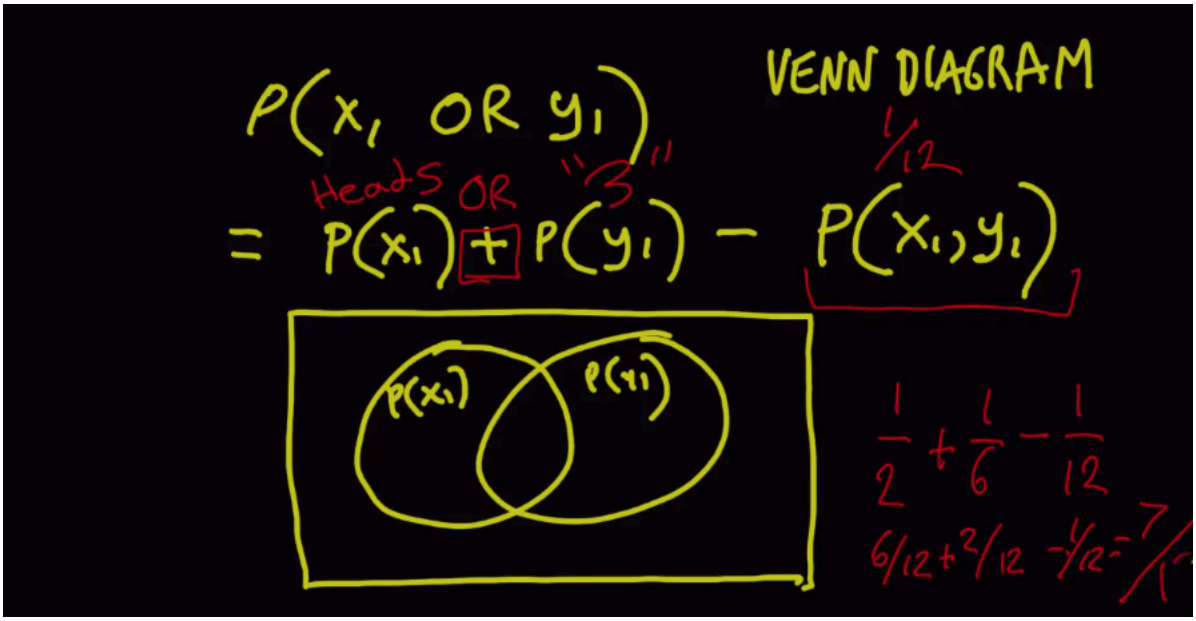


Figure 9.2: Joint probabilities explanation through Venn diagrams. In this example, we want to calculate probability of 1 of 2 events (one event or another event) using joing probabilities. The result is $\frac{7}{12}$.

- **Conditional probability**:
  - The probability that a statement is true given that some other statement is true with certainty.
  - It is calculated as $P(A|B)$, where A are relevant outcomes, and B are total outcomes remaining in universe, when B is true (so not the whole universe, just a subset where B is true).
  - $P(apple|heavy)$
  - For example, if I throw a six-sided dice and it comes up odd, what is the conditional probability that it is a 3? Well that would be three odd rolls, 1, 3, and 5, so the conditional probability would be 1/3.
  - Another example, if I throw 3 with certainty? What is the conditional probability that my throw is odd? In this case, the probability is 1. It's odd with certainty if it's 3 with certainty.
  - **Product rule** - calculation of **conditional** probability from **joint** probability and **marginal** probability.

    $P(A|B) = \frac{P(A,B)}{P(B)}$ .... joint probability of both A and B are true, divided by a marginal probability that B is true. BTW, important, if we multiply both sides by $P(B)$, then $P(A|B)P(B) = P(A,B)$.

There is independence in product rule: knowing, that B is true, gives us nothing about probability of A: $P(A, B) = P(A).P(B)$ ... let's divide both sides with $P(B)$: $P(A|B) = P(A)$. If A and B are dependent, so the opposite, then conditional probability of $P(A|B) \neq P(A)$. Probability distributions are either dependent, or independent, there is nothing between it.

- **Marginal probabilities**
  - It often happens in probability problems that **we know the joint probabilities** that 2 things will happen together. What we want to know is = individual probability that only one of those things will happen, regardless of the other event.
  - **Sum rule** - calculation of marginal probabilities. We can add together joint probabilities to get a marginal probability.

    $P(A) = P(A, B) + P(A, \neg B)$ .... for binary probability distribution

    $P(A) = P(A, B_1) + P(A, B_2), ...P(A, B_n)$ ... for series of probabilities of B - marginal probability of A is calculated as their sum

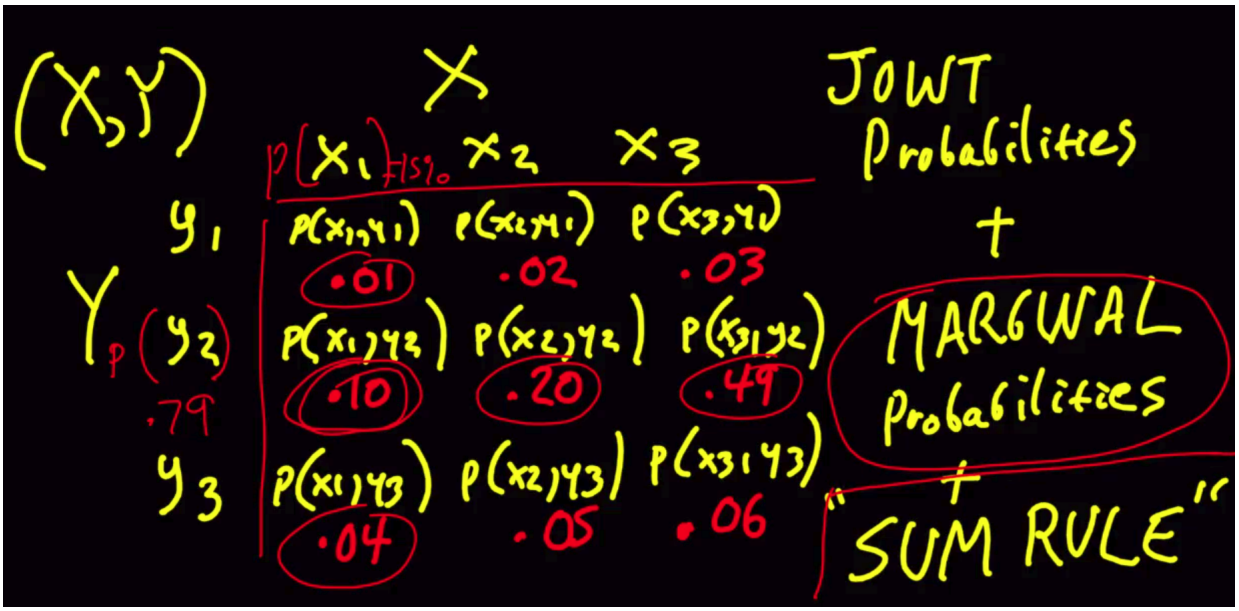


Figure 9.3: Marginal probabilities and the sum rule.

Often know the joint probabilities, but don't know individual probabilities.

Table of known joint probabilities:

(X, Y)		X		
		x_1	x_2	x_3
Y	y_1	$P(x_1, y_1)$	$P(x_2, y_1)$	$P(x_3, y_1)$
		0.01	0.02	0.03
	y_2	$P(x_1, y_2)$	$P(x_2, y_2)$	$P(x_3, y_2)$
		0.10	0.20	0.49
	y_3	$P(x_1, y_3)$	$P(x_2, y_3)$	$P(x_3, y_3)$
		0.04	0.05	0.06

Can refer to $P(x_1)$ as the "marginal probability of x_1" because it is in the margins of the matrix.

Sum rule: The marginal probability is equal to the sum of the joint probabilities.

For x_1, this means:

$$\begin{aligned} P(x_1) &= P(x_1, y_1) + P(x_1, y_2) + P(x_1, y_3) \\ &= 0.01 + 0.10 + 0.04 = 0.15 \end{aligned}$$

and for y_2:

$$\begin{aligned} P(y_2) &= P(x_1, y_2) + P(x_2, y_2) + P(x_3, y_2) \\ &= 0.10 + 0.20 + 0.49 = 0.79 \end{aligned}$$

Figure 9.4: Marginal probabilities and the sum rule 2.

Binomial Theorem

- It's binomial because it's used when there are two possible outcomes - a success or a non-success.

$$\binom{n}{s} p^s (1-p)^{(n-s)} \tag{9.1}$$

where

  - n = number of independent **trials** (with replacement)
  - s = number of **successes**
  - p = probability of 1 success

  - which is basically a probability of s successes in n trials, when probability of 1 success is p. $\begin{pmatrix} n \\ s \end{pmatrix}$ represents the number of ways in which you can have s successes in n trials, $(1-p)$ represents a probability of failures, and $(n-s)$ is a number of failures.

- For example, let's say I want to know what is the probability of getting a certain number of heads in a string of coin tosses ()only 2 outcomes possible - heads or tails). Binomial theorem will tell me the answer.

Bayes Theorem For Discrete Features

$$P(a|b) = \frac{P(b|a) \,.\, P(a)}{P(b)} \tag{9.2}$$

- (probability of some feature in class * probability of class) divided by probability of feature
  - a: class (it is omega)
  - b: feature (or data)
  - **posterior probability**: $P(a|b)$ - "posterior" means "after" - so this is a probability after a new data are observed
  - **likelihood**: $P(b|a)$ - likelihood of data given a parameter
  - **prior probability**: $P(a)$ - before any/new data were observed
  - **evidence**: $P(b)$, it is same like $SUM_w P(a,x) = SUM_w P(x|a) \,.\, P(a)$ which is also known as marginal probability (of the data)

- See **Product rule above**. We have product rule, and from it we can calculate joint probabilities. Since $P(A,B) = P(B,A)$, we can apply product rule to right side, resulting in $P(A|B)P(B) = P(B|A)P(A)$ and by dividing by $P(B)$, we have resulting Bayes Theorem $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$.
  - One of the most powerful uses of Bayes Theorem (so called **Inverse probability problems**) is where B are observed data, and A_i is possible process i with probability parameter θ_i. This parameter is causing or generating data we observe.
  - Example of such usage of Bayes Theorem from above - we got 2 urns: A_1 (probability of white marble is here 20%) and A_2 (probability of white marble is here 10%). We observe 3 marbles in a row, drawing with replacement. But we don't know which urn we are observing. The probability that we are observing urn1 (here urn1 is a process with parameter(white marble)=0.2, similarly urn2 is also a process). So this problem is about what is a probability of unknown process. So, $P(process\,parameter|observed\,data)$.

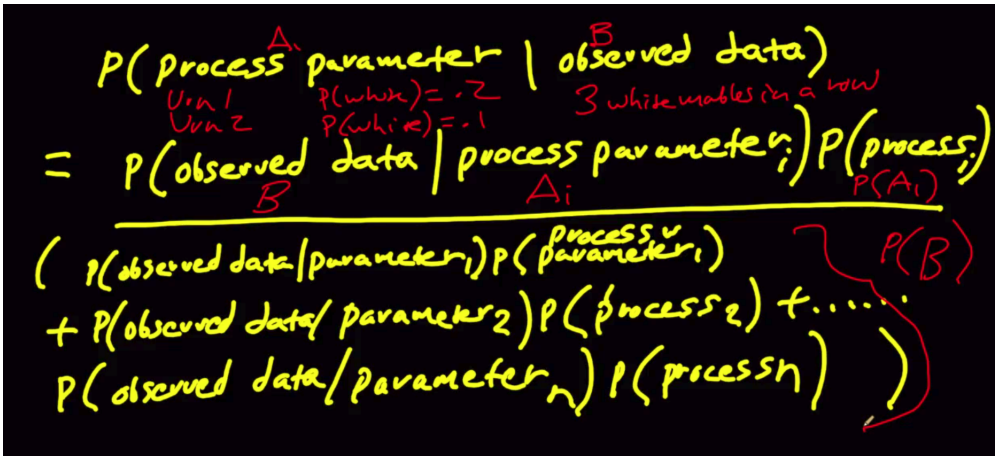


Figure 9.5: Bayes theorem, example application.

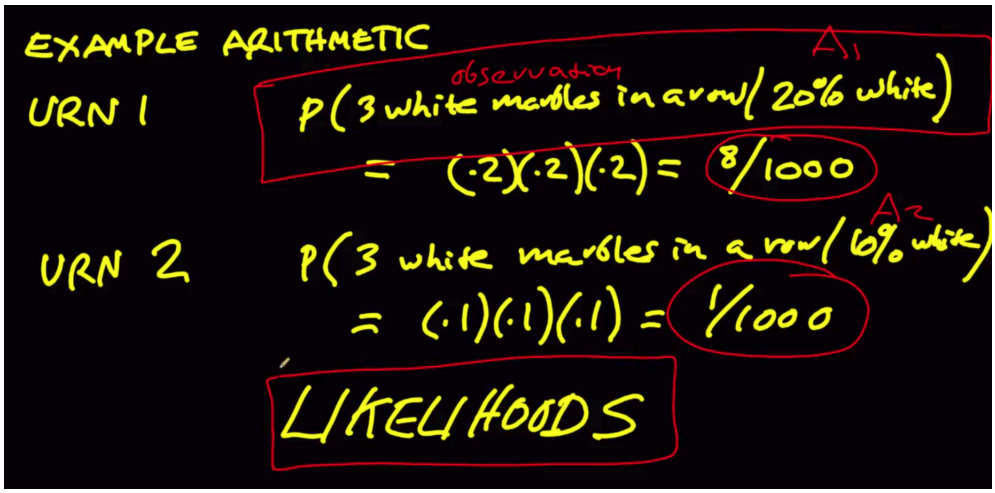


Figure 9.6: Bayes theorem, cont. with example application with certain numbers.

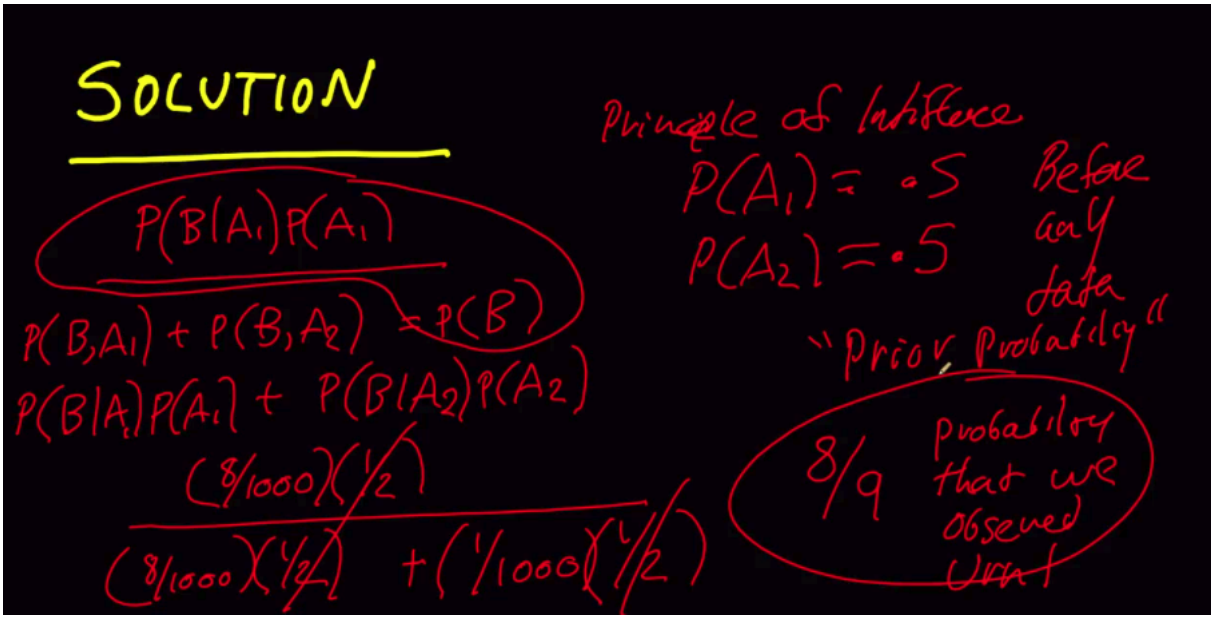


Figure 9.7: Bayes theorem, cont. with example application with a solution.

- The strength of evidence - the rule saying how much to revise our probabilities (change our minds) when we learn a new fact or observe new evidence.[1]
- **One of the really powerful things about Bayes Theorem is it allows us an update of our probabilities based on new data.**
- **Proof:**
  1. $P(a,b) = P(a|b).P(b)$ and $P(b,a) = P(b|a).P(a)$
  2. The order of a presence of events in probabilities does not matter. Therefore, we can write: $P(a,b) = P(b,a)$. So, the following must be also equal: $P(a|b).P(b) = P(b|a).P(a)$. Then the following must be also equal: $P(a|b) = \frac{P(b|a).P(a)}{P(b)}$, so $Posterior = \frac{Likehood\,.\,Prior}{Evidence}$.
- An example (can be taken as an interview question) is on figure below.

[1] https://arbital.com/p/bayes_rule/?l=1zq

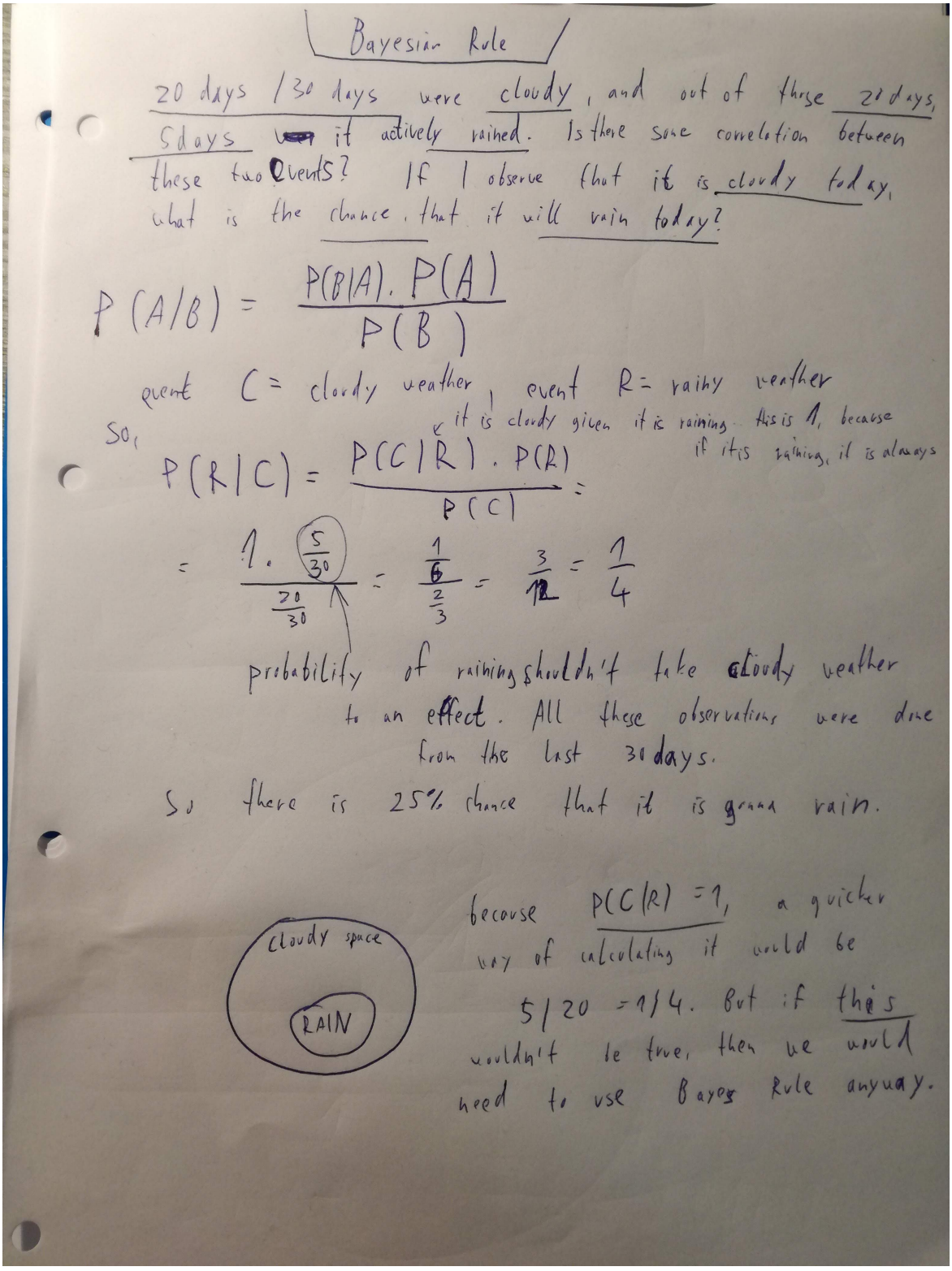


Figure 9.8: Bayesian rule applied to a simple example with weather calculation.

9.1 Naive Bayes

- The major advantage of the naive Bayes classifier is its short computational time for training.
- Naive Bayes methods train very quickly since they require only a single pass on the data either to count **frequencies** (for **discrete variables**) or to compute the **normal probability density function** (for **continuous variables** under normality assumptions).
- It is naturally robust to missing values since these are simply ignored in computing probabilities and hence have no impact on the final decision.
- **Naive Bayes is a probabilistic classifier that is based on Bayes Rule with the assumption that all the input attributes are independent.**
- Different explanation of Bayes Rule and Naive Bayes - conditional probability of observing event A given B, can be computed from conditional probability of observing event B given A, and prior probabilities B and A. In classification task, we are interested in computing probability that class y of an instance $x = \{x^1, x^2, ...x^D\}$ is c, leading into equation $P(y = c|x) = \frac{P(y=c)P(x|y=c)}{P(x)} = \frac{P(y=c)P(x^1,x^2,...,x^D|y=c)}{P(x^1,x^2,...,x^D)}$. The most difficult is to compute $P(x^1, x^2, ..., x^D|y = c)$ as it is difficult to compute conditional joint probability of the features. That is why in Naive Bayes there exists an assumption of the independence, which allows rewriting the equation as: $P(y = c|x) = \frac{P(y=c)\prod_{i=1}^{D} P(x^i|y=c)}{P(x^1,x^2,...,x^D)}$. Then, training of models consists of computing the probabilities of each value of all the attributes x^i being of the given classes $c \in C$. During the prediction, the class with the highest probability is returned as the predicted class.

9.2 Gaussian Naive Bayes*

9.3 Multinomial Naive Bayes*

9.4 Averaged One-Dependence Estimators*

9.5 Bayesian Belief Networks*

9.6 Bayesian Networks*

- Graphical model for probability relationships among a set of variables (features).
- The Bayesian network structure S is a directed acyclic graph (DAG) and the nodes in S are in one-to-one correspondence with the features X.

- Learning is a 2 stage process:
  - learning of the DAG structure of the network,
  - determination of its parameters.

10 Dimensionality Reduction Techniques

- Simplifies inputs by mapping them into a lower-dimensional space.
- Removing of redundant of highly correlated features, but is also reduces the noise in the data.
- Usage: data visualization, and also some learning algorithms may perform faster.
- Algorithms usually creates some new "feature" which is calculated by several older ones, which are deleted. We can visualize data once we have 2 or 3 dimensions.
- Three algorithms are widely used - **PCA**, **UMAP**, and **autoencoders**.

10.1 Linear Discriminant Analysis

- **Supervised learning method**.
- Statistical learning algorithm.
- Simple method used in statistics and machine learning to find the linear combination of features which best separate two or more classes of object.
- It works when the measurements made on each observation are continuous quantities. When dealing with categorical variables, the equivalent technique is Discriminant Correspondence Analysis.
- Method is trying to project data to such direction, that the distances between mean values of classes are maximized, and the average variance of classes is minimized.
- Covariant matrix is same for all classes.

10.2 Principal Component Analysis

- **Unsupervised learning method.**
- It is basically linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.
- By minimizing the average square area between a data point and its projection onto the principle subspace, the best thing to do is to project onto the subspace that is spanned by the eigenvectors that belonged to the largest eigenvalues, which is the goal of PCA.
- **Usage** (to reduce dimensionality)
  - Compression: speed up learning algorithms, reducing memory/disk needed to store data.
  - Data visualization (2D/3D).
  - **NOT** for preventing overfitting! We have regularization for that!
- It does not use labels, so it can throw away some potential information. **Do not use PCA if not needed.** Always perform whatever you want to do without PCA on original raw data.
- If we think of variance in the data as information contained in the data, this means that PCA can also be interpreted as a method that retains as much information as possible.
- We can also think of PCA as a linear autoencoder. An autoencoder encodes a data point x and tries to decode it to something similar to the same data point. The mapping from the data to the code is called an encoder. If the encoder and decoder are linear mappings, then we get the PCA solution when we minimize the squared auto-encoding loss. If we replace the linear mapping of PCA with a nonlinear mapping, we get a nonlinear autoencoder. A prominent example of this is a deep autoencoder when the linear functions of the encoder and decoder are replaced with deep neural networks.
- What it does is that if we took a high dimensional vector (data point) x, we can project it onto a lower dimensional representation z using the matrix B^T (this part can be thought as an encoder). The columns of this matrix B are the eigenvectors of the data covariance matrix that are associated with the largest eigenvalues. The z values are the coordinates of our data point with respect to the basis vectors which span the principal subspace, and that is also called the code of our data point. Once we have that lower dimensional representation z, we can get a higher dimensional version of it by using the matrix B again, so multiplying B onto z to get a higher dimensional version of the z in the original data space (we can think of this as decoder). We have, of course some reconstruction error here, but PCA is trying to minimize such error.

- PCA should run on training set. The resulting mapping can run also on testing/validation set.
- Sensitive to relative scaling of original dataset. It is good enough to perform mean normalization, so that all features have comparable range of values.
- Provides de-correlation by orthogonal transformation.
- There exist also kernel version (like "kernel trick" in SVM) - **Kernel PCA**, for non-linear dimensionality reduction.
- PCA is not linear regression, as we can see on the following figure. In linear regression, we have features and then value y which we are trying to predict. In PCA, we have features which are treated equally.

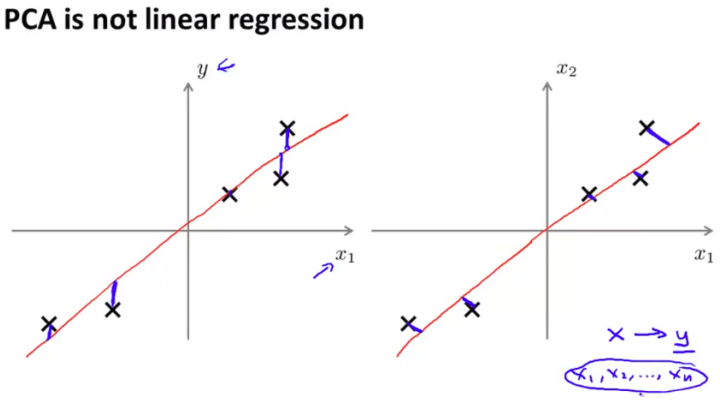


Figure 10.1: PCA and linear regression uses two very different algorithm. In logistic refression (left), the algorithm is trying to lower down (minimize) a vertical distance between value from data and predicted value from the hypothesis. PCA (right) is trying to lower down the magnitude of the blue lines which are drawn at an angle giving the shortest orthogonal distances between the red line and a given point x.

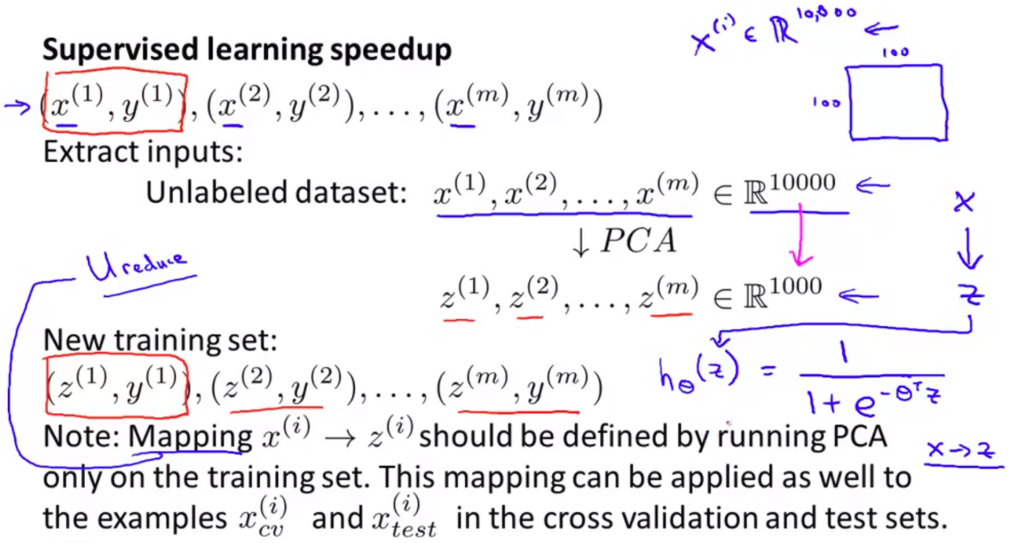


Figure 10.2: **An intuition about algorithm, how to use PCA on our data.** To reduce dimensions from 10,000 to 1,000 is not that unrealistic, in real world we usually reduces by 5x or 10x without a significant impact on accuracy.

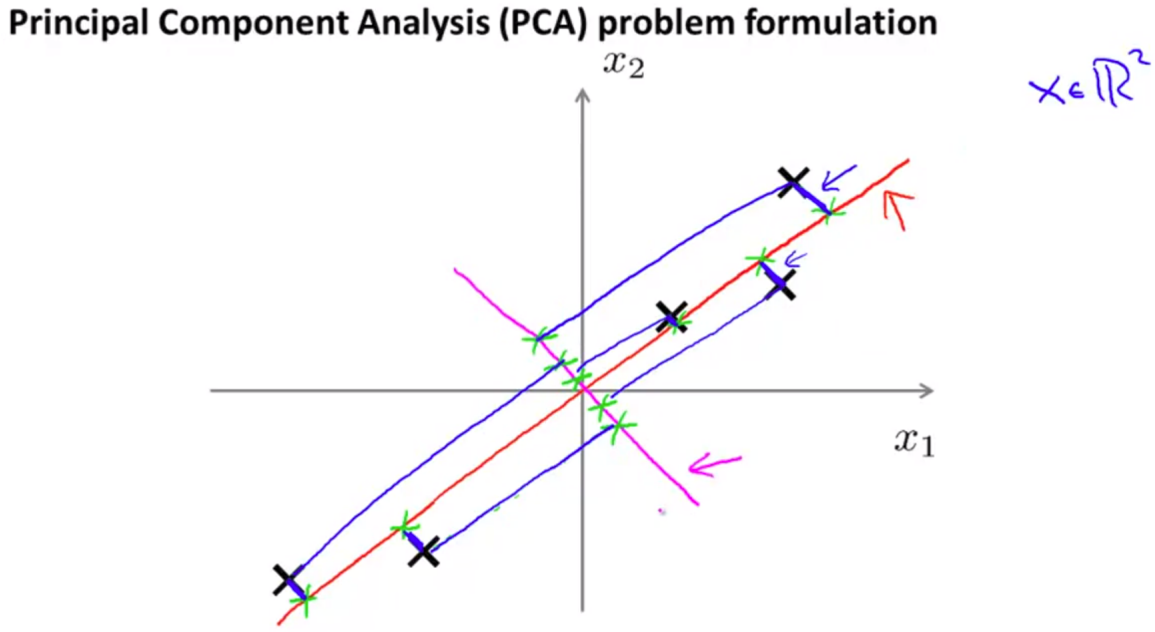


Figure 10.3: An example of dimensionality reduction of the data from 2D to 1D. So we wanted to find a line onto which to project the data. **Red line**: the distance between each point and its projected version is pretty small (=blue line segments on plot are pretty short). In fact, PCA tries to find a lower dimensional surface, in this case a line, onto which to project data so that the **sum of squares of these blue line segments is minimized. The length of these segments is called the projection error. Magenta line:** we can see a great projection error for blue lines. And it is very bad, PCA would choose red line, not magenta line. **So, PCA tries to find a direction (a vector) onto which to project data so that the projection error is minimized.**

- **Algorithm**
  1. Perform mean normalization (so that every feature has zero mean), you can also perform feature scaling. If you do not perform mean normalization, PCA will rotate the data in a possibly undesired way. So by subtracting the mean value from each data point, data is now centered and we can avoid numerical problems.
  2. Next, we divide by the standard deviation. Now the data is unit-free (and by doing this, we can avoid having centimeters in one dimension, and meters in another one, for example). Also, data has variance equals to 1 along each axes.
  3. Compute **covariance matrix** ($n\,x\,n$ matrix, because $x^{(i)}$is $n\,x\,1$ vector, and then there is its transposed version $1\,x\,n$):

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T \tag{10.1}$$

  4. Compute **eigenvectors**[1] (also known as **principal components**), computed from **covariance matrix** (there are multiple ways, one is to calculate SVD - singular value decomposition).
     - From n dimensions to k dimensions - find k eigenvectors $u^{(1)}, u^{(2)}, \dots u^{(k)}$ onto which to project the data, so that projection error is minimized (there are more eigenvectors, n totally, and we will pick only the first k, because this is our resulting dimension we want to have).
     - All eigenvectors have the same dimension (n dimensions each eigenvector).
     - From eigenvectors and original data $(x^1, y^1), (x^2, y^2)...(x^{(m)}, y^{(m)})$, where x^j is n-dimensional (we are not interested in labels), we can compute a new feature space for datapoints, resulting in $z^{(1)}, z^2, ..., z^{(m)}$ where $z^{(j)}$is k-dimensional and we have m samples.
     - How to estimate a number of dimensions (so, how to select k)? Iteratively, to obtain as smallest projection error as possible as detailed in the following figure.

[1] There is also a term eigenvalues. The eigenvectors are scaled by the magnitude of the corresponding eigenvalue. Eivengectors refers to diagonal matrix created from eigenvalues. So these are scalar values representing principal components. We can work with them, or just with eigenvalues matrix for transforming feature space of our examples for visualizing, training, or so.

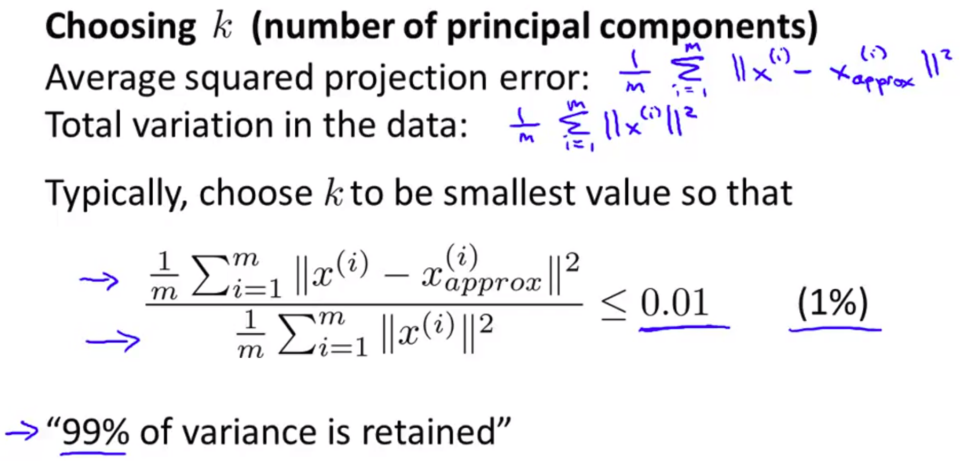


Figure 10.4: Choosing a number of principal components in PCA should not be based on some guessed number, but on retained variance - it should be at least 99% (or 95%, different people use different values). And based on this, choose as smallest k as possible. It is recommended to start with $k = 1$, and then compute $U_{reduce}, z^{(1)}, ...z^{(m)}, x_{approx}^{(1)}, ...x_{approx}^{(m)}$ and then check how much variance is retained. If it is not at least 99%, then increment k and repeat. There is no need to compute PCA again with different $k =$ just once by computation of variance retained by choosing different k using equation on the figure.

5. Then if you want to have $z \in \mathbb{R}^k$ (z^j is "transformed" example x^j after PCA), just multiply these vectors (put them into 1 matrix, resulting in $n\,x\,k$ matrix and then transpose it) with x (which can be training set, evaluation set, or test set data, $x \in \mathbb{R}^n$). Figure below describes this step better.
6. If we want to add a new example x^{new}, we have to perform mean normalization on it, divide it by the previously computed standard deviation (so we have to remember these 2 terms). And we have to do it obviously in every dimension. And then we can project resulting normalized data point $x^{norm\,new}$ onto principal subspace ($BB^T x^{norm\,new}$, where B is the matrix that contains the eigenvectors that belong to the largest eigenvalues - largest variance, which is what we want here. And $B^T x^{new}$ are the coordinates of the projection with respect to the basis of the principal subspace).

In PCA, we obtain $z \in \mathbb{R}^k$ from $x \in \mathbb{R}^n$ as follows:

$$z = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(k)} \\ | & | & & | \end{bmatrix}^T x = \begin{bmatrix} --- & (u^{(1)})^T & --- \\ --- & (u^{(2)})^T & --- \\ & \vdots & \\ --- & (u^{(k)})^T & --- \end{bmatrix} x$$

Figure 10.5: Computation of z (also for reconstruction to the previous feature space) in PCA from eigenvectors and data sample. We can compute some particular datapoint $z_j = (u^{(j)})^T x$. Then if we want to go back, just calculate $x_{approx} = U_{reduce} * z$ where $U_{reduce}$is $n\,x\,k$ dimensional vector and z is $k\,x\,1$ vector. If we would perform $n = k$, so that we do not reduce dimensionality, then $x_{approx} = x$ for every example x, and the percentage of variance retained is 100%.

- Consider a two-dimensional data as shown on the figure below. Principal components are vectors that define a new coordinate system in which the first axis goes in the direction of the highest variance in the data. The second axis is orthogonal to the first one and goes in the direction of the second highest variance in the data. If our data was three-dimensional, the third axis would be orthogonal to both the first and the second axes and go in the direction of the third highest variance, and so on. In figure below, the two principal components are shown as arrows. The length of the arrow reflects the variance in this direction.

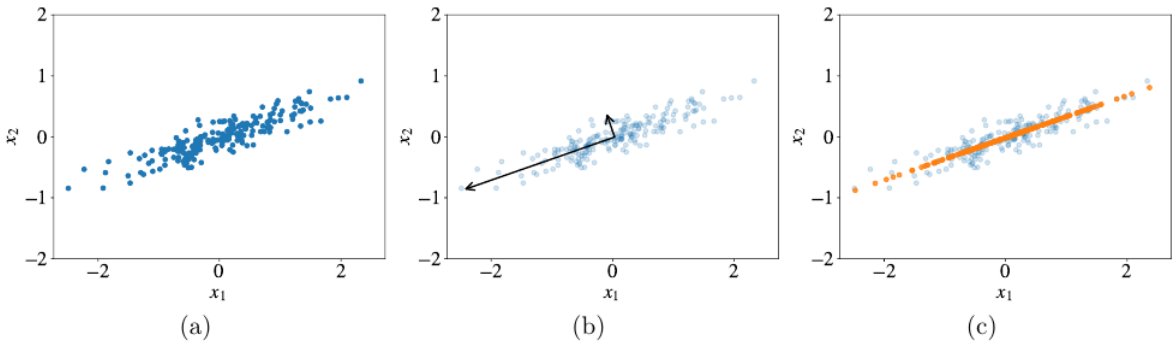


Figure 10.6: PCA: (a) the original data; (b) two principal components displayed as vectors; (c) the data projected on the first principal component.

10.3 Sammon Mapping*

10.4 Multidomensional Scaling*

10.5 Projection Pursuit*

10.6 Principal Component Regression*

10.7 Mixture Discriminant Analysis*

10.8 Quadratic Discriminant Analysis*

10.9 Flexible Disctiminant Analysis*

10.10 Uniform Manifold Approximation and Projection

- The idea behind many of the modern dimensionality reduction algorithms, especially those designed specifically for visualization purposes, such as **t-SNE** and **UMAP**, is basically the same. We first design a similarity metric for two examples. For visualization purposes, besides the Euclidean distance between the two examples, this similarity metric often reflects some local properties of the two examples, such as the density of other examples around them.

- Here, the similarity metric w is defined as:

$$w(x_i, x_j) = w_i(x_i, x_j) + w_j(x_j, x_i) - w_i(x_i, x_j)w_j(x_j, x_i) \tag{10.2}$$

  where $w_i(x_i, x_j) = exp(-\frac{d(x_i,x_j)-\rho_i}{\sigma_i})$ and $d(x_i, x_j)$ is the Euclidean distance between two examples, $\rho_i$is the distance from x_i to its closest neighbor, and σ_i is the distance from x_i to its k^{th} closest neighbor (k is a hyperparameter to the algorithm). This similarity metric w is symmetric (which means that $w(x_i, x_j) = w(x_j, x_i)$, and is in the range from 0 to 1.

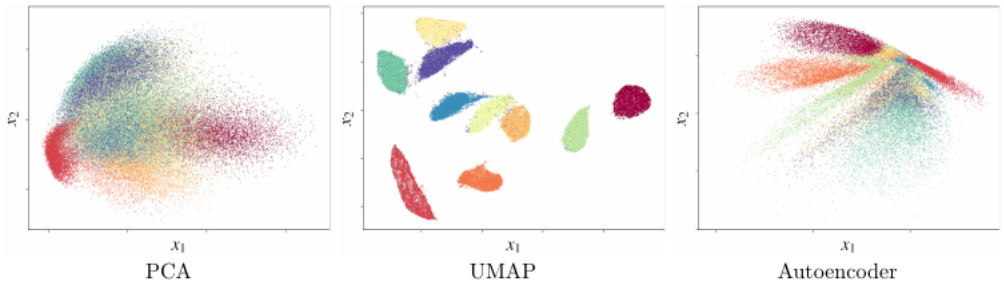


Figure 10.7: Dimensionality reduction of the MNIST dataset using three different techniques. This dataset contains 70k labeled examples. Ten different colors on the plot correspond to ten classes. Each point on the plot corresponds a specific example in the dataset. As you can see, UMAP separates examples visually better (remember, it doesn't have access to labels). In practice, UMAP is slightly slower than PCA but faster than autoencoder.

- Let w denote the similarity of two examples in the original high-dimensional space, and let $w^{'}$be the similarity given by the same equation for similarity metric above, in the new low-dimensional space.
- Because the values of w and $w^{'}$ lie in the range between 0 and 1, we can see $w(x_i, x_j)$ as membership of the pair of examples (x_i, x_j) in a certain fuzzy set. The same can be said about $w^{'}$. The **notion of similarity of two fuzzy sets** (what fuzzy set is, please see 30) is called **fuzzy set cross-entropy** and is defined as:

$$C_{w,w'} = \sum_{i=1}^{N}\sum_{j=1}^{N}[w(x_i,x_j)ln(\frac{w(x_i,x_j)}{w'(x_i^{'},x_j^{'})})+(1-w(x_i,x_j))ln(\frac{1-w(x_i,x_j)}{1-w'(x_i^{'},x_j^{'})})] \quad (10.3)$$

where $x^{'}$ is the low-dimensional "version" of the original high-dimensional example x. The unknown parameters here are $x_i^{'}$ (for all $i = 1, ..., N$), the low-dimensional examples we look for. We can compute them by gradient descent by minimizing $C_{w,w'}$.

10.11 t-distributed Stochastic Neighbor Embedding*

11 Ensemble Training

Ensemble training (which can be understood as one of possible settings of **meta-learning**) combines **more different models together** or using **many instances of certain model** (typically multiple weaker models), that are trained independently, and whose predictions are combined in some way to to gain even better performance. **The trend is toward larger and larger ensembles.** There are three typical ways to combine models: with **averaging, majority vote,** or **stacking**. **Types**:

- **Bumping** - uses 1 model, chosen from more models trained on same training data. More robust against stacking in local minima, the error of model is calculated on all training data and based on that, the best parameters of model from all created models.
- **Bagging** - (bootstrap aggregating) - **train different models on different subsets of the data.** Random forests use a lot of different decision trees trained using bagging and they work very well. But with NN bagging can be very expensive. Bagging is learning on more models of same type and classification works as **voting system**, all models are equivalent (no weights). It chooses some subset of data with method **bootstrap**, learn a model and then chooses another subset of data for learning and this process is repeated N times, so **at the end there are N learned models.** Prediction - the most occurring class is chosen (for regression, the average value is returned) - result of "voting" of learned N models. **It increases stability and reduces variance/overfitting** (so greatly reducing variance but slightly increasing bias). **Good to use when there is lack of data**, **and/or our model is unstable**, such as ANN or decision trees.[1] Not recommended to use on linear or very robust models. If the outputs of classifiers have some probabilities, there is also weighted variant **MetaCost**.
- **Boosting** - the most used method - **train a sequence of low capacity models.** Weight the training cases differently for each model in the sequence. Boost up-weight samples that the previous models got wrong and boost down-weight samples that the previous models got right. An early use of boosting was with NN for MNIST. More models of the same type with **weak learners** (a bit better than random guess = more than 50% successfully classified samples should be good enough). The algorithm creates models which are specialized on training data which were miss-classified with previous model (so it is iterative). **As a number**

[1] However, decision trees are easily interpretable and understandable, and with these techniques, this advantage will be lost)

of weak models increases, an error of one weak learner will increase, but total error of learners as a group will decrease. If could be variant with (model trained on all training data) or without weighting (model trained only on subset of training data). Example is **Adaboost**, one of the most common algorithm, in which weights are learned automatically. It is able to over-fit the algorithm, it depends on a complexity of a given models.

- **Stacking** - these methods combine various models of lower level. Good when the classifier outputs also some confidence level for predicted output, like ANN, Bayes, logistic regression. Training data chosen by cross validation, which is computationally expensive. Models have weights, and final prediction is based on computation of weights multiplicated on prediction of each model. Ideally, sum of all these weights is 1. Stacking consists of building a meta-model that takes the output of base models as input. To train the stacked model, it is recommended to use examples from the training set and tune the hyperparameters of the stacked model using cross-validation. Obviously, you have to make sure that your stacked model performs better on the validation set than each of the base models you stacked.

Mechanisms that are used to build ensemble of classifiers include using:

- different **subsets of training data** with a **single learning method**,
- different **training parameters** with a **single training method** (e.g., using different initial weights for each neural network in an ensemble), or
- different **learning methods**.

The reason that combining multiple models can bring better performance is the observation that when several **uncorrelated** strong models agree they are more likely to agree on the correct outcome. The keyword here is "uncorrelated". Ideally, base models should be obtained using different features or using algorithms of a different nature - for example, SVMs and Random Forest. Combining different versions of decision tree learning algorithm, or several SVMs with different hyperparameters **may not** result in a significant performance boost.

11.1 Adaptive Mixtures of Local Experts

- The idea of this model is to train a number of neural nets, each of which specializes in a different part of the data. That is, we assume we have a data set which comes from a number of different regimes, and we train a system in which one neural net will specialize in each regime, and managing neural net will look at the input data, and decide which specialist to give it to.
- This kind of system, doesn't make very efficient use of data, because the data is fractionated over all these different experts. And so with small data sets, it can't be expected to do very well. But as data sets get bigger, this kind of system may

well come into its own, because it can make very good use of extremely large data sets.

- In boosting, the weights on the models are not all equal, but after we finish training, each model has the same weight for every test case. We don't make the weights on the individual models depend on which particular case we're dealing with. **In mixture of experts, we do**. So the idea is that we can look at the input data for a particular case during both training and testing to help us decide which model we can rely on.

- During training this will allow models to specialize on a subset of the cases. They then will not learn on cases for which they're not picked. So they can ignore stuff they're not good at modeling. This will lead to individual models that are very good at some things and very bad at other things.

- But how do we partition data to different regimes? The answer is clustering the training data into subsets. But we don't want to cluster data based on the similarity of input vectors. We are only interested in similarity of input-output mappings.

- **IF** we want to encourage **cooperation** by comparing the **average of all predictors with the target** and train to reduce difference between target and prediction. **This is not specialization, this is cooperation. However, want to achieve specialization here. We do this by comparing each predictor separately with the target. We also need a "manager" to determine the probability of picking each expert.**

- Given a test case, some expert might be good at classifying it, but a lot of them might be bad. How do we prevent the bad experts from influencing the classification prediction? The bad experts would be assigned very low probability by the manager and would not contribute much to weighted average.

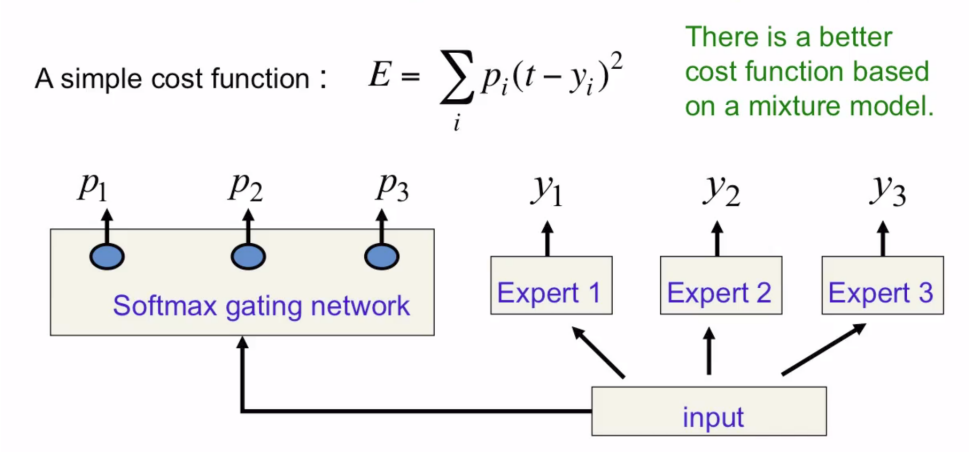


Figure 11.1: Mixture of Experts model architecture. Here, the cost function (actually, this one is simplified) is the squared difference between the output of each expert in the target, averaged over all the experts. But the weights in that average are determined by the manager. So we have an input. Our different experts will look at that input. They all make their predictions based on that input. In addition we have a manager, a manager might have multiple layers and the last layer for manager is a soft max layer, so the manager outputs as many probabilities as there are experts, And using the outputs of the manger and outputs of the experts, we can then compute the value of that error fraction.

11.2 Random Forest

- Random forest is different from the vanilla bagging in just one way. It uses a modified tree learning algorithm that inspects, at each split in the learning process, a random subset of the features. The reason for doing this is to avoid the correlation of the trees: if one or a few features are very strong predictors for the target, these features will be selected to split examples in many trees. This would result in many correlated trees in our "forest." Correlated predictors cannot help in improving the accuracy of prediction. The main reason behind a better performance of model ensembling is that models that are good will likely agree on the same prediction, while bad models will likely disagree on different ones. Correlation will make bad models more likely to agree, which will hamper the majority vote or the average.

- The most important hyperparameters to tune are the number of trees, and the size of the random subset of the features to consider at each split.

- Random forest is one of the most widely used ensemble learning algorithms. Why is it so effective? The reason is that by using multiple samples of the original dataset, we reduce the variance of the final model. Remember that the low vari-

ance means low overfitting. Overfitting happens when our model tries to explain small variations in the dataset because our dataset is just a small sample of the population of all possible examples of the phenomenon we try to model. If we were unlucky with how our training set was sampled, then it could contain some undesirable (but unavoidable) artifacts: noise, outliers and over- or underrepresented examples. By creating multiple random samples with replacement (we randomly pick a sample from a training set, we put this sample into some training subset, but this sample will still stay in the original training set) of our training set, we reduce the effect of these artifacts.

11.3 Gradient Boosting

- Very effective ensemble learning algorithm, based on the idea of boosting.
- It is very powerful machine learning algorithms. It creates very accurate models, and also it is capable of handling huge dataset with millions of examples and features. It usually outperforms random forest in accuracy, but because of its sequential nature, it can be significantly slower in training.
- Gradient boosting can work for regression as well as for classification.
- To build a strong **regressor**,
  - We will start with a constant model (just as in ID3 algorithm) $f = f_0(x) = \frac{1}{N}\sum_{i=1}^{N} y_i$, and then we modify labels of each example $i = 1, ..., N$ in our training set as follows: $\hat{y}_i \leftarrow y_i - f(x_i)$, where $\hat{y}_i$ is called residual and it is the new label for example x_i.
  - Now we will use the modified training set, with residuals instead of original labels, to build a new **decision tree model**, f_1. The boosting model is now defined as $f = f_0 + \alpha f_1$, where α is the learning rate (hyperparameter).
  - Then we again, recompute residuals (again with $\hat{y}_i \leftarrow y_i - f(x_i)$) and replace the labels in the training data once again, train the new decision tree model f_2,redefine the boosting model as $f = f_0 + \alpha f_1 + \alpha f_2$and the process continues until the predefined maximum M of trees are combined.
  - Intuitively, by computing the residuals, we find how well (or poorly) the target of each training example is predicted by the current model f. We then train another tree to fix the errors of the current model (this is why we use residuals instead of real labels), and add this new tree to the existing model with some weight α. Therefore, each additional tree added to the model partially fixes the errors made by the previous trees until the maximum number M (another hyperparameter) of trees are combined.
  - By the way, we don't calculate any gradient and it is still called gradient boosting algorithm. Here, instead of getting the gradient directly (and evaluated our direction at each step with some learning rate), we use its proxy in

the form of residuals: they show us how the model has to be adjusted so that the error (the residual) is reduced.

- There are three principal hyperparameters to tune: learning rate, the depth of the trees, and the number of trees.

- For **classification**,
  - Gradient boosting is similar, but the steps are slightly different. Assume we have M regression decision trees, considering binary case, then also similar to for example logistic regression, the prediction of the ensemble of decision trees is modeled using the sigmoid function: $P(y = 1|x, f) = \frac{1}{1+e^{-f(x)}}$ where $f(x) = \sum_{m=1}^{M} f_m(x)$ and f_m is a regression tree. We apply maximum likelihood principle by trying to find such an f that maximizes $L_f = \sum_{i=1}^{N} ln[P(y_i = 1|x_i, f)]$ (again, to avoid numerical overflow, we maximize the sum of log-likelihoods rather than the product of likelihoods).
  - The algorithm starts with the initial constant model $f = f_0 = \frac{p}{1-p}$,where $p = \frac{1}{N}\sum_{i=1}^{N} y_i$ (it can be shown that such initialization is optimal for the sigmoid function).
  - Then at each iteration m, a new tree $f_m$is added to the model. To find the best f_m, first the partial derivative g_i of the current model is calculated for each $i = 1, ..., N$: $g_i = \frac{dL_f}{df}$, where f is the ensemble classifier model built at the previous iteration $m-1$. So to calculate g_i, we need to find the derivatives of $ln[P(y_i = 1 + x_i, f)] = ln[\frac{1}{1+e^{-f(x_i)}}]$. The derivative of this with respect to f equals to $\frac{1}{e^{f(x_i)}+1}$.
  - Then, we transform our training set by replacing the original label y_i with the corresponding partial derivative g_i, and build a new tree f_m using the transformed training set.
  - Then we find the optimal update step $\rho_m \leftarrow arg\,max_p L_{f+\rho f_m}$and at the end of iteration m, we update the ensemble model f by adding the new tree f_m : $f \leftarrow f + \alpha\rho_m f_m$.
  - We iterate until $m = M$ and then we stop and return the ensemble model f.

11.4 Boosting*

11.5 AdaBoost*

11.6 Bootstrapped Aggregation (Bagging)*

11.7 Stacked Generalization (Blending)*

11.8 Gradient Boosted Regression Trees*

11.9 XGBoost*

- Introduced in 2014. The beauty of this powerful algorithm lies in its scalability, which drives fast learning through parallel and distributed computing and offers efficient memory usage.
- XGBoost is a popular implementation of Gradient Boosted Trees algorithm, a supervised learning method that is based on function approximation by optimizing specific loss functions as well as applying several regularization techniques..

12 Practical Hints & Observations

"It's not who has the best algorithm that wins. It's who has the most data."

Working with our dataset

More details will be in the next subsection. Just an brief idea how it should work - split (ideally randomly) to, for example:

- 70% of data is for training, 30% is for testing. This is now not considered to be a good idea in many cases. This old heuristic does not apply for problems where you have a lot of data; the dev and test sets can be much less than 30% of the data.
- or another example - 60% for training set (optimization of parameters θ), 20% for cross validation set (or just validation set, dev set, or hold-out cross validation set for finding out the best hyperparameters, like polynomial degree of linear regression, select features, and make other decisions regarding the learning algorithm), and 20% for testing set (estimate generalization error with cost function with learned thetas and hyperparameters - you are not making any decisions regarding what learning algorithm or parameters to use). You should choose dev and test sets to reflect data you expect to get in the future and want to do well on. **But don't assume your training distribution is the same as your test distribution.** Try to pick test examples that reflect what you ultimately want to perform well on, rather than whatever data you happen to have for training. The purpose of the dev and test sets are to direct your team toward the most important changes to make to the machine learning system. The dev and test sets allow your team to quickly see how well your algorithm is doing.
- another example[1]: 70% training set, 15% dev set, and 15% test set. Because, if you have data, you don't want to spend only 60% on training, like the previous example. There is no need to have excessively large dev/test sets beyond what is needed to evaluate the performance of your algorithms. Dev sets with sizes from 1,000 to 10,000 examples are common. Your dev set should be large enough to detect meaningful changes in the accuracy of your algorithm, but not necessarily much larger. Your test set should be big enough to give you a confident estimate of the final performance of your system.

Concrete example

[1] https://www.youtube.com/watch?v=F1ka6a13S9I&t=421s&index=2&list=WL

- 50,000 hours of some audio data form some distribution. Then, we also came up with different 10 hours of audio data, from other distribution.
- It is a bad idea to use these 10 hours of data for testing set. Rather, use half of that 10 hours **for dev set** and half of that for **test set**. Then, from these 50,000 hours, cut 20 hours for **train-dev set**, and train a model on data consisting of the rest 49,980 hours. So, **dev, train-dev, and test set are from the same distribution**, but training set is not.
- **There is no need for dev and test sets to come from the same distribution**. It is an important research problem to **develop learning algorithms that are trained on one distribution and generalize well to another**. **But if your goal is to make progress on a specific machine learning application rather than make research progress, I recommend trying to choose dev and test sets that are drawn from the same distribution.** This will make your team more efficient.
- Then, measure error on: training set, training-dev set, dev set, test set. Based on the values, we can estimate bias/variance and make further improvements.
- **BTW, never change your test set = it's your problem specification!!**
- When you are done developing, you will evaluate your system on the test set. If you find that your **dev set performance is much better than your test set performance**, it is a sign that you have **over-fit to the dev set. In this case, get a fresh dev set.**
- Choose dev and test sets from a distribution that reflects what data you expect to get in the future and want to do well on. This may not be the same as your training data's distribution.
- If your dev set and metric are no longer pointing your team in the right direction, quickly change them: (i) If you had over-fit the dev set, get more dev set data. (ii) If the actual distribution you care about is different from the dev/test set distribution, get new dev/test set data. (iii) If your metric is no longer measuring what is most important to you, change the metric.

Bad results for prediction

Be very careful, because you can spend few months on "randomly" picking one of these below, and will end up with almost no result. What you should do, is to implement some diagnostic test for measuring a performance of some learning algorithm. This can take time to implement, but it is worth it.

- Get more training samples -> fix **high variance**.
- Reduce a number of features -> fix **high variance** (lowering features to lower down high bias does not help).

- Increase a number of features -> fix **high bias** (current hypothesis is too simple, so after few more features, it better fits to the training set).
- Try to add polynomial features. What polynomial degree? Try more variants, for example up to degree = 10, and evaluate a cost function on evaluation dataset.
  - Note - if we evaluate what degree is the best on test set, this is an optimistic estimate of generalization error. Because I fit this parameter on test set and it is no longer fair to evaluate a hypothesis on this test set. So it would do better on this test set than on new examples a model has not seen before.
  - Fix **high bias** (current hypothesis is too simple, so after adding more polynomial features, it better fits to the training set).
- Try to increase regularization -> fix **high variance**.
- Try to decrease regularization -> fix **high bias**.

Overfitting

- =**high variance** - if a model is too complex (too many features/parameters and too less data) and it overreacts on even a small fluctuations in training data.
- On graph - performs very good on training data, but there are unnecessary curves.
- High variance can be observed from learning curve.
- Performs very good during learning, but bad with previously unseen data.
- Solution:
  - A bigger dataset should help.
  - Reduction of a number of features:
    - use a model selection algorithm, or
    - manually select which features to keep.
  - Regularization - all features are kept, we all consider to be good and important. Reduce magnitude/values of parameters θ_j.
  - Perform a statistical significance test - for example chi-square, before adding new structure, to decide whether the distribution of a class really is different with and without this structure.
  - **Learning algorithms with a high-variance** profile can generate arbitrarily **complex models** which fit data variations more readily. Examples of high-variance algorithms are **decision trees, neural networks** and **SVMs**. The obvious pitfall of high-variance model classes is overfitting.

Underfitting

- = **high bias** - if a model does not fit training data very well. This is caused by too few features or some function is too simple on a given problem.
- High bias can be observed from learning curve.
- Bigger dataset is not enough. With high bias, the model is not fitting the training data currently present, so adding more data is unlikely to help.
- A "compromise" can be seen as well[2].
- **Learning algorithms with a high-bias** profile usually generate **simple**, highly constrained **models** which are quite insensitive to data fluctuations, so that variance is low. **Naive Bayes** is considered to have high bias, because it assumes that the dataset under consideration can be summarized by a single probability distribution and that this model is sufficient to discriminate between classes.

Data mismatch

- The algorithm generalizes well to new data drawn from the same distribution as the training set, but not to data drawn from the dev/test set distribution. This is because the training set data is a poor match for the dev/test set data. To address this, you might try to make the training data more similar to the dev/test data.
- **Example**: Suppose you have developed a speech recognition system that does very well on the training set and on the training dev set. However, it does poorly on your dev set. Recommended is to:
  1. Try to **understand** what **properties of the data differ between the training and the dev set distributions by manual examination** - the purpose of the error analysis is to understand the significant differences between the training and the dev set, which is what leads to the data mismatch.
  2. **Try to find more training data** that better **matches the dev set** examples that your algorithm has trouble with.

Diagnosing bias vs variance

- Below are 2 examples of how the problem could look like on plot. In reality, the graphs can be a bit more messy and just a little bit more noisy. Selecting the most optimal point ("just fine") between bias and variance, can be done manually or automatically.

[2] 'Just fine' is something between underfitting and overfitting.

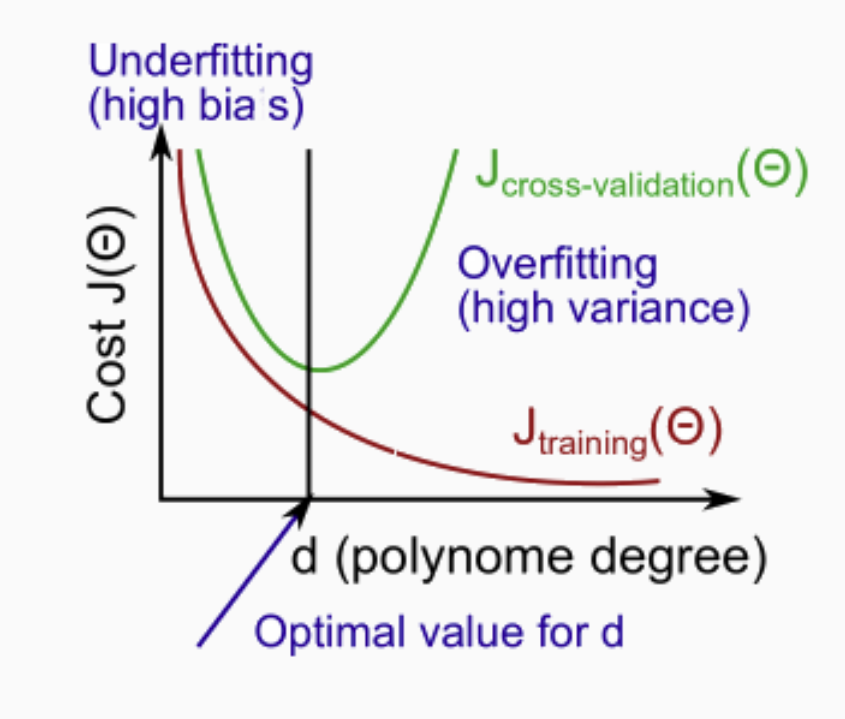


Figure 12.1: An example of high bias vs high variance during choosing a best candidate of polynome degree d. Hyperparameter d is for estimating a polynome degree of linear regression. During high bias, an error is very high for both train and validation set. During high variance, training error is low because it overfitted the training set, but a model is performing poorly on validation set. Cross validation error is usually a bit higher than error on training set.

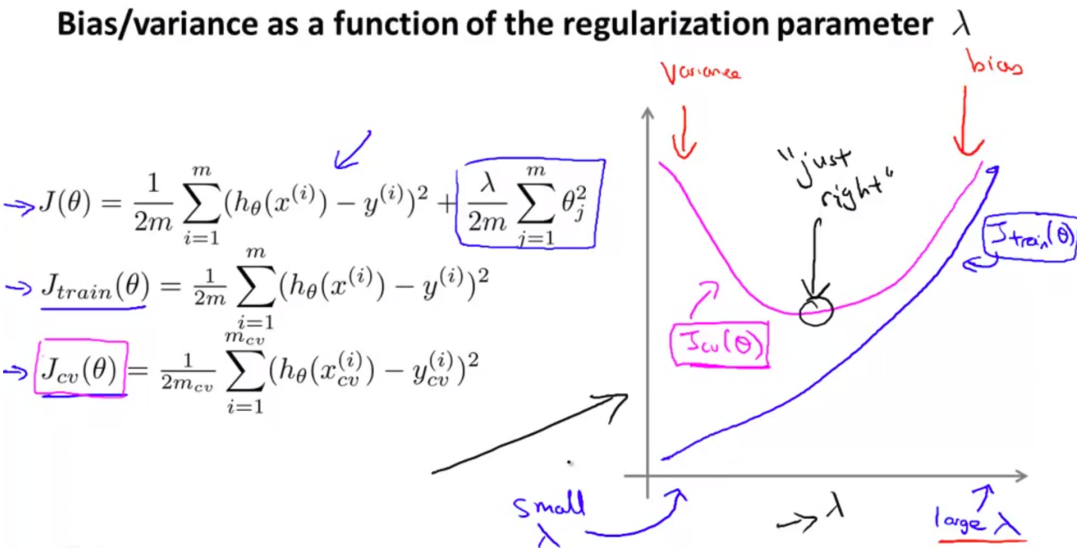


Figure 12.2: Another example of high bias vs high variance during choosing the best candidate of regularization parameter λ. The graph is different in comparison to the previous one, because we are dealing with different hyperparameters. Note: in the first formula, $J(\theta)$, the SUM in the regularization should be from 1 to n, not m - that is just a typo.

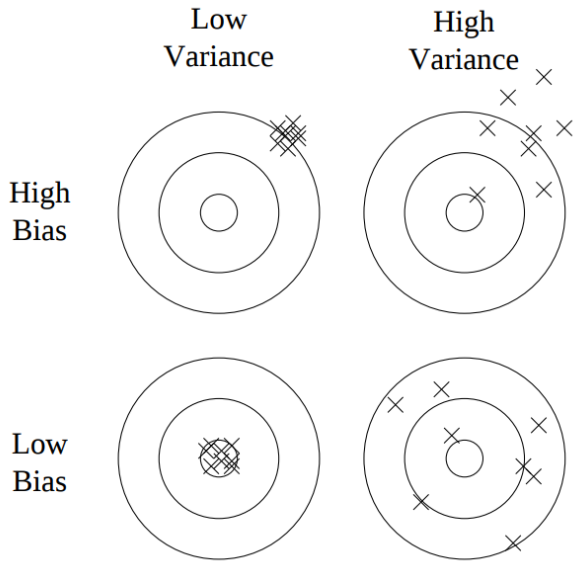


Figure 12.3: Bias and variance in dart-throwing.

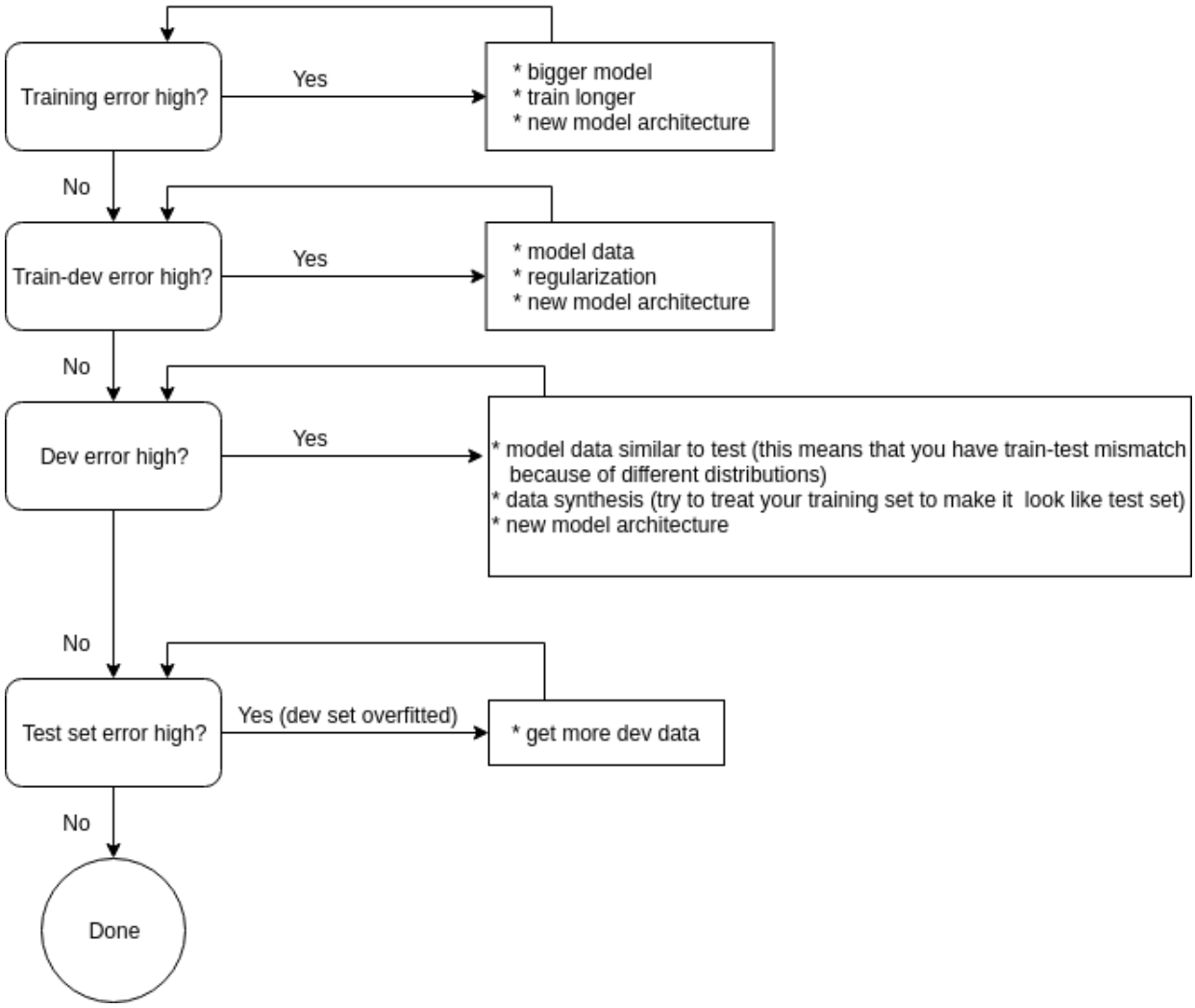


Figure 12.4: An example of responding to a training a model with alternatives solving high bias and high variance. 4 datasets were used - **training set** (data that the algorithm will learn from, not necessary from the same distribution as dev/test set), **training-dev set** (same distribution as training set, usually smaller than training set - it is usually large enough for evaluation and tracking progress of learning algorithm), **dev set** (same data distribution as test set), **test set** (data we ultimately care about doing well on). Now we can evaluate training error, the algorithm's ability to generalize to new data drawn from the training set distribution, by evaluating on the training-dev set, and the algorithm's performance on a specific task by evaluating dev / test set..

Learning curves

- It is useful thing to plot - for either to **sanity checking that our algorithm is working correctly**, or if we want to **improve the performance of the algorithm**.
- It is recommended to **estimate an optimal error rate** (and to put it to a learning curve plot) - from, if possible, human-level performance.

  An example: diagnoses from x-ray images problem: a typical person with no previous medical background besides some basic training achieves 15% error on this task. A junior doctor achieves 10% error. An experienced doctor achieves 5% error. And a small team of doctors that discuss and debate each image achieves 2% error. In this case, it is recommended to use 2% as the human-level performance proxy for our optimal error rate.

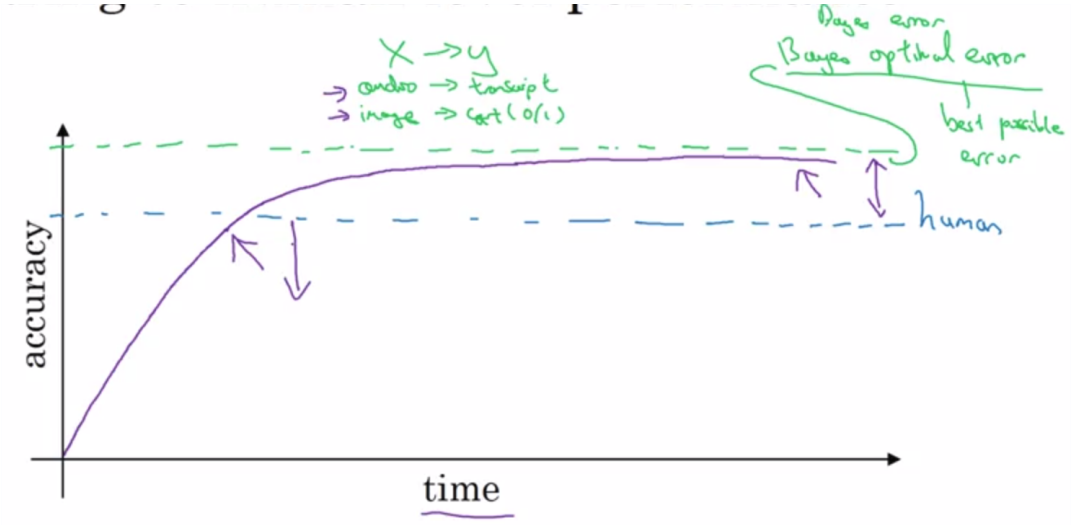


Figure 12.5: Comparing machine learning systems to human-level performance. We can see, that human level performance is not changing over time (as training continues), and that in theory, ML systems have upper bound (no matter how data and training time the algorithm has) called Bayes optimal error (or Bayes error, it is the best possible error, and there is no way for any function that it could surpass a certain level of accuracy). Also we can see, that progress seems to be relatively rapid as approaches to human level performance. After a while, the algorithm's progress actually slows down, but still increases. Why? Probably because human level performance can be relatively close to Bayes optimal error. For example, people are good at recognizing cats from a picture.

- **If your ML system is worse than humans**, it is recommended to:

  - get labeled data from humans
  - gain insight from manual error analysis - why a person get this right?
  - better analysis of bias / variance

- **Depending on what human level error is**, we can decide what to focus on:
  - **bias reduction** - training error is much bigger than human error
  - **variance reduction** - training error is similar to human level performance (this is called **avoidable bias** - calculated as a difference between training error and Bayes error - or human-level error, that can be similar), but there is a gap between training error and dev/test error

- Andrew Ng is plotting them quite often to determine if a learning algorithm is suffering from high variance or bias, or a bit of both.

- **Training error usually increases as the training set size grows** - for a model, it is more difficult to learn on newer and newer sample, and this is perfectly normal (for example, to remember - learn - 2 images of cats is easier than to learn 1,000 of images - there can be great variety, noise, and so on).

- Usually, with growing number of samples:
  - Scenario 1: **high bias**
    * for training set, the average error should grow. Because, it is more and more difficult to have a hypothesis which fits to our problem.
    * for validation set, the average error should decrease. Because more data we have, the better we do at generalizing to new examples.
    * these two curves will end up **close to each other**. Ultimately, the performance on both validation and training set will be very similar. And quite high, so that's why it is called high bias.
    * **getting more data usually does not help much!**

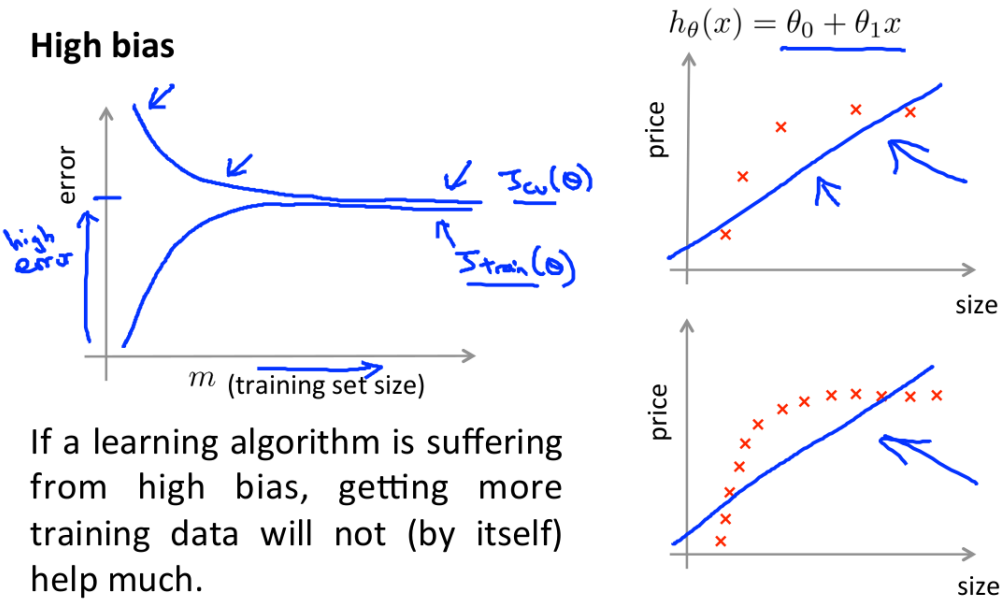


Figure 12.6: An example of high bias shown on learning curve.

– Scenario 2: **high variance**
  * for training set, the average error should grow. Because, it can be a bit harder to fit dataset correctly if we have more and more data. However, the training error will be still pretty low (in comparison to situation in Scenario 1).
  * for validation set, the average error should remain high, or perhaps a slightly be decreasing, even if we add more and more validation data.
  * high variance can be identified that there is a **large gap between those 2 curves**.
  * **getting more data is likely to help**, since these 2 curves are converging to each other.

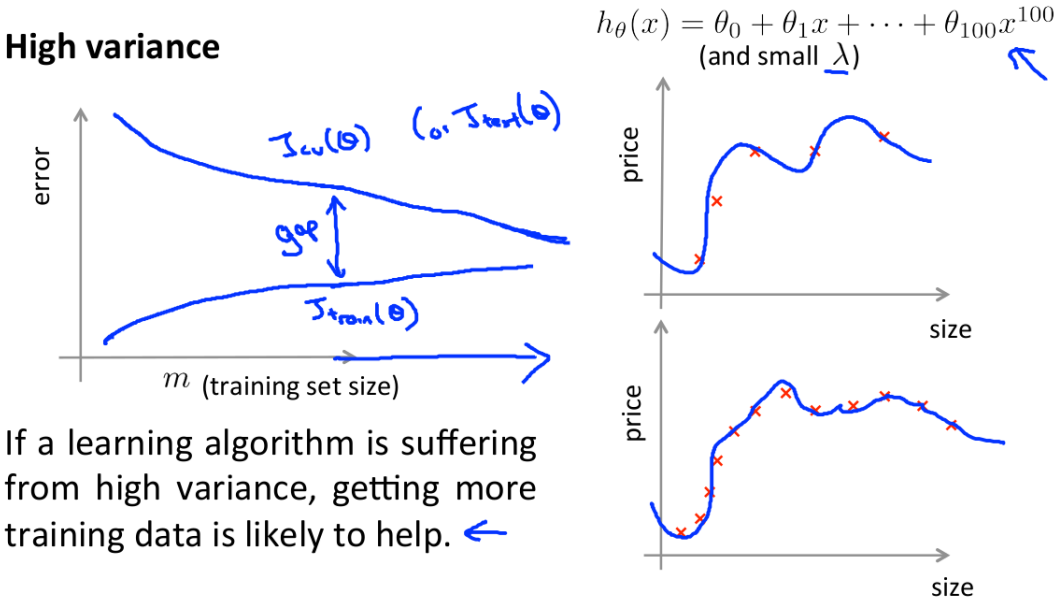


Figure 12.7: An example of high variance shown on learning curve.

Beginning & fast prototyping

Ideal case - start quick & dirty the easiest solution possible, see if that works. Give it 1 day, max 24hours.

- Simple algorithm that is easy to implement. Test it on cross-validation data.
- Draw learning curves and find out if it is needed to add more features, more data, or so.
- Then - error analysis: why the results are wrong, see manually concrete samples why they are misclassified / wrong. And by this time, create new features.
  - Error analysis on cross-validation data - if we create a new feature, we can end up that we choose features which works good on specific test set (=generalization of overall solution to newer examples). Be careful. In other words, if we develop new features by examining the test set, then we may end up choosing features that work well specifically for the test set, so $J_{test}(\theta)$ is no longer a good estimate of how well we generalize to new examples.
  - Test set should be TOTALLY SEPARATED and used on the very end.
- Then, having a numerical evaluation is a good thing - single number (floating point) which tells a performance of an algorithm. So - we have an idea - quickly try, rerun algorithm and see this number (F-score for instance). If this number is higher or lower, we can say if the idea was good or bad, no need to manually examining.
- CNN - start fitting on small number of hidden layers, scale-up to overfitting takes over. Then use regularization (or another way how to deal with overfitting), and then scale-up again.

Hyperparameter tuning

Note: In deep learning, the learning rate α is one of the most important hyperparameter to start with.

Note2: Once you find the best values of hyperparameters, you use the entire training set to build the model with these, and then you measure the performance of your model using the test set.

- There exist a lot of approaches to this problem:
  - **Grid search**
    - The most simple technique, that uses all possible combination of the input hyperparameters you specify, and find the best combination.
    - If there is a big gap between some hyperparameter values (range of possible values you want), it is more recommended to generate values with **logarithm scale** ($0.0001, 0.001, 0.01, 0.1, 1$ is better than using a linear "step" by 0.1 for instance). Once you find the best combination of hyperparameters, you can explore the values close to the best ones in some region around them. And then, finally you assess the selected model using the test set.
    - It can be used when you have enough data to have decent validation set (in which each class is represented by at least a couple of dozens examples), and the number of hyperparameters and their range is not too large. When you don't have a decent validation set, the common technique you can use is called **cross-validation**. However, you can also use for example grid search with cross-validation to find the best values of hyperparameters for you model.
  - **Random search**
    - If a number of hyperparameters is small, it is possible to use a grid and try all the combinations. Otherwise - and this is better and more recommended approach - use randomness (generate random values), or other methods such as Bayesian hyperparameter optimization.
    - Here, you no longer provide a discrete set of values to explore for each hyperparameter. Instead, you provide a **statistical distribution for each hyperparameter** from which values are randomly sampled and set the total number of combinations you want to try.
  - **Bayesian hyperparameter optimization**
    - Bayesian techniques use past evaluation results to choose the next values to evaluate.
    - The idea is to limit the number of expensive optimizations of the objective function by choosing the next hyperparameter values based on those that have done well in the past.

  - There exist also other techniques such as **gradient-based techniques**, or **evolutionary optimization techniques**.

Batch normalization

- This makes your hyperparameter search problem much easier and makes your neural network much more robust.
- Normalizing inputs to speed up learning causes, that learning algorithm converges faster. This can be applied not just to input features, but also to input of each neuron, so we can perform normalization of input of each hidden layer (=activations $a^{[l]}$, or before activations $z^{[l]}$ - there are works that compares both approaches. However, to normalize values before activations is done more often).
- In practice, batch normalization is usually applied on mini-batches of samples. So, you are normalizing a layer by layer on the whole mini-batch of samples. In neural network libraries, you can often insert a batch normalization layer between two layers.
- Batch normalization reduces **co-variate shift problem**. If values of parameters change, at least mean and variance stays the same. It limits the amount to which updating the parameters in the earlier layers can affect the distribution of values that the n layer now sees and therefore has to learn on. And so, batch normalization reduces the problem of the input values changing, it really causes these values to **become more stable**, so that the later layers of the neural network has more firm ground to stand on. And even though the input distribution changes a bit, it changes less, and what this does is, even as the earlier layers keep learning, the amounts that this forces the later layers to adapt to as early as layer changes is reduced or, if you will, it weakens the coupling between what the early layers parameters has to do and what the later layers parameters have to do. And so it allows **each layer of the network to learn by itself, a little bit more independently of other layers**, and this has the effect of **speeding up of learning in the whole network**.
- Another effect of batch normalization is a **small regularization effect** (when using mini-batch training). Each mini-batch is scaled by the mean and variance computed on just that mini-batch. This adds some noise to the values $z^{[l]}$ with that mini-batch, because it is not done on the whole training set (standard deviation and mean are noisy). So, similar to dropout, it adds some noise to each hidden layer's activations. Because by adding noise to the hidden units, it's forcing the downstream hidden units not to rely too much on any hidden unit. This regularization effect is small and it is fine to use also Dropout. If you use a bigger mini-batch size, you are reducing this noise and therefore also this regularization effect.

- **At test time** (predictions), it is needed to have slightly different approach - because you are computing mean and standard deviation on a 1 example, instead of a mini-batch of samples. So it is needed to compute these 2 values differently - using **exponentially weighted average**, where the average is across the mini-batches. Therefore it is needed to keep track of all values on all hidden layers as well as it is needed to keep a track of the current (exponentially weighted) average of mean values and also the same for standard deviation. Then, during normalization at test time, these 2 values are used (and you will use batch norm parameters γ and β learned during training process).

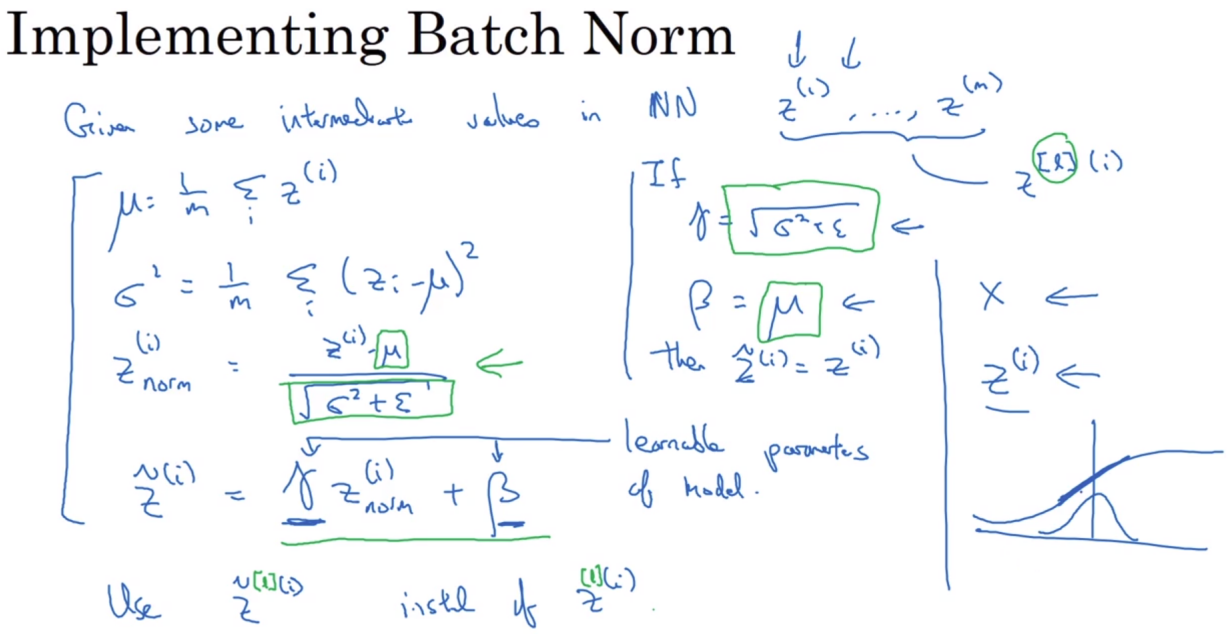


Figure 12.8: Batch normalization of a single neuron. Values β and γ set the mean and variance of the linear variable $z^{[l]}$ of a given layer. They can be learned using Adam, Gradient descent with momentum, or RMSprop, not just with gradient descent. Batch normalization is about normalizing inputs to each hidden layer = so, normalization of activations or before applying activation function - in practice, the second option is more used.

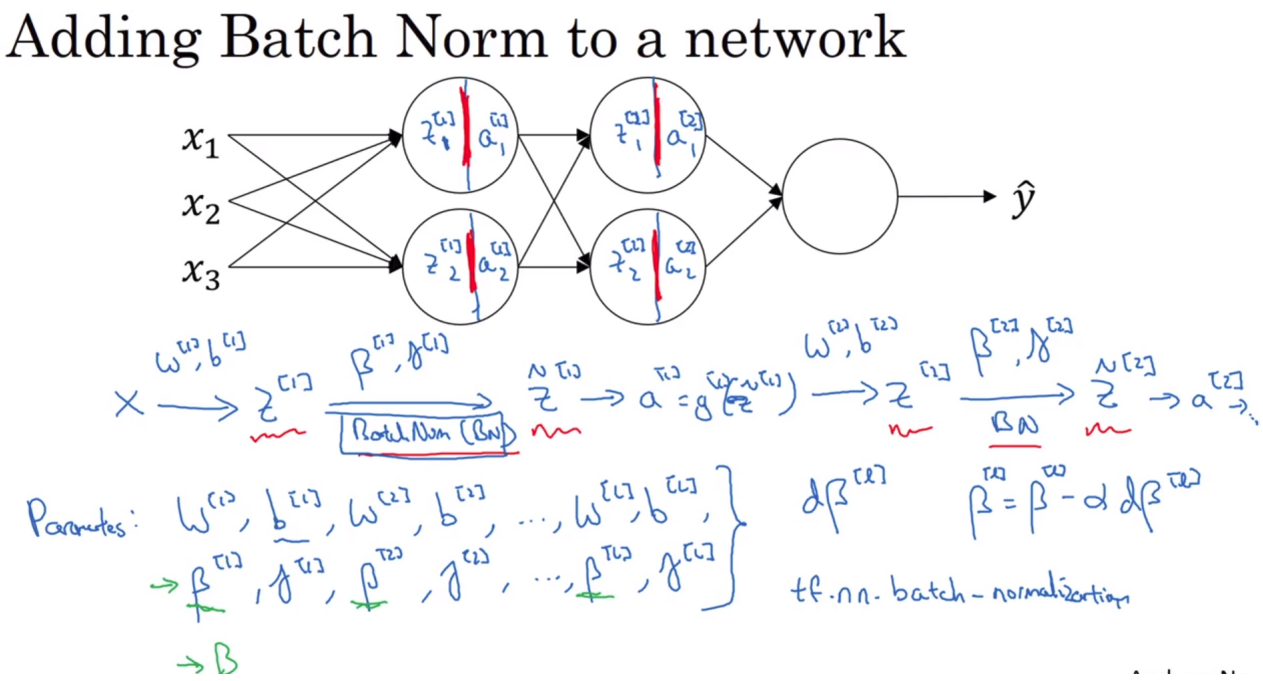


Figure 12.9: Batch normalization of the whole ANN.

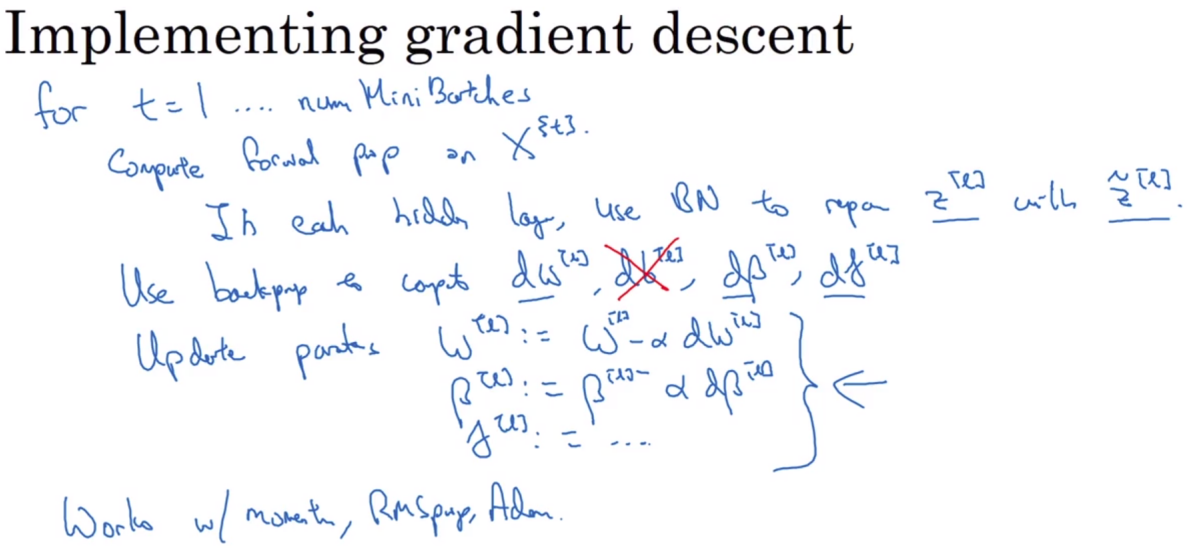


Figure 12.10: Gradient descent implementation using batch normalization with mini-batches. Batch normalization also works with Adam and RMSprop and so on.

- As been detailed in the figures above, we can set different variance and mean for normalized values. However, having $\mu = 0$ and $\sigma^2 = 1$ is not always wanted for all hidden layers. It is needed for input features, but in general - if we would do this, and we would have sigmoid activation function, then it would be in linear regime (for this to see, please see sigmoid activation function how it looks, and focus on values between -0.5 and 0.5). We want to take advantage of non-linear regimes of the sigmoid activation function, which means $\mu \neq 0$.

Transfer learning

- Information mostly from here[3] and also see tips and a list of learned models here[4].
- Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. So basically we use pre-trained models from other people by making small changes.
- Do not re-invent the wheel!
- Be careful what pre-trained model you use. The problem has to be similar. For example, a model previously trained for speech recognition would work horribly if we try to use it to identify objects using it.
- Since we assume that a pre-trained model (DNN for instance) has been trained quite well, we would not want to modify the weights too soon and too much. While modifying we generally use a learning rate smaller than the one used for initially training the model.
- There are more ways to fine tune the model:
  - Feature extraction. We can use a pre-trained model as a feature extraction mechanism. What we can do is that we can remove the output layer, and then use the entire network as a fixed feature extractor for the new data set.
  - Use the architecture of the pre-trained model. We can use architecture of the pre-trained model while we initialize all the weights randomly and train the model according to our dataset again.
  - Train some layers while freeze others. This means that we will train the pre-trained model only partially by keeping the weights of initial layers of the model frozen while we retrain only the higher layers. We can try and test as to how many layers to be frozen and how many to be trained.
- How to proceed on using pre-trained model regarding the previous ways, we can have a few scenarios; mostly relevant to DNN, because they are very resource-intensive (and can be found pre-trained and freely available on the Internet):
  - Size of **dataset is small** while **data similarity is very high.** Since data similarity is high, we do not need to re-train the model. All we need to do is to customize and modify the output layers according to our problem statement. We use the pre-trained model as a feature extractor.
  - Size of **dataset is small** as well as **data similarity is very low.** In this case we can freeze the initial k layers of the pre-trained model and train just

[3] https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/
[4] https://towardsdatascience.com/deep-learning-tips-and-tricks-1ef708ec5f53

the remaining layers again. The top layers would then be customized to the new data set. Since the new data set has low similarity it is needed to re-train and customize the higher layers according to the new dataset. The small size of the data set is compensated by the fact that the initial layers are kept pre-trained (which have been trained on a large dataset previously) and the weights for those layers are frozen.

- Size of **dataset is large** however **data similarity is very low.** Since we have a large dataset, our neural network training would be effective. However, data we have are very different as compared to data used for training of pre-trained model. The predictions made using pre-trained models would not be effective. Hence, **it is best to train the neural network from scratch according to your data**.
- Size of **dataset is large** as well as **data similarity is high.** This is ideal situation, pre-trained model should be most effective. keep weights and architecture the same, and retrain this model using the weights as initialized in the pre-trained model.

- An example of one approach to identify handwritten digits:
  1. Retrain the output layers only - we just train the weights of these layers and try to identify the digits.
  2. Freeze the weights of the first layers - this is because the first few layers capture universal features like curves and edges that are also relevant to our new problem. We want to keep those weights intact and we will get the network to focus on learning dataset-specific features in the subsequent layers.

Multi-task learning

- **In comparison to transfer learning**, in which there is a sequential process where you learn from task A and then transfer that to task B, in multi-task learning you start off simultaneously, trying to have 1 ANN do several things at the same time. And then, each of these tasks help hopefully all of the other tasks. Also, activation function for the output layer also needs to be changed from softmax to something different. Softmax is a good choice if and only if one of the possibilities is present in each picture (so 1 of distinct classes).
- **For example**, let's have a deep neural network, and output layer computes a vector of 4 attributes - if there is a pedestrian, car, stop sign, or traffic lights on a given picture. So it is not a vector where only 1 of these can be detected, and other ones not - such ANN can perform multiple "tasks" at the same time (also, our annotation in training set is represented by vector of 4 values)
- **Multi-task learning makes sense when:**
  - You are training on a set of tasks that could benefit from having shared lower-level features. As in the example above, it makes sense that recognizing traffic lights and cars and pedestrians, those should have similar features that could also help you to recognize stop signs, because these are all features of roads.
  - (Usually): amount of data you have for each task is quite similar. In comparison to transfer learning, a task A has usually a lot of data (let's say 1M), and task B has small amount of data (let's say 1k). In multi-task learning, let's say that we have 100 tasks and each has 1k examples. If we would want to do 1 task in isolation, you would have only 1k examples to train with, which is not much. However, by training all tasks simultaneously, you have 100k of training examples which could be a big boost because they could give a lot of knowledge.
  - You can train a big enough ANN to do well on all the tasks.
- An alternative is to train a separate model for each task. "Rich Carona, found many years ago was that the only time a multi-task learning hurts performance, compared to training separate neural networks, is if your neural network isn't big enough."
- In practice, multi-task learning is used much less often than transfer learning. Often it is just probably difficult to set up or to find so many different tasks that you would actually want to train a single neural network for.

Online learning

- Large-scale machine learning setting, which allows us to model problems where we have a continuous flood or stream of data coming in and we would like to have an algorithm to learn from that.
- So, we update parameter θ_j by one example after another (like in SGD). However, we use that one example, and we never use it again, we throw it away. But remember, that online learning is mostly if we have really a continuous stream of data, basically unlimited. If we have a website with a lot of online users, it is a good choice. If we don't have a lot of users, maybe it is not a good idea.
- Online learning is adaptable to changes. For example, thanks to online learning, a model can adapt due to some change in behavior of users which are coming to your website and you have data from them, let's say because of some change in economic situation.

Map Reduce and Data Paralelism

- Not all machine learning tasks can be done on just 1 machine. In some bigger tasks, we need to paralyze. For some problems, even SGD is not enough and this map reduce can be very helpful.
- If we have n machines, we split (uniformly) our training data into n parts. On each machine we perform training with Batch gradient descent (for instance) on m/n training samples, where m is the number of all training samples we have. After all n computers are finished (so we speed up the training almost by n times - just almost, because there are network latencies, overhead of combining temporary results, and so on), they will send these intermediate results to 1 centralized master server. This server will combine the results of these temporary results of SGD together: $\theta_j = \theta_j - \alpha\frac{1}{m}(temp_j^{(1)} + ... + temp_j^{(n)})$ which is equal to a simple Batch gradient descent on 1 computer.
- To use map reduce, we have to have a learning algorithm that uses a summation over the training set. And fortunately, many learning algorithms can be expressed as computing sums of some functions over the training set.
- It is possible to take a benefit of this approach on 1 computer, if we have multiple CPU cores.

Anomaly detection

- **We have small number of positive examples, and a bigger number of negative examples.** We have a big amount of anomaly types, it is difficult to find an algorithm which would learn from positive examples about how anomaly looks like and new ones can come.

- **We learn model basically what is normal. Everything else (beyond some threshold for example) is considered to be an anomaly.**
- The algorithm(s) can use a set of features for training (can be from just one, bigger class) in Gaussian distribution. If values in some feature are not Gaussian, then we can transform them to be in one. An example is depicted on the next figure.

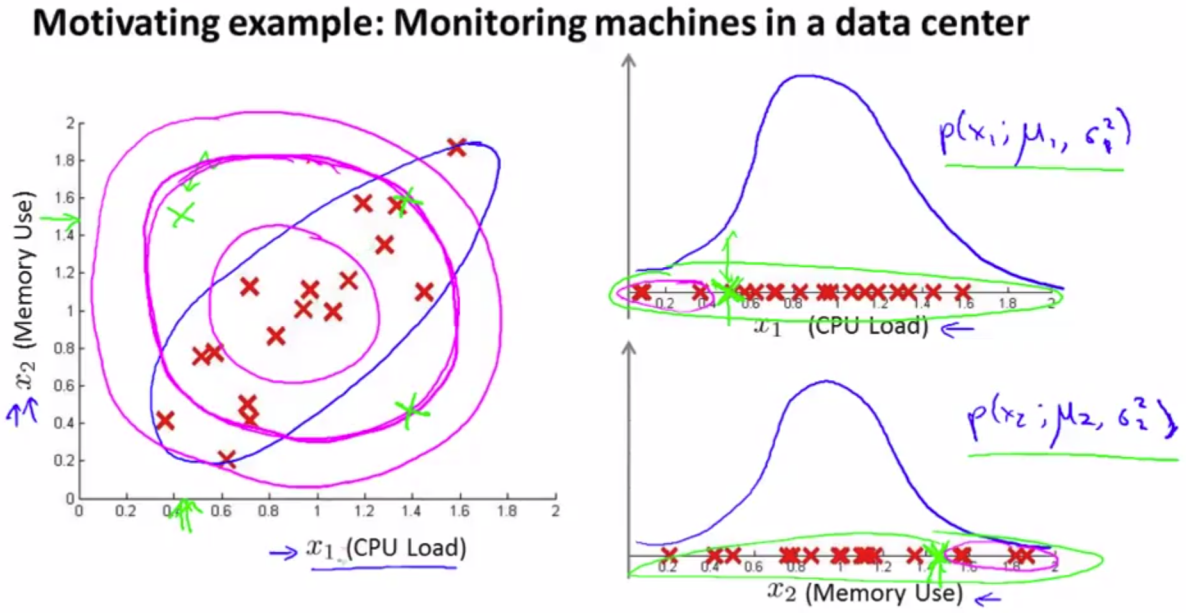


Figure 12.11: An example of anomaly detection algorithm with 2 features. All the green points are approximately equal in the terms of (low) probability of being an anomaly, even to the fact that they are really distant and placed somewhere else. What is inside of the blue (left) curve, that should be normal activity. Green ones are anomalies, but the algorithm cannot see it that way. This simple anomaly detection algorithm is not realizing that this blue ellipse shows the high probability region. To fix this problem, we can use for example Multivariate Gaussian Distribution.

- **Multivariate Gaussian Distribution**
  - idea: we have $x \in \mathbb{R}^n$. Don't model $p(x_1), p(x_2), ...$ separately, but model $p(x)$ all in one go (together).
  - parametrized by mean $\mu \in \mathbb{R}^n$ and covariance matrix $\Sigma \in \mathbb{R}^{n\,x\,n}$ and we also need determinant $|\Sigma|$.

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)) \tag{12.1}$$

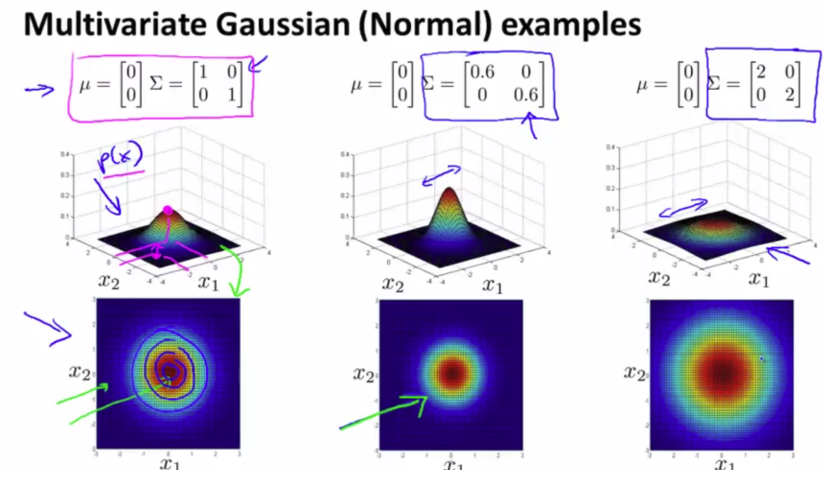


Figure 12.12: Example of multivariate gaussian distribution when changing values of covariance matrix (I).

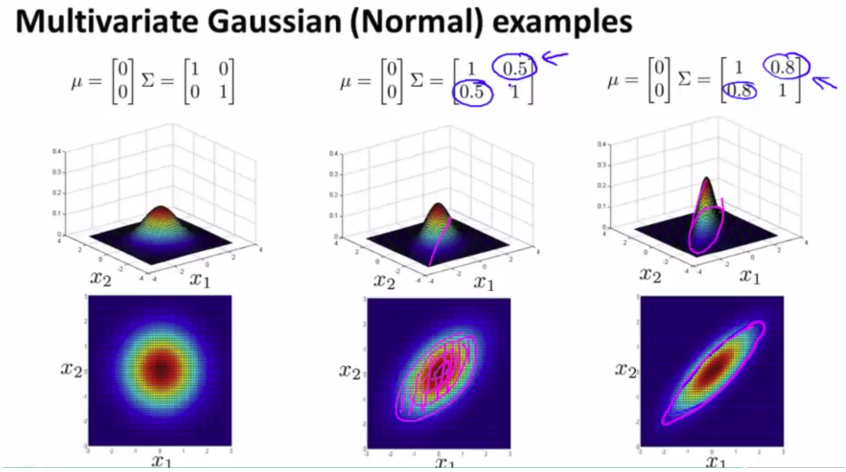


Figure 12.13: Example of multivariate gaussian distribution when changing values of covariance matrix (II).

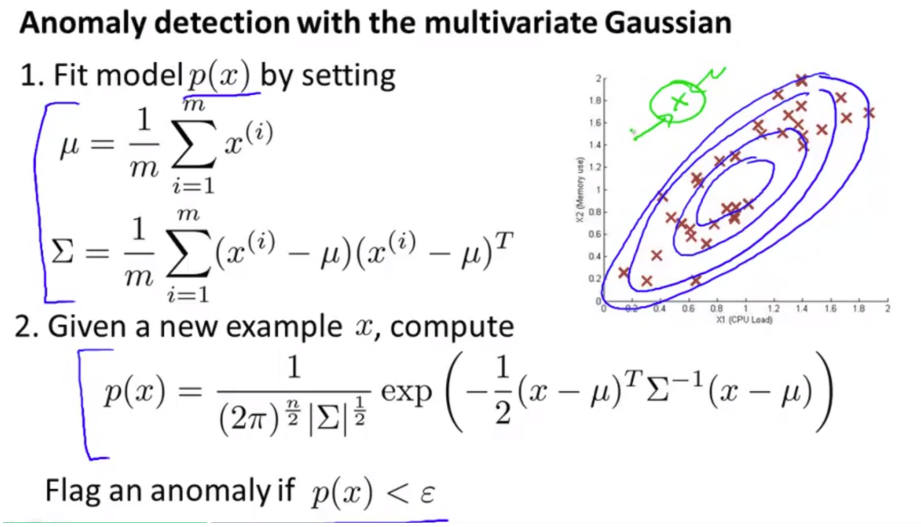


Figure 12.14: Algorithm of anomaly detection with multivariate gaussian distribution. Sample 'X' (green) which is 'outlier' is correctly considered to be an anomaly, because now with multivariate gaussian distribution the shape of elipsis is different, it is not circle like on the very first figure in this subsection. And this elipsoid shape, determined by μ and Σ, can be fit from data.

- **Anomaly detection vs supervised learning** - big amount of negative as well as positive examples. There is enough positive examples for algorithm to learn from them, the future unseen ones are similar. Examples: email spam classification, weather prediction, cancer classification.
- **Examples** - fraud detection (but this can be also Supervised Learning), or monitoring machines in data center. Another example of usage the data: let's say that we have 10,000 negative examples and 20 positives. Now we will use for training the model 6,000 negative examples and 0 positives. And then for validation (threshold parameter estimation for example) and test set 10 and 10, and 2,000 and 2,000 samples.

Working with Big Datasets

1. **Usage of Batch vs Stochastic gradient descent** (see Types of Gradient descent in Section 1.9).
2. **Map-Reduce principle and data parallelism.** Training set is uniformly divided to few parts and then the results are send to some master server, which will combine them.

Acquiring even more data - increasing training set

- Syntesis of artificial data - for example, usage of GANs.
- Gather (and label) data manually or via crows sourcing (Amazon Mechanical Turk).
- **Example:**
  - Your speech system needs more data that sounds as if it were taken from within a car. Rather than collecting a lot of data while driving around, there might be an easier way to get this data: by artificially synthesizing it.
  - However, keep in mind that artificial data synthesis has its challenges: it is sometimes easier to create synthetic data that appears realistic to a person than it is to create data that appears realistic to a computer. For example, suppose you have 1,000 hours of speech training data, but only 1 hour of car noise. If you repeatedly use the same 1 hour of car noise with different portions from the original 1,000 hours of training data, you will end up with a synthetic dataset where the same car noise is repeated over and over. While a person listening to this audio probably would not be able to tell – all car noise sounds the same to most of us - t is possible that a learning algorithm would "over-fit" to the 1 hour of car noise. Thus, it could generalize poorly to a new audio clip where the car noise happens to sound different.
  - Alternatively, suppose you have 1,000 unique hours of car noise, but all of it was taken from just 10 different cars. In this case, it is possible for an algorithm to "over-fit" to these 10 cars and perform poorly if tested on audio from a different car. Unfortunately, these problems can be hard to spot.
- **When synthesizing data, put some thought into whether you're really synthesizing a representative set of examples.** Try to avoid giving the synthesized data properties that makes it possible for a learning algorithm to distinguish synthesized from non-synthesized examples - such as if all the synthesized data comes from one of 20 car designs, or all the synthesized audio comes from only 1 hour of car noise. This advice can be hard to follow. If you are able to get all the details right, you can suddenly access a far **larger training set** than before.
- **Data augmentation**
  - This is a technique that aims at improving the performance of for example a computer vision systems (mostly?) by creating new data samples from existing ones. For example, mirroring of an image, multiple random crops, playing with RGB colors, shifting, and so on.
  - We can use this when we are overfitting and the best solution is to add more data, but that would be too expensive.

Weighting Data

Example of problem and a solution:

- Suppose you have 200k images from the Internet and 5k images from your mobile app users. So there is a 40:1 ratio between the size of these datasets. In practice, having 40x as many internet images as mobile app images might mean you need to spend 40x (or more) as much computational resources to model both, compared to if you trained on only the 5k images.
- If you don't have huge computational resources, you could give the internet images a much lower weight as a compromise.
- Suppose your optimization objective is squared error (not good for classification, but it's simple for this example), where the first sum is over 5k examples and the second one over 200k examples:

$$min_\theta \sum_{(x,y)\in MobileImg} (h_\theta(x) - y)^2 + \sum_{(x,y)\in InternetImg} (h_\theta(x) - y)^2 \tag{12.2}$$

- Instead, it is better to use additional parameter β . If you set $\beta = 1/40$, then the algorithm would give equal weight to mobile images and internet images. So you don't have to build a massive ANN to make sure that the algorithm does well on both types of tasks.:

$$min_\theta \sum_{(x,y)\in MobileImg} (h_\theta(x) - y)^2 + \beta \sum_{(x,y)\in InternetImg} (h_\theta(x) - y)^2 \tag{12.3}$$

- This type of re-weighting is needed only when you suspect the additional data (Internet images) has a very different distribution than the dev/test set, or if the additional data is much larger than the data that came from the same distribution as the dev/test set (mobile images).

Neural Networks Problems

- 37 Reasons why your Neural Network is not working[5]

Outlier Detection

- Outlier detection is the problem of detecting such examples from the dataset, that are very different from what a typical example in the dataset looks like.

[5] https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607

- There are several techniques that could help to solve this problem, for example autoencoder and one-class classifier learning.
  - If we use an autoencoder, we train it on our dataset. Then, if we want to predict whether an example is an outlier, we can use the autoencoder model to reconstruct the example from the bottleneck layer. The model will unlikely be capable of reconstructing an outlier.
  - In one-class classification, the model either predicts that the input example belongs to the class, or it's an outlier.

Performance - Micro vs Macro average methods

- **Micro-average**: individual TP, FP and FN of the system are summed for different sets and applied to the statistics. Micro-averaging may be preferred in multi-label settings, including multi-class classification where a majority class is to be ignored. It will aggregate the contributions of all classes to compute the average metric.
- **Macro-average**: (=classes are equal) more straight forward, just take the average of the precision and recall of the system on different sets. So each metric for each class and then average.
- **An alternative**, when classes are imbalanced, is to compute a weighted macro-average, in which each class contribution to the average is weighted by the relative number of examples available for it.

Classification on unbalanced data

There exist 4 approaches how to tackle this problem in literature (at least one such taxonomy)[6]. They are described in this section. Some algorithms are less sensitive to this problem, such as decision trees (also random forest or gradient boosting).

- **Data pre-processing**
  - **Re-sampling** - the idea is to modify the class distribution so that the proportion of instances of each class is balanced. There are different types:
    - **Under-sampling** - randomly select a group of **majority class** and **removes** them from training set. Smaller datasets are being generated, so learning algorithms are more time and memory efficient at the cost of losing information.
    - **Over-sampling** - randomly select a group of **minority class** instances and **duplicates** them. No information is lost, but on the other hand, it might lead to overfitting.

[6] https://dl.acm.org/citation.cfm?id=2907070 or https://arxiv.org/pdf/1505.01658.pdf

  - **Active learning** - in traditional way, active learning approach is used to select the unlabeled instances for an expert to annotate it. In imbalanced learning, this means selecting instances that are difficult to classify, those that are close to the boundaries to construct a classifier there. Afterwards, it is usually combined with another classification method, such as SVM. Problems - computational cost related to identifying the most informative instances. These methods are usually ot included in the existing algorithm comparison and reviews.
  - **Weighting the data space** - give a higher weight to examples of the minority class. This is related to cost-sensitive learning.

- **Special-purpose learning Methods** - instead of modifying the underlying data, many research efforts have focused on algorithm modifications. The goal is to put the bias towards the minority class into the learning process so that it penalizes the errors on the minority class more. Many of the existing work is based on the foundation of cost-sensitive learning, building on the concept of cost matrix, sometimes utilizing ensemble techniques; some other techniques are based on editing kernel functions.
  - **Cost-sensitive learning methods** penalize the misclassification of minority class instances more than the instances from the majority class. To achieve this, they utilize the cost matrix, which contains penalties/costs for all correct and wrong classifications. The concept is applicable not only for imbalanced data but in any case where one wants to express that one class is more important than another. Also, it is not only limited to binary classification.

- **Prediction Post-Processing**
  - **Threshold method** - easy method to deal with the imbalance after the training the classifier. Having a scoring classifier, it is possible to apply different threshold values and thus move on the ROC curve. Then, it is important to select the threshold where the error (e.g. cost error defined by the cost matrix) is minimal.
  - **Cost-sensitive post-processing** - one base learner is used to train the classifier, and then some meta-learning method (MetaCost for example) that operates only with predicted probabilities of the base learner (i.e. it post-processes the probabilities) and it chooses the class with the minimum expected cost of the wrong prediction. Partially, this method can be considered as an ensemble method as it internally uses the aggregate of predictions across more samples of the data. But these methods are not very used in practice.

- **Hybrid methods**, combining **special-purpose learning methods** and **resampling.**

You might also try to create synthetic examples by randomly sampling feature values of several examples of the minority class and combining them to obtain a new example of

that class. There are two popular informed sampling algorithms that over-sample the minority class by creating synthetic examples:

- Synthetic Minority Oversampling Technique (**SMOTE**).
- Adaptive Synthetic Sampling Method (**ADASYN**).

SMOTE and ADASYN work similarly in many ways. For a given example x_i of the minority class, they pick k **nearest neighbors** of this example (let's denote this set of k examples S_k) and then create a synthetic example x_{new} as $x_i + \lambda(x_{zi} - x_i)$, where x_{zi} is an example of the minority class chosen randomly from S_k . The interpolation hyperparameter λ is a random number in the range $[0, 1]$. Both algorithms randomly pick all possible x_i in the dataset. Both SMOTE and ADASYN randomly pick all possible xi in the dataset. In ADASYN, the number of synthetic examples generated for each x_i is proportional to the number of examples in S_k which are not from the minority class. **Therefore, more synthetic examples are generated in the area where the examples of the minority class are rare.**

There exist a lot more algorithms that uses nearest neighbors principle - **Borderline-SMOTE**, **Edited Nearest Neighbourhood** (ENN), **Neighbourhood Cleaning Rule** (NCR), **Condensed Nearest Neighbours** (CNN), **One-sided Selection** (OSS), and others.

Learn many models, not just one (ensembles)

In the early days of machine learning, everyone had their favorite learner, together with some a priori reasons to believe in its superiority. Most effort went into trying many variations of it and selecting the best one. Then systematic empirical comparisons showed that the best learner varies from application to application, and systems containing many different learners started to appear. Effort now went into trying many variations of many learners, and still selecting just the best one. But then researchers noticed that, if instead of selecting the best variation found, we combine many variations, the results are better - often much better - and at little extra effort for the user.

Ensembles are now a standard.

Simplicity does not imply accuracy

- A learner with a larger hypothesis space that tries fewer hypotheses from it is less likely to over-fit than one that tries more hypotheses from a smaller space. Simpler hypotheses should be preferred because simplicity is a virtue in its own right, not because of a hypothetical connection with accuracy.
- As a rule, it pays to try **the simplest learners first** (e.g. Naive Bayes before logistic regression, k-nearest neighbor before SVM).

A brief comparison of supervised learning algorithms

- Generally, **SVMs** and **ANNs** tend to perform much better when dealing with **multi-dimensions** and **continuous features**.
- **Logic-based systems** tend to perform better when dealing with **discrete** or **categorical features**.
- For **ANNs** and **SVMs**, a **large sample size** is required in order to achieve its maximum prediction accuracy. **Naive Bayes** may need a **relatively small dataset**.

Traditional algorithms vs deep learning

- Even as you accumulate more data, usually the performance of older learning algorithms, such as logistic regression, "plateaus". This means its learning curve "flattens out," and the algorithm stops improving even as you give it more data.
- If you train a small neutral network (NN) on the same supervised learning task, you might get slightly better performance.

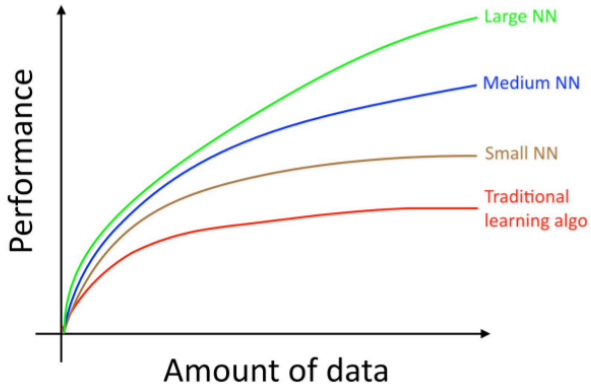


Figure 12.15: Traditional learning algorithms and artificial neural networks performance comparison.

Ceiling analysis

If you have a pipeline of multiple machine learning systems, it helps us decide on allocation of resources in terms of which component in a machine learning pipeline to spend more effort on. Because, if we spend too much time on some component, it does not mean that the overall system will be improved significantly. We can spend 1 year on improving background removal from an image, and then the overall system will be better by few %. In this case it may not be worth a significant amount of work improving, because even if it had perfect performance its impact on the overall system is small.

McNemar test

Problem - is 30 errors in 10k test examples significantly better than 40 errors?

- It depends on the particular errors!
- This test uses particular errors and can be much more powerful than a test that just uses the number of errors.

AI Transformation Playbook

This is from Andrew NG.[7] He recommend this to companies that want to become effective at using AI. Don't expect traditional planning processes to apply without changes; instead, work with AI team to establish timeline estimates, KPIs, milestones. It is iterative process! This transformation from a good company to good AI company can take 2 or 3 years. However, it is intended for companies with market cap from $500M to $500B. This is a 5 step program:

1. **Execute pilot projects to gain momentum**
   - More important for initial project is to succeed rather than be the most valuable. This can be outsourced or in-house.
   - The goal is also to gain familiarity with AI within the company and also persuade others to "join" and invest time and energy to AI as well.
   - You have to have a clearly defined and measurable objective that creates business value.
2. **Build an in-house AI team**
   - Build up an AI capability to support the whole company.
   - Execute an initial sequence of cross-functional projects to support different divisions / business units with AI projects. After completing the initial projects, set up repeated processes to continuously deliver a sequence of valuable AI projects.
   - Develop consistent standards for recruiting and retention.
3. **Provide broad AI training**
   - Not just to engineers, but also for managers, division leaders, and executives, and so on. It can be from couple of hours to even hundred (or more), depending on a role of person.
4. **Develop an AI strategy**

[7] https://landing.ai/ai-transformation-playbook/

- Leverage AI to create an advantage specific to your industry sector. Andrew Ng recommends this step to be fourth, not the first one. The reason is that firstly, a company needs to understand more about AI itself, and how to apply to a given business. After this, it is recommended to build an AI strategy. It is needed to also understand, what AI can and cannot do in a particular industry sector.
- Design such strategy, that will lead to a circle ... ->more data -> better product -> more users -> more data -> ...
- Consider also strategic data acquisition. Recognize what data is valuable, and what is not. And have ideally unified data warehouse.

5. **Develop internal and external communications**
   - Investor relations - make sure, that investor(s) value your company appropriately as an AI company. Or also maybe even government relations.
   - Consumer/user education.
   - Talent/recruiting.
   - Internal communication.

13 References

1. Machine Learning by Andrew NG | Coursera (2011)
   https://www.coursera.org/learn/machine-learning
2. FIT BUT - Classification and Recognition (IKR)
   https://www.fit.vut.cz/study/course/7996/.en
3. FIT BUT - Knowledge Discovery in Databases (ZZN)
   https://www.fit.vut.cz/study/course/7012/.en
4. FEKT VUT - Machine Learning (STU)
   http://midas.uamt.feec.vutbr.cz/STU/stu_cz.php
5. A Few Useful Things to Know about Machine Learning (Pedro Domingos)
   https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf
6. Kohavi, Ron. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. 14. (2001)
   https://www.researchgate.net/publication/2352264_A_Study_of_Cross-Validation_and_Bootstrap_for_Accuracy_Estimation_and_Model_Selection
7. S. B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques (2007)
   https://datajobs.com/data-science-repo/Supervised-Learning-[SB-Kotsiantis].pdf
8. Machine Learning Yearning by Andrew NG (2018)
   http://www.mlyearning.org
9. Deep Learning Specialization by Andrew NG | Coursera (2017)
   https://www.coursera.org/specializations/deep-learning
10. Neural Networks for Machine Learning by Geoffrey Hinton | Coursera (2013)
    https://www.coursera.org/learn/neural-networks/
11. Machine Learning Interview Preparation | Udacity
    https://classroom.udacity.com/courses/ud1001

12. Data Science Interview Preparation | Udacity
https://classroom.udacity.com/courses/ud944

13. AI For Everyone | Coursera (2019)
https://www.coursera.org/learn/ai-for-everyone/home/info

14. AI Transformation Playbook | Andrew Ng (2018)
https://landing.ai/ai-transformation-playbook/

15. Eigenvectors and Eigenvalues | Udacity
https://classroom.udacity.com/courses/ud104

16. Careers in Data Science A-Z™ | Udemy (2018)
https://www.udemy.com/careers-in-data-science-a-ztm/

17. Neural Networks and Deep Learning | Michael Nielsen (2016)
http://neuralnetworksanddeeplearning.com/

18. The Hundred-Page Machine Learning Book | Andriy Burkov (2019)
http://themlbook.com/

19. The Data Engineering Cookbook | Andreas Kretz (version 3, 2019)
https://github.com/andkret/Cookbook