Machine Learning - CPEN 355

Lecture Notes

Souradeep Dutta
(pronounced as Show-Ro-Deep Duh-tah)

Contents

Chapter 1

What is intelligence ?

Reading :

1. "A logical calculus of the ideas immanent in nervous activity " by McCulloch and Pitts (1943)
2. "Computing machinery and intelligence" by Alan Turing in 1950 (Turing, 2009).

What is intelligence? It is hard to define, I don't know a good definition. We certainly know it when we see it. All humans are intelligent. Dogs are plenty intelligent. Most of us would agree that a house fly or an ant is less intelligent than a dog. What are the common features of these species? They all can gather food, search for mates and reproduce, adapt to changing environments and, in general, the ability to survive. Are plants intelligent? Plants have sensors, they can measure light, temperature, pressure etc. They possess reflexes, e.g., sunflowers follow the sun. This is an indication of "reactive/automatic intelligence". The mere existence of a sensory and actuation mechanism is not an indicator of intelligence. Plants cannot perform planned movements, e.g., they cannot travel to new places.
A Tunicate in Fig. 1.1 is an interesting plant however. Tunicates are invertebrates. When they are young they roam around the ocean floor in search of

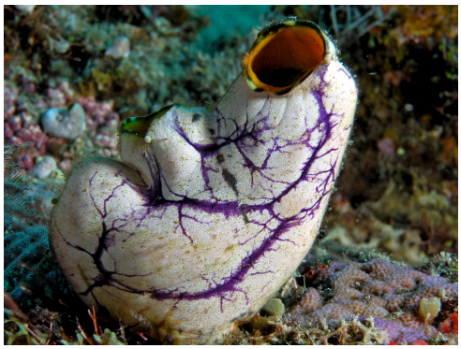


Figure 1.1: A Tunicate on the ocean floor

nutrients, and they also have a nervous system (ganglion cells) at this point of time that helps them do so. Once they find a nutritious rock, they attach themselves to it and then eat and digest their own brain. They do not need it anymore. They are called "tunicates" because they develop a thick covering (shown above) or a "tunic" to protect themselves.

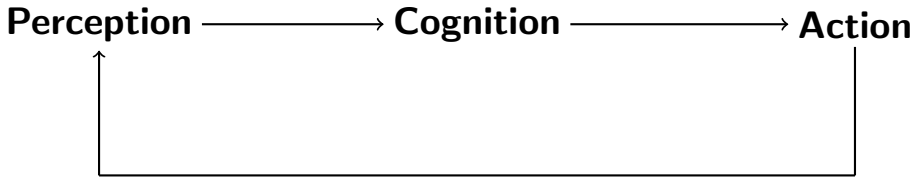
Is a program like AlphaGo intelligent? There is a very nice movie on Netflix on the development of AlphaGo and here's an excerpt from the movie. The commentator in this video is wondering how Lee Se-dol, who was one of the most accomplished Go players in the world then, might defeat this very powerful program; this was I believe after AlphaGo was up 3-0 in the match already. The commentator says so very nonchalantly: if you want to defeat AlphaGo all you have to do is pull the plug. A key indicator of intelligence (and this is just my opinion) is the ability to take actions upon the world. With this comes the ability to affect your environment, preempt antagonistic agents in the environment and take actions that achieve your desired outcomes. You should not think of intelligence (artificial or otherwise) as something that takes a dataset and learns how to make predictions using this dataset. For example, if I dropped my keys at the back of the class, I cannot possibly find them without moving around, using priors of where keys typically hide (which is akin to learning from a dataset) only helps us search more efficiently.

Is an LLM based software tool like ChatGPT/Gemini intelligent ? The short answer is no. ChatGPT does not understand meaning. It produces language by detecting and reproducing statistical patterns in data, not by forming beliefs, intentions, or comprehension about the world. Intelligence requires understanding ChatGPT has none. Philosopher John Searle, (who recently passed away in 2025 !) proposed the Chinese room argument. Consider the following thought experiment : A person who does not understand Chinese sits in a room. He receives Chinese symbols and follows a rulebook to produce correct responses. To outsiders it looks like the person understands Chinese. But

inside, there is no understanding - only *rule-following*. No matter how sophisticated this rule gets, it is still a rule. In my opinion we should not mistake this for intelligence. Just like no matter how good a store mannequin gets at mimicking human gestures, we can always tell the difference between a real human and a robot.

1.1 Key Components of Intelligence

With this definition, we can write down the three key parts that an intelligent, autonomous agent possesses as follows.



Perception refers to the sensory mechanisms to gain information about the environment (eyes, ears, smell, tactile input etc.). Action refers to your hands, legs, or motors/engines in machines that help you move on the basis of this information. Cognition is kind of the glue in between. It is in charge of crunching the information of your sensors, creating a good "representation" of the world around you and then undertaking actions based on this representation. The three facets of intelligence are not sequential and intelligence is not merely a feed-forward process. Your sensory inputs depend on the previous action you took. While searching for something you take actions that are explicitly designed to give you different sensory inputs than what you are getting at the moment.
This class will focus on learning. It is a component, not the entirety, of cognition.

> Learning is in charge of looking at past data and predicting what future data may look like.

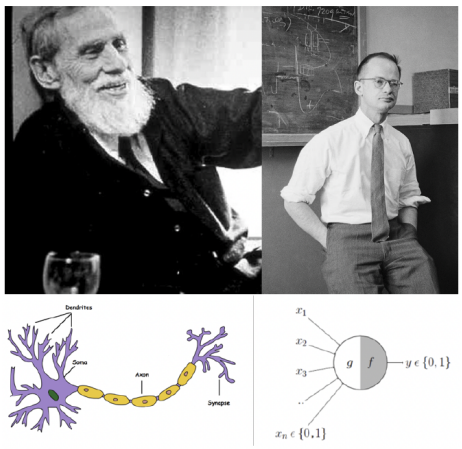
Cognition also involves handling situations when the current data does not match past data, etc. To give you an example, arithmetic problems you solved in elementary school are akin to learning. Whereas figuring out that taking a standard deduction when you file your income tax versus itemized deduction is like cognition. The objective of the learning process is really to crunch past data and learn a prior.
Imagine a supreme agent which is infinitely fast, clever, and can interpret its sensory data and compute the best actions for any task, say driving. Learning from past data is not essential for such an agent; effectively the supreme agent can simulate every physical process around it quickly and 30 decide upon the

best action it should take. Past data helps if you are not as fast as the supreme agent or if you want to save some compute time/energy during decision making.

> A deep network or a machine learning model is not a mechanism that directly undertakes the actions. It is rather a prior on the possible actions to take. Other algorithms that rely on real-time sensory data will be in charge of picking one action out of these predictions. This is very easy to appreciate in robotics: how a car should move depends more upon the real-time data than any amount of past data. This aspect is less often appreciated in non-robotics applications but it holds there as well. Even for something like a recommendation engine that recommends movies in Netflix, the output of a prediction model will typically be modified by a number of algorithms before it is actually recommended to the user, e.g., filters for sensitive information, or toxicity in a chatbot.

1.2 Intelligence : The Beginning (1942-50)

Let us give a short account of how our ideas about intelligence have evolved.

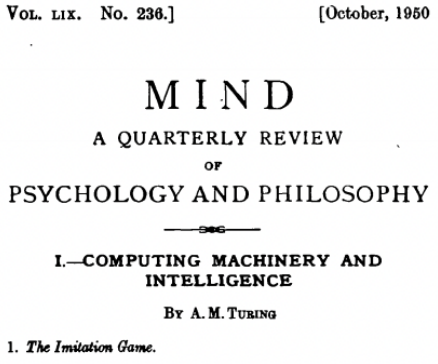


A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

WARREN S. MCCULLOCH AND WALTER PITTS
University of Illinois, College of Medicine,
Department of Psychiatry at the Illinois Neuropsychiatric Institute,
University of Chicago, Chicago, U.S.A.

The story begins roughly in 1942 in Chicago. These are Warren McCulloch who was a neuroscientist and Walter Pitts who studied mathematical logic. They built the first model of a mechanical neuron and propounded the idea that simple elemental computational blocks in your brain work together to per-

form complex functions. Their paper(McCulloch and Pitts, 1943) is an assigned reading for this lecture.



VOL. LIX. No. 236.] [October, 1950

MIND

A QUARTERLY REVIEW

OF

PSYCHOLOGY AND PHILOSOPHY

I.—COMPUTING MACHINERY AND INTELLIGENCE

BY A. M. TURING

1. *The Imitation Game.*

Around the same time in England, Alan Turing was forming his initial ideas on computation and neurons. He had already published his paper on computability by then. This paper(Turing, 2009) is the second assigned reading for this lecture.
McCulloch was inspired by Turing's idea of building a machine that could compute any function in finitely-many steps. In his mind, the neuron in a human brain, which either fires or does not fire depending upon the stimuli of the other neurons connected to it, was a binary object; rules of logic 10 where a natural way to link such neurons, just like the Pitt's hero Bertrand Russell rebuilt modern mathematics using logic. Together, McCulloch & Pitts' and Turing's work already had all the terms of neural networks as we know them today: nonlinearities, networks of a large number of neurons, training the weights in situ etc. Let's now move to Cambridge, Massachusetts. Norbert Wiener, who was a famous professor at MIT, had created a little club of enthusiasts around 1942. They would coin the term "Cybernetics" to study exactly the perception-cognition-action loop we talked about. You can read more in the original book titled "Cybernetics: or control and communication in the animal and the machine" (Wiener, 1965). You can also look at the book "The Cybernetic Brain" (Pickering, 2010) to read more.

1.2.1 Representation Learning

Perceptual agents, from plants to humans, perform measurements of physical processes ("signals") at a level of granularity that is essentially continuous. They also perform actions in the physical space, which is again continuous. Cognitive science on the other hand thinks in terms of discrete entities like concepts, ideas, objects, or categories. These can be manipulated with tools of logic and inference. It is useful to ask what information is transferred from the perception system to the cognition system to create such symbols from signals,

or from cognition to control which creates back signals from the symbols? We will often call these symbols the "internal representation" of an agent.

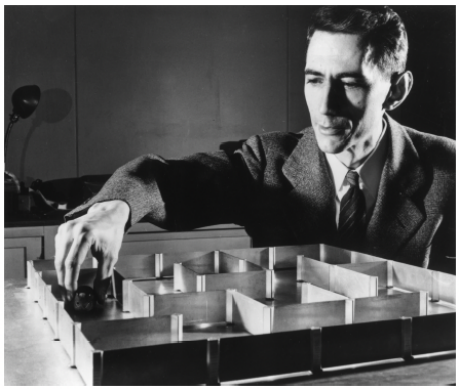


Figure 1.2: Claude Shannon studied information theory. This is a picture of a maze solving mouse that he made around 1950, among the world's first examples of machine learning; read more here

Claude Shannon formulated information theory which is one way to study these kind of ideas. Shannon devised a representation learning scheme for compressing (e.g., taking the intensities at each pixel of the camera and encoding them into something less redundant like JPEG), coding (adding redundancy into the representation to gain resilience to noise before transmitting it across some physical medium such as a wireless channel), decoding (using the redundancy to guess the parts of the data 5 packet that were corrupted during transmission) and finally decompressing the data (getting the original signal back, e.g., pixel intensities from JPEG). Information theory as described above is a tool to transmit data correctly between a sender and a receiver. We will use this theory for a different purpose. Compression, decompression etc. care about never 10 losing information from the data; machine learning necessarily requires you forget some of the data. If the model focuses too much on the grass next to the dogs in the dataset, it will "over-fit" to the data and next time when you see grass, it will end up predicting a dog. It not easy to determine which parts of the data one should forget and which parts one should remember.

The study of artificial intelligence has always had this diverse flavor. Computer scientists trying to understand perception, electrical engineers trying to understand representations and mechanical and control engineers building actuation mechanisms.

1.3 Intelligence : Reloaded(1960-2000)

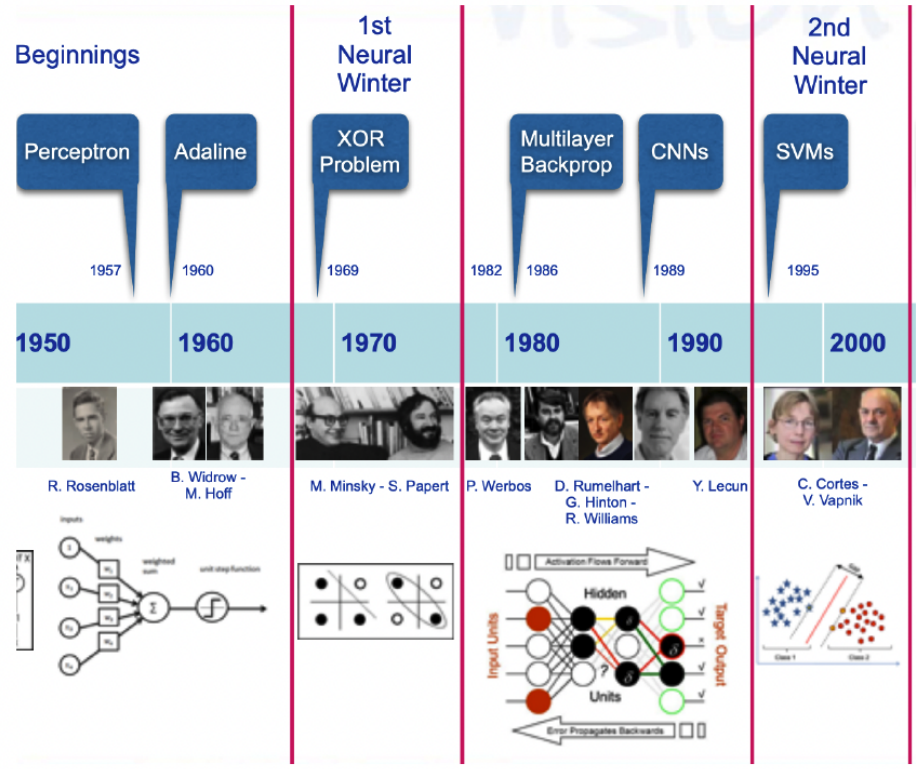
The early period created interest in intelligence and developed some basic ideas. The first major progress of what one would call the second era was made by Frank Rosenblatt in 1957 at Cornell University. Rosenblatt's model called the perceptron is a model with a single binary neuron. It was a machine designed to distinguish punch cards marked on the left from cards marked on the right, and it weighed 5 tons (Link). The input integration is implemented through the addition of the weighted inputs that have fixed weights obtained during the training stage. If the result of this operation is larger than a given threshold, the neuron fires. When the neuron fires its output is set to 1, otherwise it is set to 0. It looks like the function

$$f(x; w) = \text{sign}(w^\top x) = \text{sign}(w_1 x_1 + \cdots + w_d x_d).$$

Rosenblatt's perceptron ((Rosenblatt, 1958)) had a single neuron so it could not handle complicated data. Marvin Minsky and Seymour Papert discussed this in a famous book titled Perceptrons ((Minsky and Papert, 2017)). But unfortunately this book was widely perceived as two very well established researchers being skeptical of artificial intelligence itself. Interest in building neuron-based artificial intelligence (also called the connectionist approach) waned as a result. The rise of symbolic reasoning and the rise of computer science as a field coincided with these events in the early 1970s and caused what one would call the "first AI winter".

There was resurgence of ideas around neural networks, mostly fueled by the (re)-discovery of back-propagation by Rumelhart et al. (1985); Shunichi Amari developed methods to train multi-layer neural networks using gradient descent all the way back in 1967 and this was also written up in a book but it was in Japanese (Amari, 1967). Multi-layer networks came back in vogue because they could now be trained reasonably well. This era also brought along the rise of convolutional neural networks built upon a large body of work starting from two neuroscientists Hubel and Wiesel who did very interesting experiments in the 60s to discover visual cell types (Hubel and Wiesel, 1968) and Fukushima who implemented convolutional and downsampling layers in his famous Neocognitron (Fukushima, 1988). Yann LeCun demonstrated classification of handwritten digits using CNNs in the early 1990s and used it to sort zipcodes (LeCun et al., 1989, 1998). Neural networks in the late 80s and early 90s was arguably, as popular a field as it is today.

Support Vector Machines (SVMs) were invented in Cortes and Vapnik (1995). These were (are) brilliant machine learning models with extremely good performance. They were much easier to train than neural networks. They also had a nice theoretical foundation and, in general were a delight to use as compared to neural networks. It was famously said in the 90s that only the neural network researchers were able to get good performance with neural networks and no one else could train them well. This was largely true even until 2015 or so before the rise of libraries like PyTorch and TensorFlow. So we should



give credit to these libraries for popularizing deep learning in addition to all the researchers in deep learning. Kernel methods, although known much before in the context of the perceptron (Aizerman, 1964; Schölkopf and Smola, 2018), made SVMs very powerful. The rise of Internet commerce in the late 90s meant that a number of these algorithms found widespread and impactful applications. Others such as random forests (Breiman, 2001) further led the progress in machine learning. Neural networks, which worked well when they did but required a lot of tuning and expertise to get to work, lost out to this competition. However, there were other neural network-based models in the natural language processing (NLP) community such as LSTMs (Hochreiter and Schmidhuber, 1997) which were discovered in this period and have remained very popular and performant all through.

1.4 Intelligence: Revolutions(2006-)

The growing quantity of data and computation came together in late 2000s to create ideas like deep Belief Networks (Hinton et al., 2006), deep Boltz-

mann machines (Salakhutdinov and Larochelle, 2010), large-scale training using GPUs (Raina et al., 2009) etc. The watershed moment that got everyone's attention was when Krizhevsky et al. (2012) trained a convolutional neural network to show dramatic improvement in the classification performance on a large dataset called ImageNet. This is a dataset with 1.4 million images collected across 1000 different categories. Performing well on this dataset was considered very difficult, the best approaches in 2011 (ImageNet challenge used to be an annual competition 30 until 2016) achieved about 25% error. Krizhevsky et al. (2012) managed to obtain an error of 15.3%. Many significant results in the world of neural networks have been achieved since 2012. Today, deep networks in their various forms run a large number of applications in computer vision, natural language processing, speech processing, robotics, physical sciences such as physics, chemistry and biology, medical sciences, and many many others (LeCun et al., 2015).

1.5 A summary of our goals in this course

This course will take off from around late 1990s (kernel methods) and develop ideas in deep learning that bring us to today. Our goals are to

1. become good at using modern machine learning tools, i.e., implementing them, training them, modeling specific problems using ideas in ML;
2. understanding why the many quixotic-looking ideas in machine learning works.

After taking this course, we expect to be able to not only develop methods that use machine learning, but more importantly improve existing ideas using foundational understanding of the mathematics behind these ideas and develop new ways of improving machine learning theory and practice.

Chapter 2

Feasibility of Learning

This chapter gives a preview of generalization performance of machine learning models. We will take a more abstract view of learning algorithms here and focus only on binary classification. We will arrive at a "learning model" in Section 2.5.1, i.e., a formal description of what learning means. The topics we will discuss stem from the work of two people: Leslie Valiant who developed the most popular learning model called Probably Approximately Correct Learning (PAC-learning) and Vladimir Vapnik who is a Russian statistician who developed a theory (called the VC-theory) that provided a definitive answer on the class of hypotheses that were learnable under the PAC model.

2.1 Setup : Supervised Learning

- Data pairs $(x, y) \sim \mathcal{P}$ sampled IID from a joint distribution over (features, labels) space $\mathcal{X} \times \mathcal{Y}$.
  - Training dataset $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^n$ consisting of n training data (sampled IID from P).
  - Test data (x, y) sampled from $\mathcal{P}$. At test time, we only observe x. Label y is unknown to us.
- Learning algorithm
  - Hypothesis set $\mathcal{H}$ consists of (many) hypotheses (functions) $h : \mathcal{X} \to \mathcal{Y}$.
  - The learning algorithm $\mathcal{A}$ takes as input the training set D and selects a specific hypothesis from $\mathcal{H}$, which we denote $\hat{h}$.

Notation : h **vs** $\hat{h}$. We reserve h to denote an arbitrary hypothesis in $\mathcal{H}$, while $\hat{h}$ denotes the hypothesis selected by the learning algorithm. That is, $\hat{h}$ depends on (i) the learning algorithm and (ii) the training dataset $\mathcal{D}$. One could write $\hat{h}_{A,D}$, but we lighten notation while urging you to keep this dependence in mind.

2.1.1 Running Example: Linear Classification

Let's ground these abstract concepts with a simple, visual example that we can keep in mind throughout the chapter: binary linear classification in 2D.

- **Data Space:** Our features live in a 2D plane, so $\mathcal{X} = \mathbb{R}^2$. The labels are binary, $\mathcal{Y} = \{-1, +1\}$. Data points $(\mathbf{x}, y)$ are pairs where $\mathbf{x}$ is a point in the plane and y is its class label.
- **Hypothesis Set** $\mathcal{H}$**:** The hypotheses are lines through the origin. Each line is defined (parameterized) by a weight vector $\mathbf{w} \in \mathbb{R}^2$. The classification rule for a given $\mathbf{w}$ is $h_{\mathbf{w}}(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x})$. The set $\mathcal{H}$ is the infinite collection of all such lines.
- **Training Data** $\mathcal{D}$**:** The learning algorithm is given n data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ sampled IID from (some distribution) $\mathcal{P}$. This is our concrete set of blue '+' and red 'o' points on the plane.
- **Learning Algorithm** $\mathcal{A}$ **and Final Hypothesis** $\hat{h}$**:** The algorithm's job is to look at all the training points and pick one specific line, $\hat{h}$, that it thinks is best.
- **The Goal:** As shown in Figure 1, our ultimate goal is to use the chosen line $\hat{h}$ to perform well on **new, unseen test points** $\mathbf{x}$. The next section quantifies "perform well."

2.2 How good is a hypothesis? Memorization vs Generalization

In order to formalize the goal of learning, we first need to formalize how we quantify whether a given hypothesis $h \in \mathcal{H}$ is "good" and "how good" it is. For concreteness, assume for now a **classification** setting such that $\mathcal{Y} = \{1, \ldots, k\} := [k]$, where k is the number of classes (e.g., cats, dogs, planes, etc.).

Definition 2.2.1 (Test Error)**.** *The test error (or risk, or population error, or out-of-sample error) of a hypothesis $h \in \mathcal{H}$ is defined as*

$$R(h) := \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{P}}\big[\mathbf{1}[h(\mathbf{x}) \neq y]\big] = \Pr_{(\mathbf{x},y)\sim\mathcal{P}}\big(h(\mathbf{x}) \neq y\big),$$

where $\mathbf{1}[h(\mathbf{x}) \neq y]$ is the ***zero-one (0/1) loss****.*

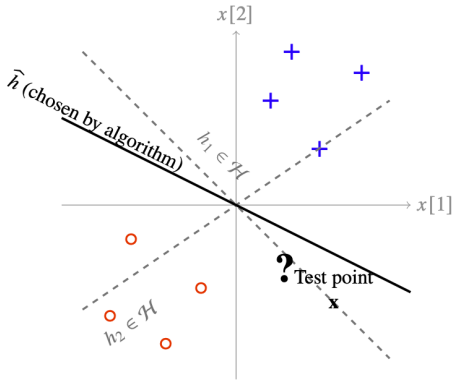


Figure 2.1: Visualizing the learning setup. The algorithm sees the blue '+' and red 'o' training points and selects the solid black line/hypothesis ($\hat{h}$) from the set of all possible lines/hypotheses (a few are shown as dashed). The goal is to perform well (see Sec. 2.2) on a new test point (the '?').

Generalization is the ultimate goal of learning. A hypothesis h that achieves small test error $R(h)$ is said to generalize well.
The probabilistic setup is crucial here. We define error as an average over all possible data points, assuming they are drawn randomly from $\mathcal{P}$. Without this assumption, generalization would be impossible. Recall the problem from our last discussion: if a test point x were chosen adversarially, our training data would provide no guarantee about its corresponding label y. This is analogous to sampling n balls from a bin; the sample tells us nothing about the color of a specific, hand-picked ball that was not part of our random draw. The IID assumption is what allows us to connect performance on seen data to performance on unseen data.
Now that we have quantified a measure of performance, we can formalize the goal of learning as that of selecting a hypothesis h that minimizes the risk $R(h)$ over all hypotheses in our set $\mathcal{H}$. The obvious challenge here is that we cannot actually measure $R(h)$ because it involves an expectation over the entire (and unknown) distribution $\mathcal{P}$. Instead, we only have access to the finite training set $\mathcal{D}$.
It is certainly possible to measure how well a given h performs on the training set. We do this by averaging the $0/1$ loss over the given examples.

Definition 2.2.2 (Training Error)**.** The training error (or empirical risk, or in-sample error) of a hypothesis $h \in \mathcal{H}$ is defined as

$$\hat{R}_n(h) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[h(\mathbf{x}_i) \neq y_i].$$

The $\hat{}$ denotes the quantity depends on the training data and the subscript n makes explicit the size of the train set.

Memorization [1] We say a hypothesis h memorizes the training data, or interpolates the data, if $\hat{R}_n(h) = 0$. That is, the training data gets memorized when the hypothesis makes no mistake when evaluated on all the training examples.

2.3 Generalization Error: From Train to Test

Question: What does the value of the training error tell us about the holy grail of learning, which is the **test error**?
Intuition: Ideally, we would be happy if small training error translates to small test error. This would give a ready recipe for learning: select the hypothesis that minimizes training error! So, is $\hat{R}_n(h)$ close to $R(h)$? This is where the IID assumption is handy. Once h is fixed, the individual-sample errors $R_i := \mathbf{1}[h(\mathbf{x}_i) \neq y_i]$ are IID Bernoulli random variables with mean $R(h)$. By the Law of Large Numbers (LLN), their average converges to the mean:

$$\hat{R}_n(h) \to R(h).$$

With infinite samples, the training error of a fixed hypothesis approaches the true risk! Are we done?
Subtleties: There are two important subtleties to discuss.

- The asymptotic statement from the LLN is encouraging, but it does not quantify the **rate** of convergence. We need to know how good our approximation is for a finite number of samples n.
- The approximation is only valid for a single, fixed hypothesis h. But what we really care about is the performance of $\hat{h}$, the hypothesis we choose after seeing the data.

Rate: How fast the train error of a fixed hypothesis approaches the test error? We can answer this by recalling a statement stronger than the LLN, that is the central limit theorem (CLT)!

[1] Memorization can often be an overloaded term in ML. The technical term for achieving zero training error is *interpolation*.

First, review the central limit theorem quickly. Let $X_1, X_2, X_3, \ldots, X_n$ be independently and identically distributed random variables with :

- finite mean $\mu = \mathbb{E}[X_i]$
- finite non-zero variance $\sigma^2 = Var(X_i)$

Let

$$\hat{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i$$

, be the sample mean. Then,

$$\sqrt{n}(\hat{X}_n - \mu) \xrightarrow{d} \mathcal{N}(0, \sigma^2).$$

Equivalently for large n, $\hat{X}_n \approx \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right)$

According to the CLT, the normalized and centered empirical mean $\sqrt{n}\left(\hat{R}_n(h) - R(h)\right)$ converges in distribution to a Gaussian random variable $\mathcal{N}(0, \sigma^2)$. Equivalently, the gap $\hat{R}_n(h) - R(h)$ is distributed as $\mathcal{N}(0, \sigma^2/n)$. Importantly, note that the variance of the gaussian decreases with n and in the limit of $n \to \infty$ the Gaussian has its entire mass at zero, recovering the LLN. Recall also that the Gaussian distribution has an *exponential tail*. Thus, intuitively the gap $\hat{R}_n(h) - R(h)$ goes to zero exponentially fast in n. This intuition can be formalized by an inequality known as **Hoeffding inequality**:

$$\Pr\left(\, |\hat{R}_n(h) - R(h)| > t \,\right) \leq 2e^{-2t^2 n} \qquad \text{for all } t > 0. \tag{2.1}$$

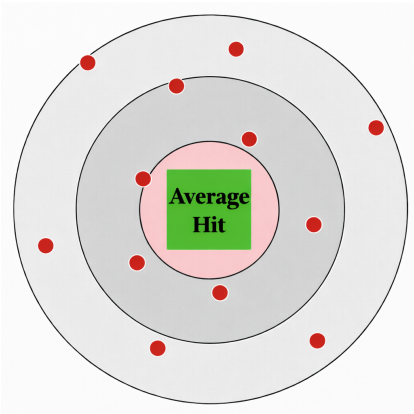
What is the probability over in the above expression? Remember, we took h to be fixed and also $R(h)$ already involves an expectation, so is itself a deterministic quantity (convince yourself!). The only random variable is $\hat{R}_n(h)$ which depends on the training set $\mathcal{D}$. The training set $\mathcal{D}$ consisting of n IID samples $(\mathbf{x}_i, y_i)$ is itself random. So, the probability above is over the randomness of drawing n IID samples $(\mathbf{x}_i, y_i)$ from the distribution $\mathcal{P}$. What the inequality says is that for any threshold t, no matter how small, if we draw a random training set from $\mathcal{P}$ and compute the training error $\hat{R}_n(h)$ then it will be at most t away from the holy grail test-error $R(h)$ with probability at least $1 - 2e^{-2t^2 n}$. To better see how the threshold determines the probability and vice versa, an equivalent way to state Hoeffding's inequality is to first pick some desired probability of success $1 - \delta \in (0, 1)$ (say $\delta = 0.05 \Rightarrow 1 - \delta = 0.95$) and select

$$t = \sqrt{\frac{\log(2/\delta)}{2n}}.$$

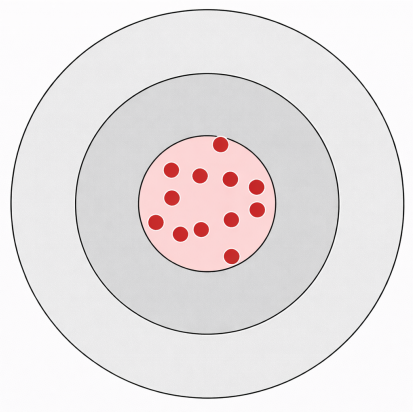
Then,

$$\text{With probability at least } 1 - \delta, \qquad |\hat{R}_n(h) - R(h)| \leq \sqrt{\frac{\ln(2/\delta)}{2n}}.$$

2.4 Remarks on Concentration



(a) **Expectation**: This estimator's "shots" are widely scattered. While any single shot is unreliable, the process is unbiased: the average position of all shots is exactly on the bullseye. The estimator is correct in expectation.



(b) **Concentration**: This archer/estimator is not only unbiased but also precise. Every single shot is tightly clustered around the bullseye, making any single shot a reliable indicator of the true center. In machine learning, concentration guarantees that the single estimate we calculate from our one dataset is very likely to be close to the true value.

Figure 2.2: Expectation vs Concentration

It is important to understand and appreciate how concentration is more powerful that something holding in expectation. The training error $\hat{R}_n(h)$ is an **estimator** of the true, but unknown, test error $R(h)$. For a fixed hypothesis set, this estimator is **unbiased**, that is, it is correct in expectation[2]:

$$\mathbb{E}_{\mathcal{D}}\left[\hat{R}_n(h)\right] = R(h).$$

What this means is that if we could draw many many different datasets and calculate our estimator on each one, the average of all those estimates would be the true value we're trying to find!
Unbiased-ness, however, does *not* promise anything about the single estimate we get from our *one* dataset :(
Concentration inequalities (like Hoeffding's, which we discussed) give us a guarantee about this. They tell us that if our dataset is large enough, the result of our *single* experiment becomes very likely ("concentrated") to be extremely close to the true expected value. This is exactly what we want in ML practice.

[2]Convince yourself this is true. Expectation is over realizations of the training set. *Hint:* Use THE property of expectation, i.e. linearity

We only have one dataset and we compute our estimator (the training error) just once. **Concentration gives us confidence that the single value we calculated is a reliable reflection of the true, underlying value.**
Hoeffding's inequality is just an example of a concentration inequalities. Concentration inequalities more generally can be thought of as finite-sample refinements of the LLN and the CLT. The term 'concentration' captures the essence of these inequalities (as well as the CLT and the LLN): a function (e.g., sum) of many many (large n) random variables that does not depend too much on any small change of any individual variable (e.g., the sum of large n terms doesn't change much by changing only one of the terms a little) concentrates to its expectation. More details are far beyond the scope of the course, but keep in mind that the concentration phenomenon is key in machine learning and more generally in high-dimensional data analysis. In an important sense, concentration is a blessing of dimensionality (contrast to computation which is often regarded as a curse of dimensionality).

What's next?

Does the above inequality hold for the final chosen hypothesis $\hat{h}$? The answer is no. The reason is that $\hat{h} = \hat{h}_{\mathcal{D},\mathcal{A}}$ depends on the training data. This causes the independence assumption to break. Concretely, the set of binary random variables $R_i = \mathbf{1}[h(x_i) \neq y_i]$ are no longer independent if h depends on the set $\{(x_i, y_i)\}_{i\in[n]}$. The final hypothesis $\hat{h}$ certainly does and so we will have to work around this technical challenge. Once we do that, we can revisit the question of how to achieve the goal of learning (minimizing the test error) by only having access to training data.

2.5 Generalization Bound

Goal of Learning. Find a hypothesis

$$h : \mathcal{X} \to \mathcal{Y}$$

that minimizes the *test error*

$$R(h) = \mathbb{E}_{(x,y)\sim P}\left[\mathbf{1}\left[h(x) \neq y\right]\right].$$

That is, solve the following optimization problem:

$$h^* = \arg\min_h R(h).$$

Optimization lingo : An optimization problem consists of an *objective function* (here, $R(\cdot)$) and an *optimization variable* (here, h). Solving an optimization problem amounts to finding the h that minimizes the objective. We call this specific h the *solution* to the optimization and denote it by

$$\arg\min_h R(h),$$

or simply by $h^\star$. We call the value of the objective at that solution, namely $R(h^\star)$, the *optimal cost* of the problem. In our learning setup, $h^\star$ can be thought of as the *golden hypothesis*, and $R(h^\star)$ as the *minimum risk*.

The challenge, of course, is that we cannot evaluate the objective function of this minimization problem, since it involves computing an expectation over the unknown data distribution P. Instead, we are given access to a training set

$$D := \{(x_i, y_i)\}_{i\in[n]}$$

consisting of n examples sampled IID from the underlying (unknown) distribution P.

Holy Grail of Machine Learning Practice: Find a "good" empirical estimate (i.e., one that can be evaluated on training data) of the test error, and minimize that instead.
A natural guess for such an estimate is the *training error* $R_n(h)$, and one may attempt to minimize this quantity instead. That is, solve the following optimization problem:

$$\hat{h} = \arg\min_{h\in\mathcal{H}} R_n(h).$$

Recall that we use the $\widehat{\cdot}$ notation for quantities that depend on the training set, and the solution h of the above minimization certainly depends on the training set, since $R_n(\cdot)$ depends on the training data. Also note that the minimization is constrained to be over a hypothesis set $\mathcal{H}$, which we get to choose. Finally, note that empirical risk minimization (ERM) is an example (in fact, a very popular one) of a learning algorithm.

So suppose (for now) that we pick a hypothesis set $\mathcal{H}$ and solve the above **empirical risk minimization problem (ERM)**. Certainly, $\widehat{R}_n(\hat{h})$ is the smallest among $\widehat{R}_n(h)$ for all $h \in \mathcal{H}$. But is the test error $R(\hat{h})$ evaluated at $\hat{h}$ also small? How small or large is it?
We could answer that question if we had a way to upper bound the so-called **generalization gap**

$$\hat{\Delta}_n := \left|\widehat{R}_n(\hat{h}) - R(\hat{h})\right|.$$

If $\hat{\Delta}_n$ is small, say smaller than some threshold t, then we are guaranteed that the unknown test error $R(\hat{h})$ is at most $\hat{R}_n(\hat{h})$, which we can measure and have

ensured is small, plus t. Concretely, if we can select $\hat{h}$ such that $\widehat{R}_n(\hat{h}) \approx 0$, then

$$R(\hat{h}) \lesssim t,$$

that is, the test error is at most t.
Let us pause for a moment and inspect the nature of the quantity $\hat{\Delta}_n$ that we want to bound. Since $\widehat{R}_n(\hat{h})$ is the empirical risk evaluated on the training set, and the training set consists of n random examples (x_i, y_i), the quantity $\widehat{R}_n(\hat{h})$ is itself random. Thus, $\hat{\Delta}_n$ is also random ! What does it mean, then, to bound a random variable by some quantity t?
What we aim for is a bound that holds with high probability. This probability is over the source of randomness, which—as explained above—comes from randomly drawing the n examples of the training set. Thus, probabilities are taken over the randomness of the training set D.
Concretely, we seek a bound that holds with probability at least $1 - \delta$, for some very small $\delta \in (0, 1)$ (e.g., $\delta = 0.05$, so the bound holds with probability at least 0.95). Formally, our goal becomes to find a threshold t (possibly depending on the number of samples n, the hypothesis set $\mathcal{H}$, and the failure probability δ) such that

$$\Pr\Big(\big|\widehat{R}_n(\hat{h}) - R(\hat{h})\big| \leq t\Big) \geq 1 - \delta,$$

or equivalently (check your understanding of this equivalence!),

$$\Pr\Big(\big|\widehat{R}_n(\hat{h}) - R(\hat{h})\big| > t\Big) < \delta \tag{2.2}$$

This should remind you of Section 2.3 where we bound the gap between empirical and test error for an arbitrary fixed hypothesis $h \in \mathcal{H}$. Using Hoeffding's inequality, we showed that for any fixed hypothesis $h \in \mathcal{H}$, with probability at least $1 - \delta$,

$$\big|\hat{R}_n(h) - R(h)\big| \leq \sqrt{\frac{\ln(2/\delta)}{2n}},$$

or equivalently, with probability at most δ,

$$\big|\hat{R}_n(h) - R(h)\big| > \sqrt{\frac{\ln(2/\delta)}{2n}}. \tag{2.3}$$

How do we arrive at a statement like Equation 2.2 that holds for $\hat{h}$(the data-dependent hypothesis) from the statement in Equation 2.3 that holds for a fixed ("not dependent on the data") hypothesis ?

2.5.1 The PAC-Learning Model

One way to go about it is to observe that the event

$$\big\{|\hat{R}_n(\hat{h}) - R(\hat{h})| > t\big\}$$

is a subset of the event " there exists some hypothesis in $\mathcal{H}$ for which the gap is greater than t." After all, if the specific data-dependent hypothesis $\hat{h}$ has a large gap,it immediately implies that at least one hypothesis in the set has a large gap. Therefore,the probability of the first event must be less than or equal to the probability of the second, more general event

$$\Pr\left(|\hat{R}_n(\hat{h}) - R(\hat{h})| > t\right) \leq \Pr\big(\exists h \in \mathcal{H} : \ |\hat{R}_n(h) - R(h)| > t\big)\,.$$

Now, the probability that there exists $h \in \mathcal{H}$ for which $|\widehat{R}_n(h) - R(h)| > t$ is exactly the probability of the union of events $|\widehat{R}_n(h) - R(h)| > t$ over $h \in \mathcal{H}$. That is,

$$\Pr\left(|\hat{R}_n(\hat{h}) - R(\hat{h})| > t\right) \leq \Pr\big(\bigcup_{h\in\mathcal{H}} |\widehat{R}_n(h) - R(h)| > t\big)$$

To bound this, we use the union bound. For this, assume (for now) that the hypothesis set $\mathcal{H}$ is finite and contains m hypotheses $h_1, h_2, \ldots, h_m$.The probability that at least one hypothesis has a large error gap is bounded by the sum of individual probabilities:

$$\begin{aligned}\Pr\left(\bigcup_{h\in\mathcal{H}} |\widehat{R}_n(\hat{h}) - R(\hat{h})| > t\right) &\leq \sum_{j=1}^{m} \Pr\left(|\widehat{R}_n(h_j) - R(h_j)| > t\right) \\ &\leq \sum_{j=1}^{m} 2e^{-2t^2 n} = 2me^{-2t^2 n}\end{aligned} \tag{2.4}$$

Here, after the union bound on the first inequality, for each individual probability $h_1, h_2, \ldots, h_m \in \mathcal{H}$ we then used Equation 2.1. By setting this probability of failure to a small value δ, we derive our main result.

Key Result. With probability at least $1 - \delta$, for all $h \in \mathcal{H}$ (and therefore for our chosen $\hat{h}$),

$$|\widehat{R}_n(h) - R(h)| \leq \sqrt{\frac{\ln(2m/\delta)}{2n}}.$$

This gives us the famous **generalization bound**,

$$R(\hat{h}) \leq \underbrace{\widehat{R}_n(\hat{h})}_{\text{Training Error}} + \underbrace{\sqrt{\frac{\ln(2m/\delta)}{2n}}}_{\text{Complexity Penalty}} \tag{2.5}$$

That, the estimated hypothesis $\hat{h}$ is approximately (t) correct with some probability (at least $1 - \delta$). Giving it the name the **PAC learning** framework.
We have shown that under the probabilistic setup, the training set D tells us something likely about fresh data: We can indeed trade the task of minimizing the test error $R(\hat{h})$ (which we cannot evaluate) to that of minimizing the training error $\hat{R}_n(\hat{h})$ (which we can evaluate on the training set). Moreover, this

trading results in paying a complexity penalty term that increases logarithmically with the number of hypothesis (m) and decreases proportional to $1/\sqrt{n}$ with the number of examples n.
Note that the way we arrived at bounding the generalization error of $\hat{h}$ (the empirical risk minimizer in (ERM)) is via having obtained a bound in Eq. 2.3 that holds simultaneously for all hypotheses $h \in \mathcal{H}$. Such bounds are called uniform bounds. The remark to be made is that from such a uniform bound, we can immediately get the same generalization bound as in Eq 2.5 for any data-dependent hypothesis $\hat{h}$ irrespective of how this was chosen after seeing the data. Choosing $\hat{h}$ by solving(ERM)is one option (often a good one), but the bound holds more generally for any learning algorithm $\mathcal{A}$. To give an example, $\hat{h}$ could have been the solution of a so-called regularized ERM:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \widehat{R}_n(h) + \Omega(h) \tag{2.6}$$

where now the objective function is augmented by a regularization term Ω. We will revisit regularization later in the course. For now, think of $\Omega(h)$ as a heuristic proxy that aims to capture the complexity of the hypothesis: 2.6 minimizes the sum of the training error and this complexity penalty term motivated by Eq 2.5. The important thing to note is that the bound in 2.5 holds exactly as is for both the (ERM) or the (rERM) solution. This is both a blessing (the bound is very general!) and a curse (the bound is not properly capturing the impact on generalization of the learning algorithm!)

2.6 The Tradeoff of Model Complexity

Looking closer at (2.5), the generalization bound reveals a fundamental tradeoff. To minimize the right-hand side, we must balance two competing goals:
Competing Goal #1: Minimize training error. To achieve a low $\widehat{R}_n(\hat{h})$, we prefer a large, complex hypothesis set (large m) so we have a better chance of finding a function that fits the data well.
Competing Goal #2: Minimize generalization gap. The complexity penalty grows with m. A more complex model requires more data (n) to ensure the gap between training and test error is small. The number of samples required to achieve a certain error is called the sample complexity [3].
This trade off is captured nicely by what is known as the error vs model-complexity curves. On the one hand,training error $\hat{R}_n$ is (likely to be) monotonically decreasing with increasing model complexity. On the other hand, the generalization gap $\sqrt{\frac{\log(2m/\delta)}{2n}}$ is increasing. This results in a U-shaped test-error curve for $R(h)$. This tells us that there is an optimal, not too small but also not too large,complexity level for the hypothesis set.

[3] According to (2.5),to have error at most ϵ hold with probability at least $1-\delta$, we need $n \geq \frac{2}{\epsilon^2}\log(m/\delta)$

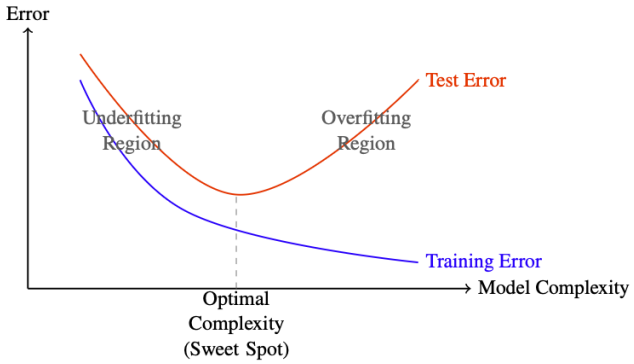


Figure 2.3: A typical U-shaped curve showing the relationship between model complexity and error. As complexity increases, training error decreases, but test error initially falls and then rises.

2.7 Infinite Hypothesis Class and VC Dimension

Let's take a second look at our friend Equation 2.4, it is straightforward to see that if our desiderata includes an error probability of at most δ, then it can be achieved if $2me^{-2t^2n} \leq \delta$. In other words, the least number of training samples required is given by

$$n \geq \frac{1}{2t^2} \log\left(\frac{2m}{d}\right)$$

Thus, if our hypothesis class $\mathcal{H}$ has infinite number of elements in it ($m \to \infty$) , the number of samples n goes to infinity ($n \to \infty$) as well. Note, that Equation 2.5 is still valid since its an upper bound. However, it's a vacuous bound.
Vladimir Vapnik and Alexey Chernovenkis (Vapnik, 2013) developed the so-called VC-theory to answer the above question. Technically, VC-theory transcends PAC-Learning but we will discuss only one aspect of it within the confinements of the PAC framework. VC-theory assigns a "complexity" to each hypothesis $h \in \mathcal{H}$. Before, we can explore VC-dimension we need to understand a basic concept along the way.

Shattering of a set of inputs We say that the set of inputs $D = \{x_1, x_2, \ldots, x_n\}$ is shattered by the hypothesis class $\mathcal{H}$, if we can achieve every possible labeling out of the 2^n labellings using some hypothesis $h \in \mathcal{H}$. The size of the largest set D that can be shattered by $\mathcal{H}$ is called the VC-dimension of the hypothesis class $\mathcal{H}$. It is a measure of the complexity/expressiveness of the class; it counts how many different classifiers the class can express.
If we find a configuration of n inputs such that when we assign any labels to these data, we can still find a hypothesis in $\mathcal{H}$ that can realize this labeling, then $VC(\mathcal{H}) \geq n$. On the other hand, if for every possible configuration of $n+1$ inputs, we can always find a labeling such that no hypothesis in $\mathcal{H}$ can

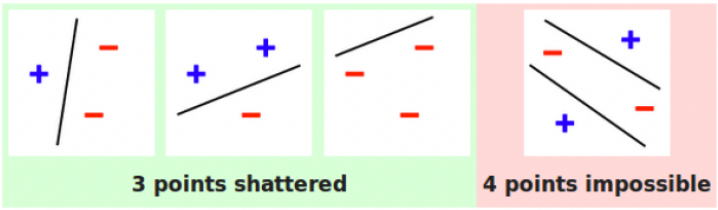


Figure 2.4: d=2: See that for the lower bound, we found some configuration of the 3 points, such that a linear threshold function always separates the points consistently with the labels; for any possible labeling. 3 such labellings are shown, convince yourselves that it can be done for all 8 cases. Observe that we cannot do the same for 4 points. In the figure above one such unrealizable configuration is given (With the "XOR" labeling). To prove the upper bound we need to talk about ANY configuration though. See that the only other case for 4 points, is that one point is inside the convex hull generated from the other 3. Find the labeling that cannot be obtained with linear classifiers in this case.

realize this labeling, then $n \geq VC(\mathcal{H})$. If we find some n for which both of the above statements are true, then $VC(\mathcal{H}) = n$.
Some examples.

- d-dim Linear Threshold Functions: VC-dim $= d + 1$.
- 2 dimensional axis aligned rectangles: VC-dim = 4 (exercise)
- If the hypothesis class is finite, then

$$VC(\mathcal{H}) \leq \log |\mathcal{H}|$$

- For a neural network with p weights and sign activation function $VC = O(plogp)$.

It is a deep result that if the VC-dimension of hypothesis space is finite $V = VC(\mathcal{H}) < \infty$, then this class has the uniform convergence property (for any $h \in \mathcal{H}$, the empirical and population error are close). The number of samples required is lower bounded in the following fashion :

$$n \geq O\left(\frac{V + \log(1/\delta)}{t^2}\right)$$

If a hypothesis class has infinite VC-dimension, then it is not PAC-learnable and it also does not have the uniform convergence property. For a more in depth study and proof of the above theorem result, I would encourage you to read Chapter 6, 7 in Shalev-Shwartz and Ben-David (2014).

Chapter 3

Perceptron and Stochastic Gradient Descent

3.1 Perceptron

We now consider a classification problem. As in the previous setting, we avoid addressing the model selection problem and restrict our attention to linear classifiers. We assume binary class labels $Y \in \{-1, 1\}$. To simplify the notation, we augment each input vector x with an additional constant component equal to 1, allowing the bias term to be incorporated into the weight vector. The linear classifier is then given by

$$\begin{aligned} f(x; w) &= \text{sign}(w^\top x) \\ &= \begin{cases} +1 \text{ if } w^\top x \geq 0, \\ -1 \text{ else} \end{cases} \end{aligned}$$

> The linear classifier remains unchanged if we reorder the pixels of all images consistently in our entire training set and the weights w. The images will look nothing like real images to us. The perceptron does not care about which pixels in the input are close to which others.

We apply the sign function, denoted by $\text{sign}(\cdot)$, to convert the real-valued prediction $w^\top x$ into binary outputs in $\{-1, +1\}$. This formulation corresponds to the classic perceptron model introduced by Frank Rosenblatt (we discussed this history in Section 1.3). As with linear regression, the perceptron can be visualized using the same geometric interpretation.

We now define an objective function for training the perceptron. As usual, our goal is for the model's predictions to agree with the labels in the training data. To this end, we introduce the following loss function:

$$l_{zero-one}(w) := \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{y^i \neq f(x^i;w)} \tag{3.1}$$

The indicator function inside the summation counts the number of classification errors made by the perceptron on the training set. Consequently, the objective seeks a weight vector w that minimizes the average number of mistakes, commonly referred to as the *training error*. A loss function that assigns a penalty of 1 to an incorrect prediction and 0 otherwise is known as the "zero–one loss".

Notation Alert : From this chapter onwards we will denote the i^{th} sample as (x^i, y^i) instead of (x_i, y_i).

3.2 Surrogate Losses

The zero–one loss provides the most direct measure of the perceptron's performance. However, it is non-differentiable, which prevents the use of powerful tools from optimization theory to minimize it and obtain the optimal weight vector w^* . For this reason, machine learning methods commonly employ surrogate loss functions. These losses act as tractable proxies for the true objective—namely, minimizing the number of classification errors. The essential requirement of a surrogate loss is that achieving a small surrogate loss should correspond to making fewer classification mistakes.
The hinge loss is one such surrogate loss. It is given by :

$$l_{hinge}(w) = \max(0, -yw^\top x)$$

If the predicted label $\hat{y} = \text{sign}(w^\top x)$ has the same sign as that of the true label y, then the hinge-loss is zero. If they have opposite signs, the hinge loss increases linearly. The exponential loss

$$l_{exp}(w) = e^{-y\,(w^\top x)}$$

or the logistic loss

$$l_{logistic}(w) = \log\left(1 + e^{-y\,(w^\top x)}\right)$$

Draw the three losses and observe their differences. Also, you may have seen the hinge loss written as $l_{hinge}(w) = \max(0, 1 - y\,w^\top x)$. Why?

3.3 Stochastic Gradient Descent

We now train the perceptron using the hinge loss and a straightforward optimization method. At each iteration, the algorithm updates the weight vector w by moving in the direction of the negative gradient of the loss. We therefore begin by computing the gradient of the hinge loss, which is readily obtained as follows.

$$\frac{dl_{hinge}(w)}{dw} = \begin{cases} -y\,x & \text{for incorrect prediction on } x \\ 0 & \text{else} \end{cases} \tag{3.2}$$

We will use a very naive algorithm, called the perceptron algorithm, to update the weights using this gradient.

The Perceptron Algorithm Perform the following steps for iterations $t = 1, 2, \ldots$

1. At the t^{th} iteration, sample a datum with index $k_t \in \{1, \ldots, n\}$ from D_{train} uniformly randomly, call it (x^{k_t}, y^{k_t}).
2. Update the weights of the perceptron as

$$w^{t+1} = \begin{cases} w^t + y^{k_t} x^{k_t} & \text{if } \operatorname{sign}((w^t)^\top x^{k_t}) \neq y^{k_t} \\ w^t & \text{else} \end{cases} \tag{3.3}$$

Observe that a mistake happens if $(w^t)^\top x^{k_t}$ and y_t^k are of different signs, i.e their product $(w^t)^\top x^{k_t} y_t^k$ is negative. The perceptron's weight vector is updated only when it makes an error on the current datum (x^{k_t}, y^{k_t}). In such cases, the update is designed to improve the perceptron's prediction on that specific sample. This can be seen from the fact that the updated weights corresponding to the most recent sample satisfy the following identity.

$$\underbrace{y^{k_t}\left(w^t + y^{k_t} x^{k_t}\right)^\top x^{k_t}}_{\text{new value}} = y^{k_t}\langle w^t, x^{k_t}\rangle + (y^{k_t})^2\langle x^{k_t}, x^{k_t}\rangle = \underbrace{y^{k_t}\langle w^t, x^{k_t}\rangle}_{\text{previous value}} + \|x^{k_t}\|_2^2$$

In simple words, the value of $y^{k_t}\langle w^t, x^{k_t}\rangle$ increases and becomes more positive. If the perceptron repeatedly misclassifies the same datum, the value will eventually turn positive. However, errors on other training examples may push the perceptron in different directions, potentially causing the updates to continue indefinitely. It is straightforward to show that the algorithm stops updating once all training data are classified correctly. More precisely, if the training set is linearly separable, the perceptron is guaranteed to find a separating linear predictor after a finite number of iterations.

We have in fact just encountered one of the most powerful algorithms in machine learning: stochastic gradient descent (SGD). This method is highly general—whenever the gradient of an objective function can be computed, SGD

can be applied. The procedure used above to train the perceptron was originally introduced by Frank Rosenblatt in 1957 and is commonly referred to as the perceptron algorithm. Viewed from the perspective developed here, the perceptron algorithm is simply an instance of SGD applied to the hinge loss. Variants of SGD had already been studied in the optimization literature well before 1957 (Robbins and Monro, 1951).

3.4 The General Form of SGD

Stochastic gradient descent is a highly general algorithm that can be applied whenever a dataset is available and the objective function is differentiable. The goal of the following section is to introduce foundational notation and concepts related to SGD and optimization, which will be used throughout the subsequent lectures.
Consider an optimization problem

$$w^* = \arg\min_w \frac{1}{n}\sum_{i=1}^{n} l^i(w)$$

where the function l^i denotes the loss on the sample (x^i, y^i) and $w \in \mathbb{R}^p$ denotes the weights. Solving this problem using SGD corresponds to iteratively updating the weights using :

$$w^{t+1} = w^t - \eta \left.\frac{\mathrm{d}l^{k_t}(w)}{\mathrm{d}w}\right|_{w=w^t}$$

The index of the sample in the training set over which we compute the gradient is k_t. This is a random variable

$$k_t \in \{1, \ldots, n\}$$

The gradient of the loss $l^{k_t}(w)$ with respect to w is denoted by :

$$\begin{aligned}\nabla l^{k_t}(w^t) &:= \left.\frac{\mathrm{d}l^{k_t}(w)}{\mathrm{d}w}\right|_{w=w^t} \\ &= \begin{bmatrix} \nabla_{w_1} l^{k_t}(w^t) \\ \nabla_{w_2} l^{k_t}(w^t) \\ \vdots \\ \nabla_{w_p} l^{k_t}(w^t) \end{bmatrix} \\ &\in \mathbb{R}^p\end{aligned}$$

The gradient $\nabla l^{k_t}(w^t)$ is therefore a vector in $\mathbb{R}^p$. We have written

$$\nabla_{w_1} l^{k_t}(w^t) = \left.\frac{\mathrm{d}l^{k_t}(w)}{\mathrm{d}w_1}\right|_{w=w^t}$$

for the scalar-valued derivative of the objective $l^{k_t}(w^t)$ with respect to the first weight $w_1 \in \mathbb{R}$. We can therefore write SGD as

$$w^{t+1} = w^t - \eta \nabla l^{k_t}(w^t) \tag{3.4}$$

The non-negative scalar $\eta \in \mathbb{R}_+$ is called the step-size or the learning rate. It governs the distance traveled along the negative gradient $-\eta \nabla l^{k_t}(w^t)$ at each iteration.

Chapter 4

Kernels, Beginnings of Neural Networks

4.1 Digging Deeper into the Perceptron

4.1.1 Convergence Rate

A natural question is: how many iterations does the perceptron require to fit a given dataset? We assume that the training data are bounded, that is,

$$\|x^i\| \leq R \quad \text{for all } i \in \{1, \ldots, n\},$$

for some constant $R > 0$.
We further assume that the training dataset is linearly separable. That is, there exists a weight vector $w^\star$ such that the perceptron achieves zero training error:

$$y^i \langle w^\star, x^i \rangle > 0 \quad \text{for all } i.$$

We also assume that this classifier separates the data with a positive margin. The distance of an input x^i from the decision boundary $\{x : \langle w^\star, x \rangle = 0\}$ is given by the component of x^i in the direction of $w^\star$ if $y^i = +1$, and in the direction of $-w^\star$ if $y^i = -1$. Equivalently, this distance can be written as

$$\rho_i = \frac{y^i \langle w^\star, x_i \rangle}{\|w^\star\|}.$$

The quantity ρ_i is called the *margin* of sample i. The margin of the dataset is defined as

$$\rho = \min_{i \in \{1,\ldots,n\}} \rho_i.$$

This margin provides a useful measure of the difficulty of a learning problem. We now analyze how the perceptron behaves during training.

Suppose the perceptron makes a mistake at iteration t on the datum (x^{k_t}, y^{k_t}), and performs the update

$$w^t = w^{t-1} + y^{k_t} x^{k_t}.$$

Why ? Check Equation 3.3. You can now prove that after each update of the perceptron the inner product of the current weights with the true solution $\langle w^t, w^* \rangle$ increases at least linearly and that the squared norm $\|w^t\|^2$ increases at most linearly in the number of updates t. The inner product between the current weights and the optimal separator satisfies

$$\begin{aligned} \langle w^t, w^\star \rangle &= \langle w^{t-1}, w^\star \rangle + y^{k_t} \langle x^{k_t}, w^\star \rangle \\ &\geq \langle w^{t-1}, w^\star \rangle + \rho \|w^\star\|. \end{aligned}$$

By induction, after t updates we obtain

$$\langle w^t, w^\star \rangle \geq t\, \rho \|w^\star\|.$$

Next, consider the squared norm of the weight vector:

$$\begin{aligned} \|w^t\|^2 &= \|w^{t-1} + y^{k_t} x^{k_t}\|^2 \\ &= \|w^{t-1}\|^2 + 2 y^{k_t} \langle w^{t-1}, x^{k_t} \rangle + \|x^{k_t}\|^2 \\ &\leq \|w^{t-1}\|^2 + R^2, \end{aligned}$$

where the cross term is non-positive because a mistake was made. By induction,

$$\|w^t\|^2 \leq t R^2.$$

Combining the two bounds, we obtain

$$\cos(w^t, w^\star) = \frac{\langle w^t, w^\star \rangle}{\|w^t\|\, \|w^\star\|} \geq \frac{t\rho}{R\sqrt{t}}.$$

Since $\cos(\cdot, \cdot) \leq 1$, this implies

$$t \leq \frac{R^2}{\rho^2}. \tag{4.1}$$

Therefore, after at most R^2/ρ^2 updates, the perceptron correctly classifies all training data. Notice a few things about this 11 expression.

1. The quantity $\frac{R^2}{\rho^2}$ is dimension independent; that the number of steps reach a given accuracy is independent of the dimension of the data will be a property shared by optimization algorithms in general.

2. There are no constant factors, this is also the worst case number of updates; this is quite rare and we cannot get similar results usually.

3. We can think of the quantity $\frac{R^2}{\rho^2}$ as a measure of the difficulty of the problem. The number of updates scales with the difficulty; if the margin ρ were small, we need lots of updates to drive the training error to zero.

4. This formula also provides some insight into generalization. Let us make two assumptions (a) both train and test data have bounded inputs $\|x\| \leq R$, (b) there exists some weights w^* and margin ρ such that $y\,(w^*)^\top x \geq \rho\|w^*\|\ \forall x, y$. In short, we are assuming that the data can be classified perfectly by some perceptron. Now imagine a different training procedure that does not reuse any data, i.e., the perceptron looks at each sample only once, updates the weights and then throws away that sample after that iteration. The number of mistakes that the perceptron makes before it starts classifying every new datum correctly is also exact $\frac{R^2}{\rho^2}$. In other words, if the perceptron algorithm gets $n \sim \frac{R^2}{\rho^2}$ samples for a problem where the two assumptions hold, then it achieves perfect generalization.

4.1.2 Dual representation

Let us see how the parameters of the perceptron look after training on the entire dataset. At each iteration, the weights are updated in the direction of the sampled datum (x^t, y^t), or they are not updated at all. Therefore, if α^i is the number of times the perceptron sampled the datum (x^i, y^i) during the course of its training and got it wrong, we can write the weights of the perceptron as

$$w^* = \sum_{i=1}^{n} \alpha^i\, y^i\, x^i + w^0. \tag{4.2}$$

where $\alpha^i \in \{0, 1, \dots\}$ and w^0 is the initial weight configuration of the perceptron. Let us assume that $w^0 = 0$ for the following discussion.
The perceptron therefore is using the classifier

$$f(x, w) = \text{sign}(\hat{y}),$$

where

$$\hat{y} = \left(\sum_{i=1}^{n} \alpha^i\, y^i\, x^i\right)^\top x = \sum_{i=1}^{n} \alpha^i\, y^i\, (x^i)^\top x. \tag{3.3}$$

Remember this special form: the inner product of the new input x with all the other inputs x^i in the training dataset is combined linearly to get the prediction. The weights of this linear combination are the dual variables which measure how many tries it took the perceptron to fit that sample during training.

4.2 Creating nonlinear classifiers from linear ones

Linear classifiers such as the perceptron, or the support vector machine (SVM), can be extended to nonlinear ones. The trick is essentially the same that we

use when we fit polynomials (polynomials are nonlinear) using the formula for linear regression.
We are interested in mapping input data x to some different space. This is (usually) a higher-dimensional space called the feature space:

$$x \mapsto \phi(x).$$

The quantity $\phi(x)$ is called a feature vector.

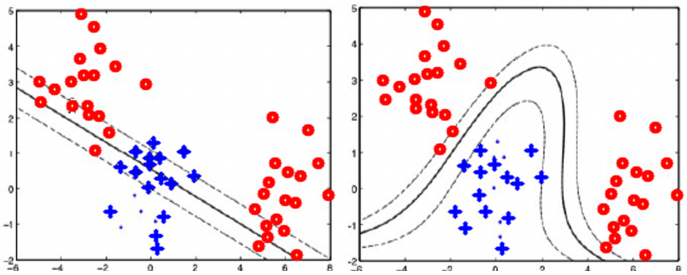


Figure 4.1: Feature vectors can be used to induce non-linear features in the classification space.

For example, in the polynomial regression case for scalar input data $x \in \mathbb{R}$, we used

$$\phi(x) := \left[1,\ \sqrt{2}x,\ x^2\right]^\top$$

to get a quadratic feature space. The role of $\sqrt{2}$ will become clear shortly. Certainly, this trick of creating polynomial features also works for higher-dimensional input:

$$\phi(x) := \left[1, x_1, x_2, \sqrt{2}\,x_1x_2, x_1^2, x_2^2\right]^\top$$

Having fixed a feature vector $\phi(x)$, we can now fit a linear perceptron on the input data $\{(\phi(x^i), y^i)\}$. This involves updating the weights at each iteration as

$$w^{t+1} = \begin{cases} w^t + y_t\,\phi(x^t), & \text{if } \operatorname{sign}\big((w^t)^\top \phi(x^t)\big) \neq y^t, \\ w^t, & \text{otherwise.} \end{cases} \tag{4.3}$$

At the end of such training, the perceptron weights are

$$w^* = \sum_{i=1}^{n} \alpha^i\, y^i\, \phi(x^i),$$

and predictions are made by first mapping the new input to the feature space:

$$f(x; w) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha^i\, y^i\, \phi(x^i)^\top \phi(x)\right) \tag{4.4}$$

Notice that we now have a linear combination of the features $\phi(x^i)$, not the data x^i, in our formula to compute the output.

The concept of a feature space seems like a panacea. If we have complex data, we simply map it to some high-dimensional feature and fit a linear function to these features. However, the "curse of dimensionality" coined by Richard Bellman states that to fit a function in $\mathbb{R}^d$ the number of samples needs to be exponential in d. It therefore stands to reason that we need a lot more data to fit a classifier in feature space than in the original input space. Why would we still be interested in the feature space then?

4.3 Kernels

Observe the expression of the classifier in Eq. 4.4. Each time we make predictions on a new input, we need to compute n inner products of the form

$$\phi(x^i)^\top \phi(x).$$

If the feature dimension is high, we need to enumerate a large number of feature dimensions if we are using the weights of the perceptron, or these inner products if we are using the dual variables. Observe however that even if the feature vector is large, we can compactly evaluate certain inner products.
For example, consider the feature vector

$$\phi(x) = \left[1,\ \sqrt{2}\,x,\ x^2\right],$$

$$\phi(x') = \left[1,\ \sqrt{2}\,x',\ x'^2\right],$$

for scalar input $x \in \mathbb{R}$. Then

$$\phi(x)^\top \phi(x') = 1 + 2xx' + (xx')^2 = (1 + xx')^2.$$

Kernels are a formalization of this idea. A kernel is a function

$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

which is symmetric and positive semi-definite with two arguments such that,

$$k(x, x') = \phi(x)^\top \phi(x')$$

for some feature map $\phi(\cdot)$ and for all $x, x' \in \mathcal{X}$.
A few examples of kernels are

$$k(x, x') = (x^\top x' + c)^2, \qquad k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right).$$

4.3.1 Kernel perceptron

We can now give the kernel version of the perceptron algorithm. The idea is to simply replace any inner product in the algorithm that looks like $\phi(x)^\top \phi(x')$ by the kernel $k(x, x')$.

Kernel perceptron algorithm. Initialize dual variables $\alpha^i = 0$ for all $i \in \{1, \dots, n\}$. Perform the following steps for iterations $t = 1, 2, \dots$:

1. At the t-th iteration, sample a data point with index ω_t from D_{train} uniformly at random, and call it $(x^{\omega_t}, y^{\omega_t})$.
2. If there is a mistake, i.e., if

$$0 \geq y^{\omega_t} \left(\sum_{i=1}^{n} \alpha^i y^i \phi(x^i)^\top \phi(x^{\omega_t}) \right) = y^{\omega_t} \left(\sum_{i=1}^{n} \alpha^i y^i k(x^i, x^{\omega_t}) \right),$$

then update

$$\alpha^{\omega_t} \leftarrow \alpha^{\omega_t} + 1.$$

Notice that we do not ever compute $\phi(x)$ explicitly, so it does not matter what the dimensionality of the feature vector is. It can even be infinite, for example when using the radial basis function kernel. Observe also that we do not maintain the weights w. Instead, we maintain the dual variables $\{\alpha_1, \dots, \alpha_n\}$ while running the algorithm.

Note that the kernel perceptron computes the kernel over all data samples in the training set at each iteration. This can be expensive and wasteful. The Gram matrix $G \in \mathbb{R}^{n \times n}$,

$$G_{ij} = k(x^i, x^j), \tag{3.6}$$

helps address this problem by computing the kernel on all pairs in the training dataset.

We can now write the mistake condition in step 2 as

$$y^{\omega_t} \left(\sum_{i=1}^{n} \alpha^i y^i k(x^i, x^{\omega_t}) \right) = y^{\omega_t} (\alpha \odot Y)^\top G e^{\omega_t},$$

where $e^{\omega_t} = [0, \dots, 0, 1, 0, \dots]^\top$ is the vector with a 1 in the ω_tth position, $\alpha = [\alpha_1, \dots, \alpha_n]^\top$ denotes the vector of dual variables, $Y = [y_1, \dots, y_n]^\top$ is the vector of labels, and $\odot$ denotes the element-wise (Hadamard) product.

This expression involves only a matrix–vector multiplication, which is more convenient than computing the kernel at each iteration.

Gram matrices can become very large. If the number of samples is $n = 10^6$, not an unusual number today, the Gram matrix has 10^{12} elements. The main drawback of kernel methods is that they require a large amount of memory at training time. Nyström methods compute low-rank approximations of the Gram matrix, which makes operations with kernels easier.

4.3.2 Mercer's theorem

This theorem shows that any kernel that satisfies some regularity properties can be rewritten as an inner product in some feature space.

A function $f : \mathcal{X} \to \mathbb{R}$ is said to be square integrable if

$$\int_{x \in \mathcal{X}} |f(x)|^2 \, dx < \infty.$$

We can think of a function $f(x)$ as a long vector with one entry for each $x \in \mathcal{X}$. The integral in Theorem 4.3.1 is analogous to a vector–matrix–vector multiplication of the form $u^\top G u$.

Theorem 4.3.1 (Mercer's Theorem)**.** For any symmetric function

$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

which is square integrable in $\mathcal{X} \times \mathcal{X}$ and satisfies

$$\int_{\mathcal{X} \times \mathcal{X}} k(x, x') \, f(x) \, f(x') \, dx \, dx' \geq 0, \tag{4.5}$$

for all square integrable functions $f \in L^2(\mathcal{X})$, there exist functions $\phi_i : \mathcal{X} \to \mathbb{R}$ and non-negative numbers $\lambda_i \geq 0$ such that

$$k(x, x') = \sum_{i=1}^{\infty} \lambda_i \, \phi_i(x) \, \phi_i(x')$$

for all $x, x' \in \mathcal{X}$. The condition in Eq. 4.5 is called Mercer's condition.

You may also have seen Mercer's condition written as follows: for any finite set of inputs $\{x_1, \ldots, x_n\}$ and any choice of real-valued coefficients $c_1, \ldots, c_n$, a valid kernel should satisfy

$$\sum_{i,j} c_i c_j k(x^i, x^j) \geq 0.$$

There can be an infinite number of coefficients ϕ_i in the summation.

Remark 3.2 (Checking if a function is a valid kernel). Note that Mercer's condition states that the Gram matrix of any dataset is positive semi-definite:

$$u^\top G u \geq 0 \quad \text{for all } u \in \mathbb{R}^n. \tag{4.6}$$

This is easy to show. We have

$$\begin{aligned} u^\top G u &= \sum_{i,j=1}^{n} u_i u_j G_{ij} \\ &= \sum_{i,j} u_i u_j \left(\sum_{k=1}^{\infty} \lambda_k \phi_k(x^i)^\top \phi_k(x^j) \right) \\ &= \sum_{k=1}^{\infty} \lambda_k \left(\sum_i u_i \phi_k(x^i)^\top \right) \left(\sum_j u_j \phi_k(x^j) \right) \\ &= \sum_{k=1}^{\infty} \lambda_k \| \sum_i u_i \phi_k(x^i) \|^2 \\ &\geq 0. \end{aligned}$$

On the second line, we expanded the term $G_{ij} = k(x^i, x^j) = \sum_{k=1}^{\infty} \lambda_k \phi_k(x^i)^\top \phi_k(x^j)$ using Mercer's condition. Therefore, if you have a function that you would like to use as a kernel, checking its validity is easy by showing that the Gram matrix is positive semi-definite.

> Checking your Python function for whether it is a good kernel is great using Eq. 4.6.

Kernels are powerful because they do not require you to think of the feature and parameter spaces. For instance, we may wish to design a machine learning algorithm for spam detection that takes in a variable length of feature vector depending on the particular input. If $x[i]$ is the i^{th} character of a string, a good way to build a feature vector is to consider the set of all length k sub-sequences. The number of components in this feature vector is exponential. However, as you can imagine, given two strings x, x'

```
UBC has a pretty campus
UBC hbs a pxegty cdfvus
```

you can write a Python function to check their similarities with respect to some rules you define, e.g., a small edit distance between the strings. Mercer's theorem is useful here because it says that so long as your function satisfies the properties of a kernel function, there exists some feature space which your Python function implicitly constructs.

4.4 Learning the feature vector

> The central idea behind deep learning is to learn the feature vectors ϕ instead of choosing them *a priori*.

How do we choose what set of feature vectors to learn from? For instance, we could pick all polynomials; we could also pick all possible string kernels.

4.4.1 Random features

Let us go through a thought experiment. Suppose that we have a finite-dimensional feature vector $\phi(x) \in \mathbb{R}^p$. We saw in the perceptron that

$$f(x; w) = \text{sign}\left(\sum_i w_i \phi_i(x)\right),$$

where $\phi(x) = [\phi_1(x), \ldots, \phi_p(x)]^\top$ and $w = [w_1, \ldots, w_p]^\top$ are the feature and weight vectors respectively.
We will set

$$\phi(x) = \sigma\left(S^\top x\right), \tag{4.7}$$

where $S \in \mathbb{R}^{d \times p}$ is a matrix. The function $\sigma(\cdot)$ is a nonlinear function of its argument and acts on all elements of the argument element-wise:

$$\sigma(z) = [\sigma(z_1), \ldots, \sigma(z_p)]^\top.$$

We will abuse notation and denote both the vector version of σ and the element-wise version using the same Greek letter. Notice that this is a special type of feature vector (or a special type of kernel): it is a linear combination of the input elements. What matrix S should we pick to combine these input elements? The paper by Rahimi and Recht (2008) proposed the idea that for shift-invariant kernels (which have the property that $k(x, x') = k(x - x')$[1]), one may use a matrix with random elements as S:

$$S^\top = \begin{bmatrix} \omega_1^\top \\ \vdots \\ \omega_p^\top \end{bmatrix},$$

where $\omega_i \in \mathbb{R}^d$ are random variables drawn from, for example, a Gaussian distribution, and

$$\sigma(z) = \cos(z).$$

[1] An example is RBF (Gaussian Kernel) $k(x, x') = \exp(-\frac{\|x-x'\|^2}{2\sigma^2})$

Using a random matrix is a cheap trick. It lets us create a lot of features quickly without worrying about their quality. Our classifier is now

$$f(x; w) = \text{sign}\left(w^\top \sigma\left(S^\top x\right)\right). \tag{4.8}$$

We can again solve the optimization problem

$$w = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y^i, \hat{y}^i; w), \tag{4.9}$$

with

$$\hat{y}^i = w^\top \sigma\left(S^\top x^i\right),$$

and fit the weights w using stochastic gradient descent as before.

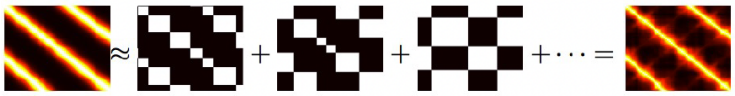


Figure 4.2

As an example, consider the heatmap of a Gabor-like kernel $k(x, x')$ in Fig. 4.2 on the left. Each row and column corresponds to one particular input, x^i or x^j, so regions in the heatmap which are warm are pairs (x^i, x^j) that are similar under the kernel. We can think of the decomposition as follows:

$$\begin{aligned}
&\text{pixel } (i,j) \text{ of the left-most picture} \\
&= k(x^i, x^j) \\
&= \phi(x^i)^\top \phi(x^j) \\
&= \sum_{k=1}^{p} \sigma\left(\omega_k^\top x^i\right) \sigma\left(\omega_k^\top x^j\right) \quad \text{(each black-white matrix)} \\
&= \text{pixel } (i,j) \text{ in the right-most picture.}
\end{aligned}$$

In other words, the p random elements of the matrix S, namely ω_k, come together to give us a useful kernel on the left. A large random matrix S has many such terms on the right hand-side.

> Convince yourself that Figure 4.2 is the right structure. Think in terms of multiplying a graph with a constant factor. Cosine has an interesting effect, in that it only cares or 'clicks' when the arguments are small.

4.4.2 Learning the feature matrix as well

Random features do not work well for all kinds of data. For instance, if you have an image of size 100×100, and you are trying to find a fruit, we can

Figure 4.3

design random features of the form

$$\phi_{ij,kl} = \mathbf{1}_{\text{mostly red color in a box formed by pixels } (i,j) \text{ and } (k,l)}$$

We will need lots and lots of such features before we can design an object detector that works well for this image. In other words, random features do not solve the problem that you need to be clever about picking your feature space or kernel.

Simply speaking, deep learning is about learning the matrix S in Eq. 4.8 in addition to the coefficients w. The classifier now is

$$f(x; w, S) = \text{sign}\left(w^\top \sigma\left(S^\top x\right)\right) \tag{4.10}$$

We now solve the optimization problem

$$w^\star, S^\star = \arg\min_{w,S} \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y^i, \hat{y}^i), \tag{4.11}$$

with

$$\hat{y}^i = w^\top \sigma\left(S^\top x^i\right),$$

as before. Equation 4.10 is our first deep network; it is a two-layer neural network.

Moving from the problem in Eq. 4.9 to this new problem in Eq. 5.5 is a very big change.

1. **Nonlinearity.** The classifier in Eq. 4.10 is not linear anymore. It is a nonlinear function of its parameters w and S, both of which we will call weights.

2. **High dimensionality.** We added a lot more weights to the classifier. The original classifier had $w \in \mathbb{R}^p$ parameters to learn, while the new one also has $S \in \mathbb{R}^{d \times p}$ more weights. The curse of dimensionality suggests that we will need a lot more data to fit the new classifier.

3. **Non-convex optimization.** The optimization problem in Eq. 5.5 is much harder than the one in Eq.4.9. The latter is a convex function, which are easy to minimize. The former is a non-convex function in its parameters w and S because they interact multiplicatively; such functions are harder to minimize.

We could write down the solution of the perceptron using the final values of the dual variables. We cannot do this for a two-layer neural network.

Chapter 5

Deep fully-connected networks and Backpropagation

Reading :

1. Bishop 5.1, 5.3
2. Bishop DL 6.1-6.3.3, Chapter 8
3. Goodfellow 6.3-6.5

5.1 Deep fully-connected networks

A deep neural network takes the idea of a two-layer network to the next step. Instead of having one matrix S in the classifier

$$f(x; v, S) = \text{sign}\big(v^\top \sigma(S^\top x)\big),$$

a deep network has many matrices $S_1, \ldots, S_L$:

$$f(x; v, S_1, \ldots, S_L) = \text{sign}\Big(v^\top \sigma\big(S_L^\top \cdots \sigma(S_2^\top \sigma(S_1^\top x)) \ldots\big)\Big). \tag{5.1}$$

We will call each operation of the form $\sigma(S_k \cdot)$ a *layer*. Consider the second layer: it takes the features generated by the first layer, namely $\sigma(S_1 x)$, multiplies these features using its feature matrix S_2, and applies a nonlinear function $\sigma(\cdot)$ to this result element-wise before passing it on to the third layer.

A deep network creates new features by composing older features.

This composition is very powerful. Not only do we not have to pick a particular feature vector, we can create very complex features by sequentially combining simpler ones. For example, Fig. 5.1 shows the features (more precisely, the kernel) learnt by a deep neural network. The first layer of features are called Gabor-like; they are similar to the ones you constructed in HW 1. These features are combined linearly along with a nonlinear operation to give richer features (spirals, right angles) in the middle panel. The third layer combines the lower features to get even more complex features; these look like patterns (notice a soccer ball in the bottom left), a box on the bottom right, etc.

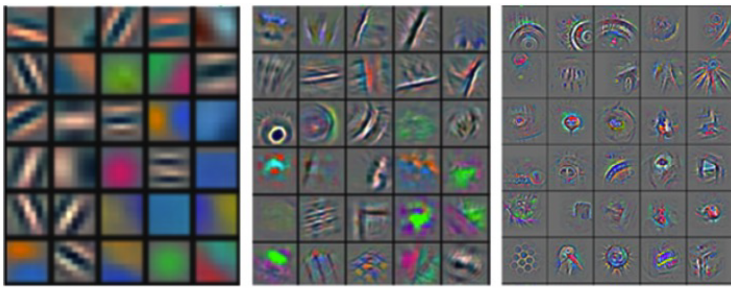


Figure 5.1

The optimization problem for fitting a deep network is written as

$$v^\star, S_1^\star, \ldots, S_L^\star = \arg \min_{v, S_1, \ldots, S_L} \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y^i, \hat{y}^i). \tag{5.2}$$

where the output prediction is :

$$\hat{y} = v^\top \sigma\big(S_L \cdots \sigma(S_2\, \sigma(S_1 x)) \ldots\big)$$

Notice that if fitting a two-layer network was difficult, then fitting a multi-layer neural network like Eq. 5.1 is even harder. There are lots of parameters and consequently we need a lot more data to fit such a model. The optimization problem in Eq. 5.2 is also naturally much harder than its two-layer version. The benefit for going through this difficulty is many fold and quite astounding.

1. **Not having to pick features is very powerful.** Notice that we do not need to worry about what kind of data x is at the input. So long as we can write it into a vector, the classifier as written in Eq. 5.1 works. In other words, the same type of classifier works for image-based data, data from natural language processing, speech processing, and many other types. This is the primary reason why a large number of scientific fields are adopting deep networks.

2. Before the resurgence of deep learning, each of these fields essentially had their own favorite kernels. These kernels were designed across decades

of insights from that specific field (wavelets in signal processing, keypoint detectors and descriptors in computer vision, n-grams in NLP, etc.). It was very difficult for a researcher to use ideas from a different field. With deep learning, this has become much easier. There is still a significant amount of domain insight that you need to make deep networks work well, but the bar for entering a new field is much lower.

3. **Deep neural networks are universal approximators.** In simple words, provided the deep network has enough number of layers and enough number of features in each layer, it can fit any dataset. This is a theorem in approximation theory.

5.1.1 Some deep learning jargon

We have defined the essential parts of a deep network. Let us briefly take a look at some typical jargon you will encounter as you read more.

Activation function. The nonlinear function $\sigma(\cdot)$ in Eq. 5.1 is called the *activation function* (motivated from the threshold-based activation of the McCulloch–Pitts neuron). It is also called a nonlinearity because it is the only nonlinear operation in the classifier. There are many activation functions that have been used over the years.

1. **Threshold**
$$\text{threshold}(x) = \begin{cases} 1, & x \geq 0, \\ 0, & \text{else.} \end{cases}$$
2. **Sigmoid / Logistic**
$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}.$$
3. **Hyperbolic tangent**
$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}.$$
4. **Rectified Linear Units (ReLU)**
$$\text{relu}(x) = |x|_{+} = \max(0, x).$$
5. **Leaky ReLUs**
$$\sigma_c(x) = \begin{cases} x, & x > 0, \\ cx, & \text{else.} \end{cases}$$
6. **Swish**
$$\sigma(x) = x\,\text{sigmoid}(x).$$

Different activation functions work differently. ReLU nonlinearities are the most popular and we will see the reasons why they work better than older ones such as sigmoid or tanh nonlinearities in the backpropagation section.

Logits for multi-class classification. The outputs

$$\hat{y} = v^\top \sigma\big(S_L \cdots \sigma(S_2\, \sigma(S_1 x)) \dots\big)$$

are called the *logits* corresponding to the different classes. This name comes from logistic regression, where logits are the log-probabilities of belonging to one of the two classes.
A deep network affords an easy way to solve a multi-class classification problem: we simply set

$$v \in \mathbb{R}^{p \times C},$$

where C is the total number of classes in the data. Just like logistic regression predicts the logits of two classes, we would like to interpret the vector $\hat{y}$ as the log-probabilities of an input belonging to one of the classes.

Mid-level features. The features at any layer can be studied once you create a deep network. You pass an input x and compute

$$h_l = S_l \cdots \sigma(S_2\, \sigma(S_1 x)) \dots \tag{5.3}$$

to get the pre-activation output of the lth layer. The post-activation output is given by applying the nonlinearity $\sigma(h_l)$.
Sometimes people will call $\sigma(h_L)$ the feature created by a deep network. The rationale here is that just like a kernel-based classifier uses features $\phi(x)$ and fits a linear classifier to these features, we may think of the feature of a deep network to be $\sigma(h_L)$. These features are often very useful. For example, you can use the lower layers of a deep network trained on a different dataset, say classifying cats versus dogs, as the feature generator, but retrain the classifier weights v on your specific problem, say classifying apples versus oranges. Such pre-training is typically used to exploit the fact that someone else has trained a large deep network on a large dataset and thereby learnt a rich feature generator. Training the large model yourself on a large dataset like ImageNet would be quite difficult.

Hidden layers / neurons. The intermediate layers that create the features $h_1, \dots, h_L$ are called the *hidden layers*. A feature is the same as a neuron. Think of the McCulloch–Pitts picture: just like a neuron takes input from all the other neurons connected to it via some weights, a feature is computed using a weighted combination of the features at the lower layer. We will say that a neural network is *wide* if it has lots of features or neurons on each hidden layer. We will say that it is *thin* if it has few features or neurons on each hidden layer.

5.1.2 Weights

It is customary to not differentiate between the parameters of different layers of a deep network and simply say *weights* when we want to refer to all parameters. The set

$$w := \{v, S_1, S_2, \dots, S_L\}$$

is the set of *weights*. This set is typically stored in PyTorch as a set of matrices, one for each layer.

Important. Every time we want to write down mathematical equations, we will imagine w to be a large vector. This is less cumbersome notation. We denote by p the dimensionality of w and imagine that

$$w \in \mathbb{R}^p.$$

The dimensionality p keeps things consistent with linear classifiers where the features were $\phi(x) \in \mathbb{R}^p$. When you use PyTorch to implement an algorithm that requires you to iterate over the weights, say you were implementing SGD from scratch, you will iterate over elements of the set of weights. Using this new notation, we will write down a deep network as simply

$$f(x, w). \tag{5.4}$$

and fitting the deep network to a dataset involves the optimization problem

$$w^\star = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} l(y^i, \hat{y}^i; w). \tag{5.5}$$

We will often denote the loss of the ith sample as simply

$$l^i(w) := l(y^i, \hat{y}^i; w)$$

5.2 The backpropagation algorithm

We would like to use SGD to fit a deep network on a given dataset. As we saw in Section 3.4, if the loss function is denoted by $l^{\omega_t}(w)$ where ω_t was the index of the datum sampled at iteration t, we would like to update the weights using

$$w^{t+1} = w^t - \eta \left. \frac{\mathrm{d}\, l^{\omega_t}(w)}{dw} \right|_{w=w^t}.$$

We have used a scalar $\eta > 0$ as the step-size or the learning rate. It governs the distance traveled along the negative gradient at each iteration. Let us ignore the index of the datum ω_t in this section, imagine $\omega_t = 1$. Implementing SGD therefore boils down to computing the gradient

$$\frac{\mathrm{d}\, l(w)}{\mathrm{d}w}.$$

Backpropagation is an algorithm for computing the gradient of the loss function with respect to weights of a deep network.

5.2.1 One hidden layer with one neuron

Consider the linear regression problem with one layer and one datum, $w, x \in \mathbb{R}^d$ and $v, y \in \mathbb{R}$:

$$l(w, v) = \frac{1}{2}\left(y - v\,\sigma(w^\top x)\right)^2$$

where $\sigma(\cdot)$ is some activation function and our weights are $\{v, w\}$. Let us understand the computational graph of how the loss is computed:

$$w, x \xrightarrow{\text{layer 1}} z \xrightarrow[\sigma]{\text{layer 2}} h \xrightarrow[v]{\text{layer 3}} vh \xrightarrow[y]{\text{layer 4}} l. \tag{5.6}$$

where $h = \sigma(z)$ and $z = w^\top x$. Each node in this graph is either the input/output or an intermediate result of the computation. The gradient of the loss with respect to the weights using the chain rule is

$$\frac{\partial l}{\partial v} = \left(y - v\,\sigma(w^\top x)\right)\left(-\sigma(w^\top x)\right) \tag{5.7}$$

$$\frac{\partial l}{\partial w} = \left(y - v\,\sigma(w^\top x)\right)\left(-v\,\sigma'(w^\top x)\right)x. \tag{5.8}$$

1. **Caching computations for computing the chain rule.** The first idea behind backpropagation is to realize that quantities like $(y - v\sigma(w^\top x))$ or $z = w^\top x$ are computed multiple times in the chain rule in Eqs. 5.7 and 5.8. If we can cache these quantities we can compute the chain rule-based gradient for the different parameters quickly.

2. **Cache is the output of each layer.** The second idea behind backpropagation is to realize that quantities like $(y - vh)$, $h = \sigma(z)$ and $z = w^\top x$ are outputs of the third, second and first layers respectively. In other words, the quantities we need to cache in the chain rule computation are simply the outputs of the individual layers.

3. **Derivatives of the loss with respect to the input of a layer only depends on what happens in that layer and the derivative of the loss with respect to the output of that layer.** The third observation is to see that the quantity $\sigma'(z)$ in Eq. 5.8 is the derivative of the output of the activation function, namely $h = \sigma(z)$ with respect to z, its input argument:

   $$\sigma'(z) = \frac{dh}{dz}.$$

   This derivative is combined with the forward computation $(y - vh)$ to get the gradient with respect to the weights w.

> Backpropagation is simply a book-keeping exercise that caches the forward computation of the graph in Eq. 5.6 and uses these cached values to compute the derivative of the loss l with respect to the parameters of each layer sequentially.

We will use a clever notation to denote the backprop gradient which will make this process very easy. Denote by

$$\overline{v} = \frac{dl}{dv} \tag{5.9}$$

the derivative of the loss l with respect to a parameter v. For our simple two layer (one neuron) neural network, we are interested in computing the quantities - $\overline{w}$ and $\overline{v}$.
Let us also denote the output of the second linear layer (layer 3) as

$$e = vh.$$

Now observe the following "forward computation"

$$z = w^\top x \tag{5.10}$$
$$h = \sigma(z) \tag{5.11}$$
$$e = vh \tag{5.12}$$
$$l = \frac{1}{2}(y-e)^2 \tag{5.13}$$

Let us imagine that we have cached all the quantities on the left hand side of the equalities above. We use these quantities to perform the "backward" computation as follows:

$$\frac{dl}{dl} = \overline{l} = 1,$$

$$\mathbb{R} \ni \overline{e} = \frac{dl}{de} = -(y-e) = \overline{l}\,(-(y-e)) \qquad \text{(from Eq. 5.13)},$$

$$\mathbb{R} \ni \overline{v} = \overline{e}\,\frac{de}{dv} = \overline{e}\,h \qquad \text{(from Eq. 5.12)},$$

$$\mathbb{R} \ni \overline{h} = \overline{e}\,\frac{de}{dh} = \overline{e}\,v \qquad \text{(from Eq. 5.12)},$$

$$\mathbb{R} \ni \overline{z} = \overline{h}\,\frac{dh}{dz} = \overline{h}\,\sigma'(z) \qquad \text{(from Eq. 5.11)},$$

$$\mathbb{R}^d \ni \overline{w} = \overline{z}\,\frac{dz}{dw} = \overline{z}\,x \qquad \text{(from Eq. 5.10)},$$

$$\mathbb{R}^d \ni \overline{x} = \overline{z}\,\frac{dz}{dx} = \overline{z}\,w \qquad \text{(from Eq. 5.10)}.$$

Remark 4.1. An interesting mnemonic to remember backprop is to see that if the forward graph is

$$z = w_1 x_1 + w_2 x_2$$

the backprop gradient is $\overline{w}_1 = \overline{z}\, x_1$ and $\overline{w}_2 = \overline{z}\, x_2$. If x_1 was large and dominated the computation of z during the forward propagation, then $\overline{w}_1$ which is the multiplier of x_1 also gets a dominant share of the backprop gradient $\overline{z}$. The backprop gradient is shared equitably among the different quantities that took part in the forward computation. This is useful to remember when you build neural networks with complex architectures on your own: if there is a part of the network whose activations are very small and it is being combined with another part of the network whose activations have a large magnitude, then the former is not going to going to get a large enough backprop gradient.

Remark 4.2 (Gradient with respect to the input x). Notice that we obtain the gradient of the loss with respect to the input x,

$$\frac{dl}{dx},$$

as a by-product of backpropagation. Backpropagation computes the gradient of the input activations to each layer because this is precisely the gradient that is propagated downwards. So the gradient $\overline{x}$ should not be surprising, after all x is nothing but the input activation to the first layer. This gradient is useful, you can use to find what are called adversarial examples, i.e., input images which look like natural images to us humans but contain imperceptible noise that gives a large value of $\overline{x}$.

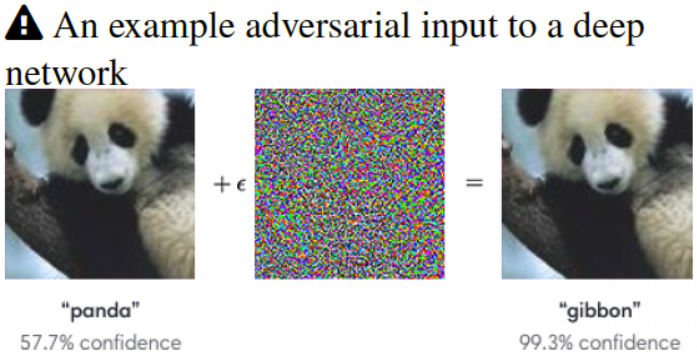


Figure 5.2

5.2.2 Implementation of backpropagation

Consider our neural network classifier given by

$$f(x; v, S_1, \ldots, S_L) = \text{sign}\left(v^\top \omega\big(S_L \cdots \omega(S_2\, \omega(S_1 x)) \ldots\big)\right).$$

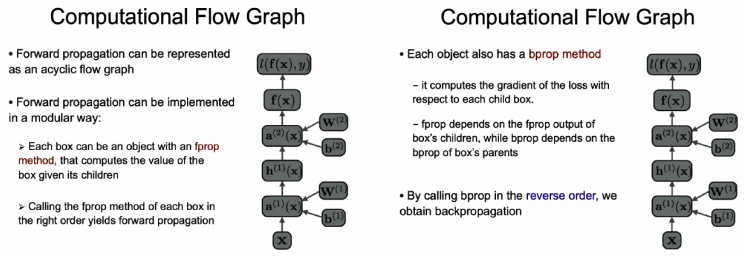


Figure 5.3: Schematic of forward and backward computations in backpropagation

Figure 5.3 shows a schematic of the forward and backward computations in backpropagation. When you build such a multi-layer network in PyTorch, the kth layer is automatically equipped with two member functions.

```
def forward(self, h^{k-1}, S_k):
    # computes the output of the k^th layer
    # given output of previous layer h^{k-1} and
    # parameters of current layer S_k
    return h^k

def backward(self, h^k, d loss/dh^{k}, S_k):
    # computes two quantities
    # 1. d loss/d{S_k}
    # 2. d loss/d{h^{k-1}}
    return d loss/d{S_k}, d loss/d{h^{k-1}}
```

Such forward and backward functions exist for every layer, including the non-linearities. If you implement a new type of layer in a neural network, say a new nonlinearity, you only need to write the forward function. The autograd module inside PyTorch automatically writes the backward function by looking at the forward function. This is why PyTorch is so powerful, you can build complex functions inside your deep networks without having to compute the derivatives yourself.

Chapter 6

Convolutional Neural Networks

Reading

1. Goodfellow 9
2. Bishop DL Chapter 10
3. "Striving for simplicity: The all convolutional net", by Springenberg et al. (2014)

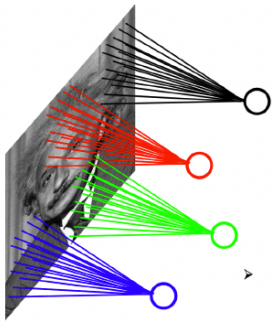


Figure 6.1

So it turns out that we have been talking about what are called "fully-connected" neural networks in the past chapter. There are a few problems that are apparent even in our limited experience.
Fully-connected layers have a lot of parameters. If an input image is of size $100 \times 100 = 10^4$ grayscale pixels and we would like to classify it as belonging to one out of 1000 classes, we need 10M parameters. It is difficult to perform so

many add-multiply operations quickly even on sophisticated GPUs. Further, the curse of dimensionality never goes away; we need lots of data to fit these many parameters.

> Let us consider an example using local connections instead of a fully-connected layer. If each output neuron is connected to only 25 pixels of the 100×100 image and there are 1000 output neurons as show in Figure 6.1, how many weights will this layer have?

Natural data is full of "nuisances" that are not useful for tasks such as classification. E.g., illumination, viewpoint, and occlusions in Figure 6.2 ?

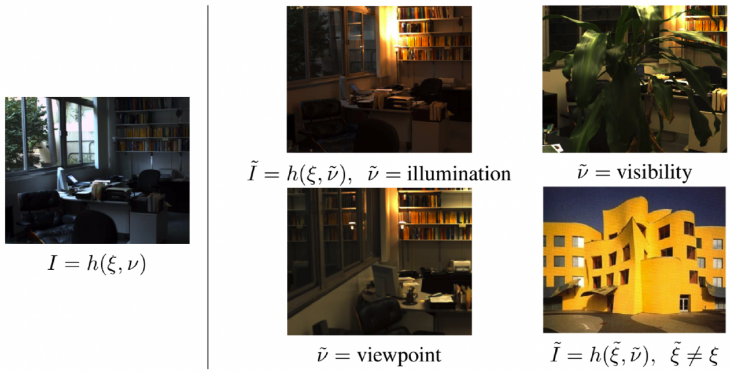


Figure 6.2

Or even semantic ones shown below, Figure 6.3.

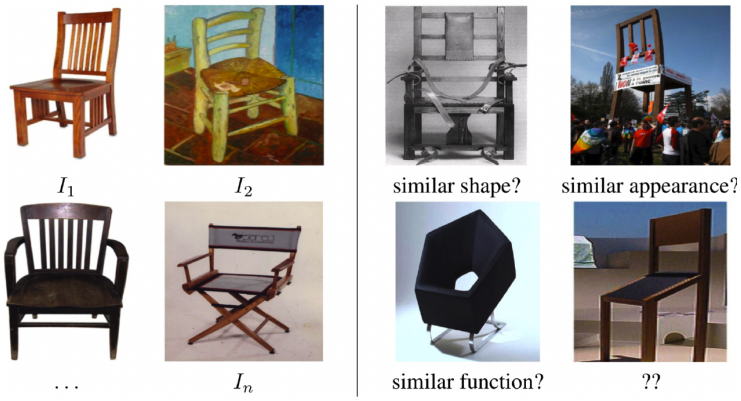


Figure 6.3

Nuisances can be defined as operations that act on the data before you get to see it (nature creates these nuisances). Some of them are special and they have

a group structure, i.e., they satisfy certain algebraic conditions. For instance, images of the same chair taken from different vantage points are projections of different rigid body transformations of the camera. Some other nuisances such as occlusions do not have a group structure, e.g., there is no rigid body transformation that allows us to backcalculate the pixels belonging to a person standing behind a car. Convolutional layers are a simple way to tackle one particular kind of nuisance, that of translations.

6.1 Basics of the convolution operation

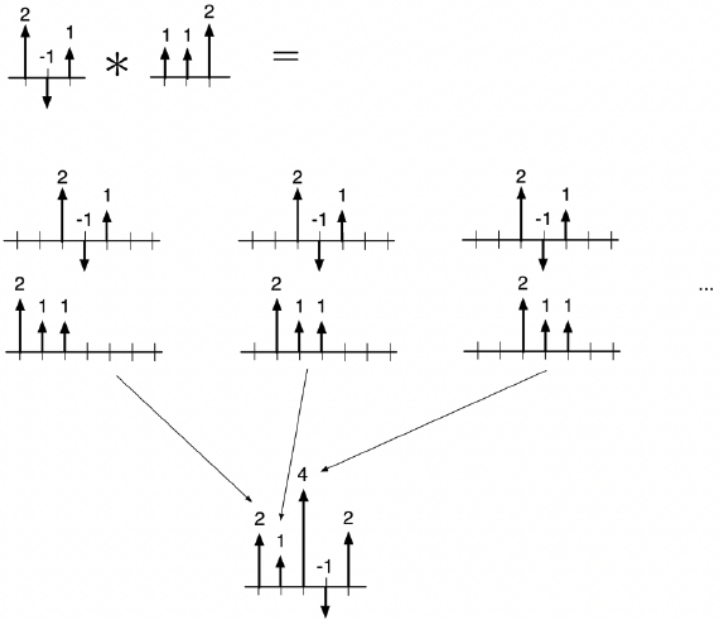


Figure 6.5: Flip and filter style computation of a convolution corresponding to the summation in Equation 6.1

So far, we have seen that the basic unit of a neural network is

$$\sigma(w^\top x).$$

The basic unit of a convolutional neural network is

$$\sigma(x \star w)$$

where the $\star$ denotes a convolution operation. Consider two one-dimensional vectors $x \in \mathbb{R}^3$ and $w \in \mathbb{R}^3$ as shown in Figure 6.5; we will imagine these to be

arrays of infinite length with all the entries at indices $[4, \infty)$ set to zero; this is known as zero-padding the input

$$x = [2, -1, 1, 0, 0, \ldots]$$

$$w = [1, 1, 2, 0, 0, \ldots].$$

> In the signal processing literature, the words filter and kernels are used equivalently, so convolutional filters are also often called convolutional kernels.

The convolution of x with w (which is called the filter) is denoted by

$$(x \star w)_k = \sum_{\tau=-\infty}^{\infty} x_\tau \, w_{k-\tau} \tag{6.1}$$

The element $(x \star w)_k$ at the kth index is a composition of all the terms in the summation on the right hand side. The term $w_{k-\tau}$ for negative arguments is interpreted as a mirror flip of the vector w. For continuous functions, you will have seen the expression

$$(x \star w)(t) = \int_{-\infty}^{\infty} x(\tau) \, w(t - \tau) \, d\tau$$

for the convolution operation.

> Discuss the convolution of a square wave x with a saw-tooth wave w.

For our vectors x, w with three entries the convolution operation looks as follows.

Remark 5.1 (Some identities regarding convolutions). Notice that we can change the variable of integration and set $s = t - \tau$ to get

$$(x \star w)(t) = \int_{-\infty}^{\infty} x(\tau) \, w(t - \tau) \, d\tau = \int_{-\infty}^{\infty} x(t - s) \, w(s) \, ds \tag{6.2}$$

$$= \int_{-\infty}^{\infty} w(s) \, x(t - s) \, ds = (w \star x)(t) \tag{6.3}$$

Convolutions are therefore commutative; you can show similarly that they are also distributive $(f \star g) \star h = f \star (g \star h)$. Convolution is a linear operator, you can show that

$$(f + g) \star h = (f \star h) + (g \star h)$$

for any integrable functions f, g, h.

Remark 5.2 (Padding for implementing convolutions). In order to implement the summation in convolution, we need to pad the input vector x by zeros. How many zeros should we pad it by? You will notice that if the kernel w has $2k+1$ elements, the input vector x need not be padded all the way to infinity, we only need to pad it with $2k$ extra elements on each side.

Most deep learning libraries implement a slightly different operation instead of convolution, even though they call it a convolution. They implement the cross-correlation operation

$$(x \star w)_k = \sum_{\tau=-\infty}^{\infty} x_\tau \, w_{k+\tau}.$$

In simple words, the kernel w is not mirror flipped about the Y axis before computing the summation in Eq. 6.1. While such an operation is not strictly a convolution (you can see the difference if you consider an asymmetric kernel w; cross-correlation and convolution are the same for symmetric kernels), the difference does not matter for deep learning because the kernel w is learned during training. You can mirror flip the kernel after training and interpret the network as indeed performing a convolution with the flipped kernel.

6.1.1 Convolutions of 2D images

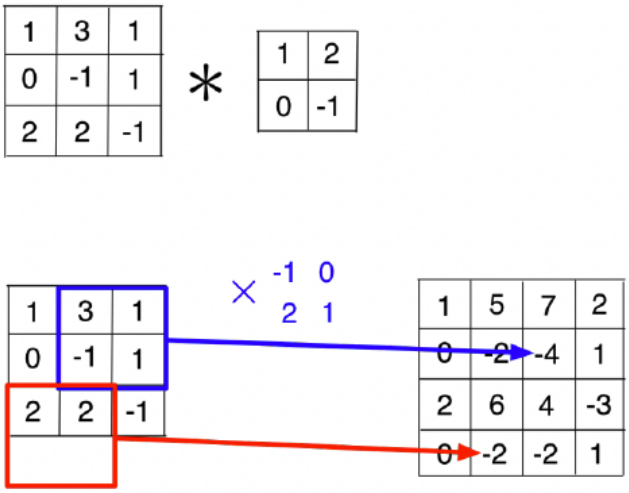


Figure 6.6: Flip and filter style computation of a convolution for a 2D input image corresponding to the summation in Eq. 6.4

Convolutions work in the same way for two-dimensional(Figure 6.6) or three-dimensional input signals. The kernel w will be a matrix of size $k \times k$ in the former case and of size $k \times k \times k$ in the latter.

$$(x \star w)_{i,j} = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} x_{s,t}\, w_{i-s,j-t} \tag{6.4}$$

6.1.2 Some examples

1. Since convolution is a linear operator we should be able to write it as a matrix-vector multiplication. We take the kernel, flip it and sweep it left to right to get the rows of the matrix.

$$(2,-1,1) \star (1,1,2) = \begin{bmatrix} 1 & & \\ 1 & 1 & \\ 2 & 1 & 1 \\ & 2 & 1 \\ & & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}.$$

   Such a matrix is called a Toeplitz matrix. Two-dimensional convolutions can be written as a matrix-matrix multiplication using a similar construction; see.

2. Lots of non-trivial transformations of the image are possible using slight changes in the weights. E.g., blurring

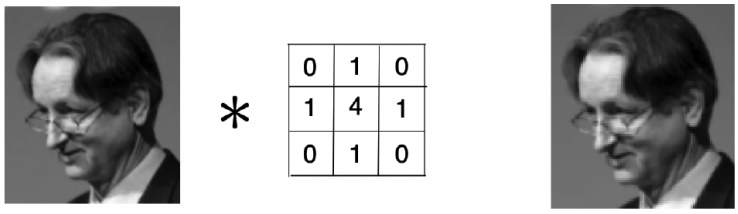


Figure 6.7

   or sharpening,

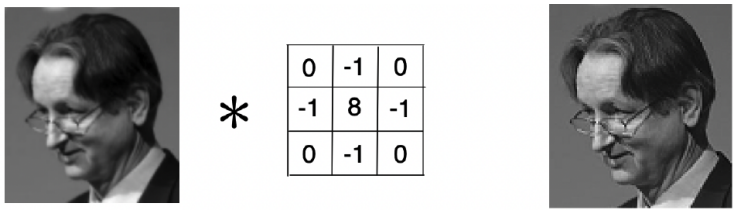


Figure 6.8

We can also detect edges

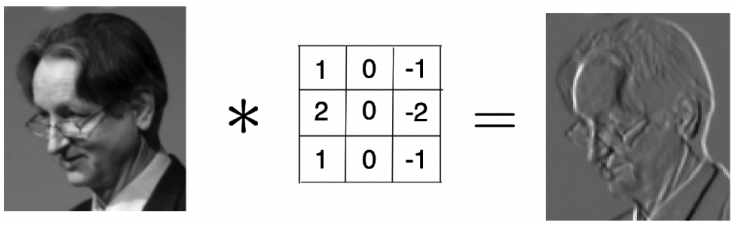


Figure 6.9

This filter is called the Sobel filter and is an integral part of image preprocessing pipelines in computer vision.

3. Just like fully-connected layers, we can also stack up convolutions. The effective receptive field, i.e., the pixels that are considered by the kernel in the convolutional operation increases as we go up the layers.

4. The operation $S \to x$ has $S \in \mathbb{R}^{d \times p}$ weights and returns a vector in R^p. A convolution operator returns a vector $(x \star w) \in R^d$ using K parameters in the kernel w. It is important to note that a lot of parameter sharing is happening while computing the values of the output neurons. You can find some animations at here and here.

5. Padding the input by zeros is common in signal processing because the signals are usually a function of time. We can do a bit better for images than zero padding (RGB = (0, 0, 0)) which is akin to creating an artifact of a dark black border around the image. Reflection padding is a technique (torch.nn.ReflectionPad2d in PyTorch) that mirrors the pixels at the boundary and does not create such artifacts.

Remark 5.3 (Dilated convolutions). You don't need to use a kernel that looks like a contiguous array. We can create holes in the kernel and expand the receptive field. Dilated convolutions do precisely this.

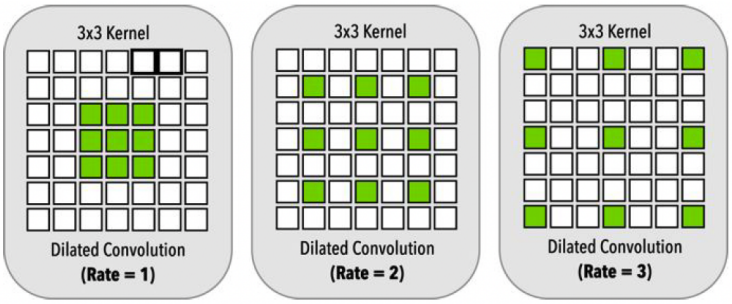


Figure 6.10

These operators are very useful for image segmentation because they capture correlations across large parts of the input image while still enabling the parameter sharing of a convolutional layer.

Remark 5.4 (Separable convolutions). There are 9 weights in a 3×3 kernel. Even convolutional layers can get really big, e.g., a standard CNN used for ImageNet has about 25M weights and is almost entirely convolutional. Thus we might want to reduce the number of weights even further. Separable convolutions are a trick to doing so. Consider a 3×3 kernel and split it into two kernels of 3×1 and 1×3

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}.$$

Using the original kernel requires 9 multiply operations to compute each pixel value. Using the split kernels requires only 6, it also has fewer weights. These are called separable convolutions. The Sobel filter which we saw before can be written as a separable convolution

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

because it measures the gradient of the image intensity independently in the two directions; an edge in an image is a region such that it is either an edge in the horizontal direction or an edge in the vertical direction.
Separable convolutions are very useful when you use high-dimensional data in deep learning, e.g., medical images out of MRI are 4-dimensional images (width, height, depth, channel).

Can we write every 2D convolutional filter as a separable convolution? The answer is no: you will notice that a separable kernel is a rank-1 matrix. The singular value decomposition (SVD) of a separable kernel A is therefore

$$A = \sigma u v^\top$$

for two vectors u, v and singular value σ. Can we however approximate any convolutional kernel as a sum of separable convolutions? The answer to this is yes: observe using the SVD of the kernel $A \in \mathbb{R}^{p \times p}$ that it can be written as

$$A = \sum_{i=1}^{p} \sigma_i u_i v_i^\top.$$

where u_i, v_i are the singular vectors and σ_i are singular values. You don't have to pick all the factors, if you pick a few terms in this summation, you get a good spectral approximation of the matrix A. You will see in Section 5.3 how the convolutional layer in a deep network is structured and may allow the network to learn a complicated kernel A even if the operations are only separable $u_i v_i^\top$.

6.2 How are convolutions implemented?

Convolutions are the most heavily used operator in a deep network. We therefore need to implement them as efficiently as we can. There are a few different ways of implementing convolutions.

1. Write a simple for loop. This works well if the kernel is small in size and this is indeed how PyTorch implements convolutions for kernels of size 3×3 (the operation is coded up in C, not Python of course).

2. We can expand out the kernel as a matrix and in this way a convolutional layer is simply a matrix-vector multiplication. This method is most commonly implemented and works well for sizes up to 5×5.

3. We can use the Fast Fourier Transform (FFT) to compute the convolution as

$$x \star w = \mathcal{F}^{-1}\big[\mathcal{F}[x]\,\mathcal{F}[w]\big].$$

This is efficient for large kernels, say greater than 7×7. Typically, deep learning libraries will choose an algorithm for convolution in run-time after looking at your neural architecture; you do not have to worry about the specific algorithm. A library called cuDNN from Nvidia implements a bunch of convolution algorithms on GPUs efficiently. PyTorch will pick one of these algorithms by checking how long it takes for the first forward-pass on your deep network.

You can set `torch.cudnn.benchmark = False` to prevent PyTorch from searching for the best algorithm to compute convolutions for your architectures every time it launches. While such automated search speeds up training by a small fraction, it may not be desirable in case when you want to debug your code, or evaluate the run time of your algorithm.

But the fact remains that large kernels which allow a larger receptive field (long-range correlations in the input image) are more expensive to compute than smaller kernels. Architectures such as Inception that we will see soon are an attempt to get a large receptive field while still keeping computations in the convolutional layer small.

Remark 5.5 (Stride in convolutional layers). If you see the documentation for the convolutional layer in PyTorch you will also see a parameter known as stride. Stride simply means that the output

$$(x \star w)_k = \sum_{\tau=-\infty}^{\infty} x_\tau w_{k-\tau}$$

is not computed at all values of k; if the stride is set to 2, the output is computed only at every alternate value of k. Note that the default stride as seen in the definition of convolution is 1. Since images change very little from pixel to pixel, this is a neat trick to reduce the redundancy of computing the convolution again and again over similar input. The important artifact of using a stride larger than 1 is that the output $(x \star w)$ is no longer the same length (even after padding) as the input, is half the length if the stride is 2.

6.3 Convolutions for multi-channel images in a deep network

We will now study how the convolutional layer is implemented in a typical deep network. Let us denote the 2D convolution operation on a single-channel 2D image $A \in \mathbb{R}^{w \times h}$ by a kernel $w \in \mathbb{R}^{k \times k}$ by

$$A \star w = B \in \mathbb{R}^{w \times h}.$$

Imagine that we have an RGB input image of size $w \times h$; the RGB indicates that there are three input channels, one for each color. The input to a convolutional layer in a deep network is therefore an array of size $3 \times w \times h$.
Typical deep learning libraries, when they implement a convolutional layer with a kernel w of size $k \times k$, will output an image of size $c \times w \times h$ where c are the number of channels in the image at the output of the layer. Effectively, a convolutional layer maps

$$\mathbb{R}^{3 \times w \times h} \ni A \longmapsto B \in \mathbb{R}^{c \times w \times h}.$$

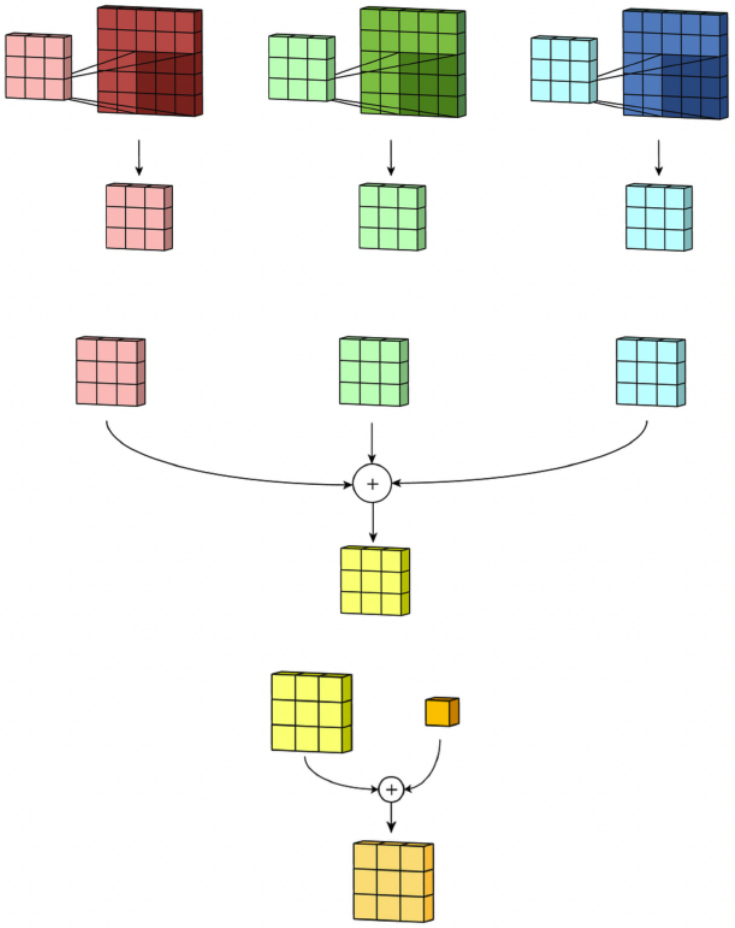


Figure 6.11: Convolutional layer in a typical deep network

The layer performs the operation

$$v_j + \sum_{i=1}^{3} A_i \star w_{ij} = B_j$$

where A_i for $i \in \{1, 2, 3\}$ denotes the ith channel of the input image and B_j for $j \in \{1, \ldots, c\}$ denotes the jth channel of the output image, and the kernel $w_{ij} \in \mathbb{R}^{k \times k}$ is the convolutional kernel. The scalar $v_j \in \mathbb{R}$ denotes the bias. Effectively, there are $3c$ different kernels in one layer and the convolutional layer sums up the result of convolutions on all the input channels and adds a bias to create each output channel.

We said that convolutional filters are used to learn the correlations across nearby pixels. What would be the utility of 1×1 convolutions?

If there are 10 input channels and 25 output channels, how many parameters does a convolutional layer with a 5×5 kernel have? What is the size of the output feature map if convolution is performed with a stride of 2? Does stride change the number of parameters in a convolutional layer?

6.4 Translational equivariance using convolutions

We now discuss the most important reason for using convolutions in deep networks. Let us take our 1-dimensional signal x and translate it by ℓ units to the right

$$x'(t+\ell) := x(t).$$

You will see from the definition of convolution in Eq. (5.1) that the convolution also gets translated

$$(x' \star w)_k = \sum_{\tau=-\infty}^{\infty} x'_\tau w_{k-\tau} = \sum_{\tau=-\infty}^{\infty} x_{\tau-\ell} w_{k-\tau} = \sum_{s=-\infty}^{\infty} x_s w_{k-s-\ell} \quad (s = \tau - \ell) = (x \star w)_{k-\ell}$$

In other words, if you translate the signal by ℓ then the output of convolution is also translated by the same amount

$$(x' \star w)_{k+\ell} = (x \star w)_k.$$

This property is called equivariance. Equivariance also holds for 2D convolutions.

Translational equivariance is much more insightful for 2D images. Let us consider an example.

Equivariance to translations allows us to build an important property in a deep network. If we have a convolutional kernel that has weights such that the output is high for a certain object (star in adjoining picture, vertical/slanted strips in your Gabor filter homework), the output of a convolutional layer is such that the features also "move" if the input moves in the receptive field.
We can easily build a binary classifier using such equivariant features. If we want to build a star classifier, we simply check if some features in the output are large after convolution, e.g., we check if the largest feature in the 2D-feature map is greater than some pre-determined threshold

$$f(x, w) := \mathbf{1}\left\{ \max_{ij} \{(x \star w)_{ij}\} \geq \epsilon \right\}.$$

The pre-activation features of a convolutional layer are sometimes called the feature map.

6.5 Pooling to build translational invariance

We would like to build a classifier such that if the object moves to some other location in the input image, the output of the classifier remains unchanged, i.e., the deep network detects a test image as a cat even if it is in some other part of the image in the training data. Equivariance is only one part of the story to doing so. Remember that the last layer in a deep network looks like

$$f(x, w) = \text{sign}\langle v, h^L \rangle = \text{sign}\left(\sum_{i=1}^{p} v_i h_i^L \right).$$

Even if the features h^L are equivariant when the input x is translated in the 2D plane, the inner product $\langle v, h^L \rangle$ cannot be equivariant. Essentially, if a few weights v_i are trained to check for objects like cat/dog in one particular part of the image, even if the features h^L move accordingly, the output $\langle v, h^L \rangle$ need not be constant because the weights v_i at those new locations of features may be different.
In other words, we want features of a deep network to be invariant to translations in the input.

Making the weights of the top layer v all equal to 1 will solve this problem, but this is of course a very poor classifier. It smears the entire input signal h^L together by just averaging the features and therefore does not have much discriminative power; it cannot easily build a multi-class classifier for instance.

Pooling is an operation that smears out the features locally in the neighborhood of each pixel.
We can use our idea of setting all the weights to 1 to get what is called the average pooling operation. It is a linear operation and equivalent to convolving the input features using a kernel

$$w_{\text{avg-pool}} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}. \tag{5.5}$$

The average-pooling kernel is fixed during training and does not have any weights, otherwise it would be just another convolutional kernel.

! Average pooling blurs the image. We saw this in the example in Section 5.1.2. Such blurring at intermediate layers gives some translational invariance by smearing out the features.

Average pooling does not solve our problem of making the features invariant; the smeared out version simply moves less than ℓ when the input translates by ℓ. If we add many average pooling layers at various stages in a deep network, we make the features move even less and this may be sufficient to allow for weights v to be discriminative.
Max-pooling is another operation that builds invariance. It takes in an input $x \in \mathbb{R}^{w \times h}$ and computes

$$(\text{max-pool}(x))_{ij} = \max_{k \le s \le k} \max_{k \le t \le k} x_{i-s, j-t}$$

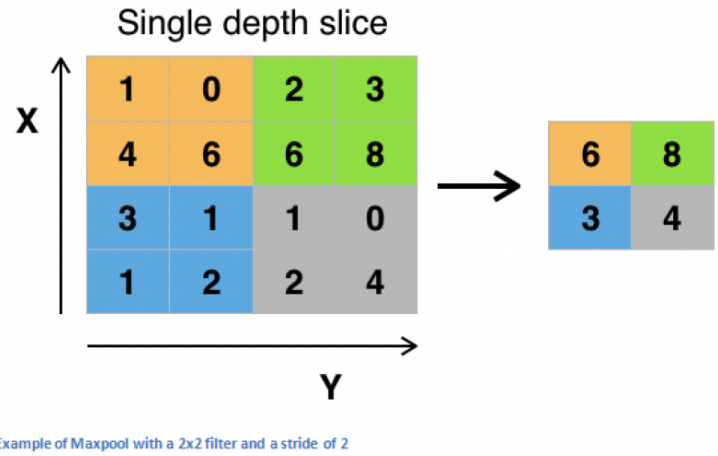


Figure 6.12: Max-pooling with a 2×2 kernel and a stride of 2 reduces the size of the input image by half. A stride of 1 would preserve the image size but would give less invariance.

This is a clever way of building invariance, you simply take the maximum value of the input in a window of size $k \times k$, so even if the input translates by k pixels in either direction, the output of a max-pooling layer remains the same. If we add multiple max-pooling layers at intermediate depths in a deep network, we achieve translational invariance in a convolutional neural network.

> Does max-pooling make sense for a fully-connected network? There is no equivariance property in such a network, so even if we do perform max-pooling, it is just like another activation function operating on the features.

Remark 5.6 (Max-pooling destroys information). As we see in Fig. 5.4, max-pooling destroys a lot of information in the input image. The result of max-pooling is a much smaller feature map. This results in a large loss of information in the input data and often leads to a loss of discriminative power, i.e.,

accuracy, during training. This trade-off between building a classifier that is invariant to changes in the input and discriminative enough to distinguish between many different categories is fundamental.

> We have talked about invariance to translations in this lecture. Images taken from a fish-eye camera are such that objects rotate in the field of view. Can you think of a trick to build invariance to rotations?

Max-pooling has a side-benefit, it reduces the number of operations in a deep network and the number of parameters by sequentially reducing the size of the feature map with layers. This is useful because a typical image you get from an autonomous car is easily about 10MP (10^7 pixels) and we need to boil it down into, say, 10 categories that are relevant to driving, i.e., $h^L \in \mathbb{R}^{10}$. Max-pooling is very useful for this, with the caveat that too much pooling will dramatically reduce the signal in the input image.

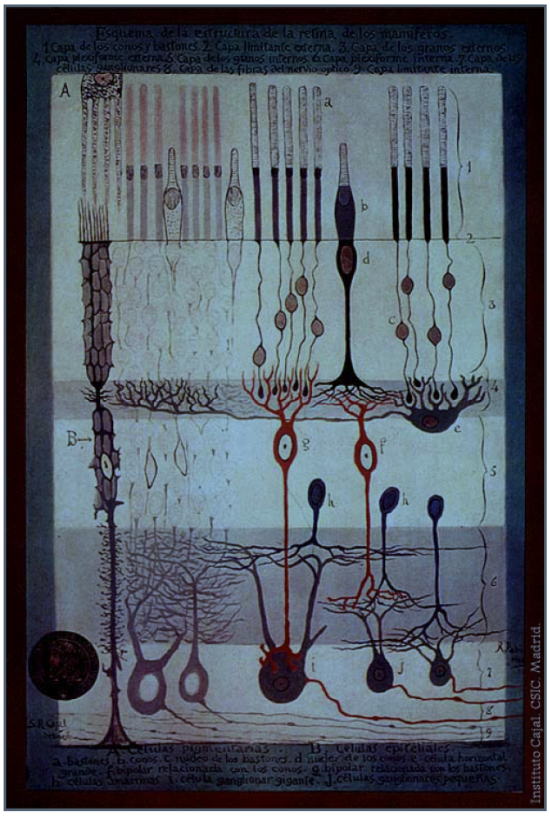


Figure 6.4: **Motivating from biology** A picture of the neurons in the retina drawn by Santiago Ramón y Cajal using a microscope in the 1900s. The mammalian retina circuits in the retina are hard-wired at birth because being able to see is so important to survival; there is no learning in the retina itself although there is a clear hierarchy of neurons that successively process information. Later parts of the visual cortex get learned during your lifetime. The retina transcribes photons that are incident upon the eye using rod cells (function better in low light) and cone cells (function better in bright conditions). This is further processed by "bipolar" cells into action potentials, or "spikes". Amacrine cells make lateral, inhibitory connections to removes redundancy in the stimuli. Ganglion cells create $\sim$ 20 visual features (edges/spots, local motions at $(90^o/120^o)$ angles, colors, etc.). Altogether, $\sim$ 80 types of neurons transmit $\sim$ 10 Mbps of information to the brain. These neurons are surprisingly similar to each other, e.g., all cell types fire at 4 $\sim$ 8 Hz and different ganglion cells learn highly redundant features. Read Balasubramanian (2015) for an exciting description of why neural circuits are wired the way they are.

Chapter 7

Classification Loss Functions

Reading

1. Bishop Chapter 5.5.3, 4.3
2. Bishop DL Chapter 6.4, 9.1
3. Goodfellow Chapter 7.4

We next discuss the various loss functions that are typically used for training neural networks. As usual, we are given a dataset

$$D = \{(x^i, y^i)\}_{i=1,\ldots,n} \tag{7.1}$$

7.1 Cross-Entropy loss

We next discuss the case when the targets are categorical and we wish to train a discriminative model that classifies the input into one of these m categories

$$y \in \{1, \ldots, m\}.$$

One hot encoding. An alternative representation of the targets in classification is the one-hot encoding where y is transformed to

$$\text{one-hot}(y) = e_y \in \mathbb{R}^m;$$

the vector e_y has a 1 at the yth element and zeros everywhere else.

Predicting class probabilities. Instead of using the regression loss by treating y as a real-valued quantity, it is more natural to predict the log-probability $\log p(k \mid x)$ for every category k and predict the category using

$$f(x; w) = \arg\max_k \log p_w(k \mid x).$$

Just like we denoted the raw predictions of the model by $\hat{y}$ in logistic regression, we will denote

$$\mathbb{R}^m \ni \hat{y} = v^\top \sigma\Big(S_L^\top \cdots \sigma\Big(S_2^\top \sigma\Big(S_1^\top x\Big)\Big)\Big), \tag{7.2}$$

where $v \in \mathbb{R}^{p \times m}$. As we saw in a previous Chapter (Section 5.1.1), $\hat{y}$ are also called *logits*. Observe that the logits $\hat{y}$ are simply vectors in $\mathbb{R}^m$.
How can we transform these logits to obtain $\log p_w(k \mid x)$ for all $k \in \{1, \dots, m\}$ as the output of the model?

Logistic loss. Linear logistic regression has a scalar output $\hat{y} \in \mathbb{R}$, which is interpreted as the log-odds of the class probabilities

$$\log \frac{p(y=1 \mid x)}{p(y=0 \mid x)} = w^\top x. \tag{7.3}$$

This expression can be rewritten as $p(1 \mid x) = \text{sigmoid}(\hat{y})$. The likelihood of the data $\{(x^i, y^i)\}_{i=1}^n$ under this model, for $y^i \in \{0, 1\}$, is

$$p_w\Big((x^1, y^1), \dots, (x^n, y^n)\Big) = \prod_{i=1}^n p_w(1 \mid x^i)^{y^i} p_w(0 \mid x^i)^{1-y^i}. \tag{7.4}$$

> We saw a different expression for the logistic loss in Section 3.2,
>
> $$\ell_{\text{logistic}}(w) = \log\left(1 + e^{-y\hat{y}}\right). \tag{7.5}$$
>
> What is the difference?

Maximizing this probability (MLE) is equivalent to minimizing the negative log-likelihood,

$$\ell_{\text{logistic}}(w) = -\log p_w\Big((x^1, y^1), \dots, (x^n, y^n)\Big) \tag{7.6}$$

$$= -\sum_{i=1}^n \Big[y^i \log p_w(1 \mid x^i) + (1 - y^i) \log p_w(0 \mid x^i)\Big]. \tag{7.7}$$

In other words, the logistic loss is simply maximum-likelihood estimation for the model in Eq. 7.3.

Binary Cross Entropy Loss Let us return to neural networks and multi-class classification. Imagine that each logit of a neural network in Eq. 7.2 acts independently, i.e., it predicts whether class k is present in the input or not without paying heed to what the other logits predict. This is not very prudent; for instance, if we know beforehand that there is only one object in the input image, then such a classifier is likely to produce many false positives. Nevertheless,

observe that this is exactly equivalent to running m independent binary logistic classifiers with the same feature representation $h_L \in \mathbb{R}^p$.
We can write the loss for such a classifier succinctly as

$$\ell_{\text{bce}}(w) = -\sum_{k=1}^{m} \text{one-hot}(y)_k \log p_w(k \mid x). \tag{7.8}$$

If the ground-truth labels y_i are such that there is only one class in each input image, all entries of one-hot(y_i) corresponding to the other categories will be zero. Consequently, this loss penalizes only the output of one of the m independent logistic classifiers.

7.2 Softmax Layer

Observe that our classifier, which employs m binary logistic classifiers for predicting all the categories independently, does not predict a valid probability distribution because

$$\sum_{k=1}^{m} p_w(k \mid x)$$

is not always equal to 1. We can, however, posit that the model predicts logits $\hat{y}$ that are proportional to the log-probabilities,

$$\log p_w(k \mid x) \propto \hat{y}_k,$$

which implies

$$p_w(k \mid x) = \frac{e^{\hat{y}_k / T}}{\sum_{k'=1}^{m} e^{\hat{y}_{k'} / T}}. \tag{7.9}$$

The resulting quantity $p_w(k \mid x)$ is a valid probability distribution over k because it sums to 1. This operation—taking the logits $\hat{y}$ and constructing probabilities from them—is called the *softmax* operator. The constant T in Eq.7.9 is called the *temperature*. A large value of T results in a smoother probability distribution $p_w(k \mid x)$ because the individual values of the logits matter less. A small value of T results in a very large weight for the largest logit due to the exponential, and the distribution $p_w(k \mid x)$ becomes highly peaked. In PyTorch, the temperature is set to $T = 1$ by default.

You will often see people refer to

$$\log \sum_{k'=1}^{m} e^{\hat{y}_{k'}/T}$$

as the "softmax" of the vector $\hat{y}$. This is actually a more appropriate usage of the term, since

$$\log \sum_{k=1}^{m} e^{\hat{y}_k/T} \approx \max_k \hat{y}_k$$

when one entry of $\hat{y}$ is much larger than the others, or when $T \to 0$. We will, however, use the word "softmax" to refer to the operation of transforming $\hat{y}$ into $p_w(k \mid x)$, as we do not require this softened version of the max operator.

The cross-entropy loss is now simply the maximum-likelihood loss after the softmax operation:

$$\ell_{\text{ce}}(w) = -\sum_{k=1}^{m} \text{one-hot}(y)_k \log p_w(k \mid x) \tag{7.10}$$

$$= -\frac{\hat{y}_y}{T} + \log\left(\sum_{k'=1}^{m} e^{\hat{y}_{k'}/T}\right). \tag{7.11}$$

Observe that the logit corresponding to the true class $\hat{y}_y$ is being pushed higher; at the same time, if the logits of the incorrect classes are large, they are pulled down through the summation term. This is an important point to keep in mind: the cross-entropy loss after softmax affects *all* logits, not just the logit of the correct class.

7.2.1 Label smoothing

Label smoothing is a trick that alleviates over-fitting: instead of using a one-hot encoding of the true label y, it uses the encoding

$$\text{label-smoothing}(y)_k = \begin{cases} 1-\delta & \text{if } k = y, \\ \frac{\delta}{m-1} & \text{else.} \end{cases}$$

The cross-entropy loss with this new encoding is

$$\ell_{\text{label-smoothing-ce}}(w) = -(1-\delta)\log p_w(y \mid x) - \frac{\delta}{m-1}\sum_{k \neq y} \log p_w(k \mid x).$$

7.2.2 Multiple ground-truth classes

If there are multiple classes present in the input image, we can use the vector

$$\text{multi-hot}(y) = \sum_k e_k$$

for all present classes k and set

$$\ell_{\text{bce}}(w) = -\sum_{k=1}^{m} \text{multi-hot}(y)_k \log p_w(k \mid x).$$

We can also use this trick in the cross-entropy loss after the softmax operator, but it will not work well because the softmax operator is designed to amplify only the largest logit in $\hat{y}$. If we attempted this, the network would still be incentivized to predict only one class instead of all classes.

Chapter 8

Practical ML : Validation and Hyperparameters

8.1 Recap: Test Risk vs. Empirical Risk

Recall our fundamental goal in machine learning: we want to find a hypothesis h from a hypothesis set $\mathcal{H}$ that minimizes the test error (or generalization error), $R(h)$. This risk is the expected loss over the entire unknown data distribution P:

$$R(h) = \mathbb{E}_{(x,y)\sim P}\big[e(h(x), y)\big],$$

where e is an error measure. For example,

$$e(h(x), y) = (h(x) - y)^2$$

for regression and

$$e(h(x), y) = \mathbf{1}[h(x) \neq y]$$

for classification.
We cannot compute this because we do not know P. Instead, we are given a training set

$$D = \{(x^i, y^i)\}_{i=1}^n$$

of n examples sampled IID from P. We use this to compute the empirical risk (or training error),

$$\hat{R}_n(h) = \frac{1}{n}\sum_{i=1}^{n} e(h(x^i), y^i).$$

8.1.1 Scenario 1: For a fixed hypothesis h

If we pick a hypothesis h before seeing the data, is $\hat{R}_n(h)$ a good proxy for $R(h)$?

- Yes. The empirical risk $\hat{R}_n(h)$ is an unbiased estimator of the test error $R(h)$. By linearity of expectation:

$$\mathbb{E}_D[\hat{R}_n(h)] = \mathbb{E}_D\left[\frac{1}{n}\sum_{i=1}^{n} e(h(x^i), y^i)\right] \tag{8.1}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{(x^i,y^i)\sim P}[e(h(x^i), y^i)] \tag{8.2}$$

$$= R(h) \tag{8.3}$$

  In the second and third equalities, we critically used the fact that h does not depend on D.

- Furthermore, by concentration inequalities (like Hoeffding's), the empirical risk concentrates around the test error as n grows:

$$|\hat{R}_n(h) - R(h)| \leq O\left(\frac{1}{\sqrt{n}}\right).$$

8.1.2 Scenario 2: For a data-dependent hypothesis $\hat{h}$

In practice, we do not fix h. We use the data D to find our hypothesis.
Now, is the training error $\hat{R}_n(\hat{h})$ an unbiased estimator of $R(\hat{h})$?

- No. The hypothesis $\hat{h}$ was specifically chosen based on the data D. It is no longer independent of the data used to evaluate it. Because of that, we cannot repeat what we did in Eq. 8.1.

Does then the training error $\hat{R}_n(\hat{h})$ tell us anything about $R(\hat{h})$? Yes, but we pay a penalty that relates to the complexity term of the hypothesis set to which $\hat{h}$ belongs. This is our generalization bound.

8.2 Generalization, Complexity, and Overfitting

To understand the test error $R(\hat{h})$, we need a generalization bound. For a data-dependent $\hat{h}$ chosen from a hypothesis set $\mathcal{H}$, the bound (with high probability) looks like:

$$R(\hat{h}) \leq \hat{R}_n(\hat{h}) + O\left(\sqrt{\frac{\text{Complexity}(\mathcal{H})}{n}}\right).$$

For a finite hypothesis set $|\mathcal{H}|$, the complexity term is $\log|\mathcal{H}|$. For linear models, the complexity is related to the number of parameters p, so

$$R(\hat{h}) \leq \hat{R}_n(\hat{h}) + O\left(\sqrt{\frac{p}{n}}\right).$$

Overall, this reveals the fundamental tradeoff of machine learning.

- To minimize the training error $\hat{R}_n(\hat{h})$, we need a more complex model.
- Increasing complexity increases the complexity penalty.
- The test risk is the sum of these two terms and follows a U-shaped curve.
- Underfitting region: simple model, high training error, high test error.
- Overfitting region: complex model, low training error but high test error.
- Sweet spot: optimal complexity minimizing test error.

8.2.1 What does the generalization bound tell us about finding a good $\hat{h}$?

We built our algorithms for finding a good hypothesis on the intuition gained from the generalization bound. Since the risk is a sum of two terms—the training error and the complexity term—our algorithms are based on the idea of combining empirical risk minimization (ERM), which targets minimizing the former, and regularization, which targets minimizing the latter by favoring simpler hypotheses. The explicit form of regularization that we have seen is in the form of *regularized ERM*.

$$\min_w \frac{1}{n} \sum_{i=1}^{n} \ell(h_w(x^i), y^i) + \lambda \Omega(w),$$

where $\Omega(w) = \|w\|_2^2$ or $\Omega(w) = \|w\|_1$.
There are also other heuristic forms of accounting for the complexity penalty that, loosely speaking, can be thought of as forms of regularization. Here, "regularization" is used in its broader sense as an algorithmic means of combating overfitting, that is, balancing the complexity of the model so that the training error is small while at the same time the hypothesis is "simple" enough so that we do not pay much in terms of the complexity penalty and the training error remains close to the test error. Other such common techniques (not going into detail, but important to be aware of) include:

- Weight decay: Comes with a discounting factor λ that is analogous to the λ in parameter regularization we saw above.
- Data augmentation : Creating more training data by applying transformations (e.g., rotating and cropping images).
- Early stopping : Selecting at which iteration t to stop gradient descent.
- Dropout : (Common in neural networks.) Comes with a hyperparameter π, the probability (or rate) that a given neuron is temporarily "dropped" (ignored) during a training step.

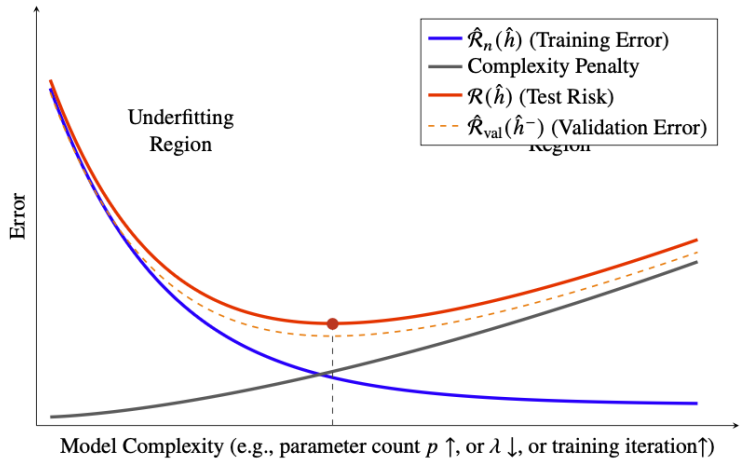


Figure 8.1: The fundamental tradeoff between training error and model complexity. The (unseen) Test Risk(red)is what we want to minimize. We use the computable Validation Error(dashed orange), rather than the Training error,to find the "sweet spot."

The U-shaped overfitting curve shown in Figure 8.1 is a general concept that applies to all of these hyperparameters. The "Model Complexity" axis can represent any of the following controls:

- **Regularization strength λ:** Complexity is inversely related to λ, so the x-axis could be $1/\lambda$.
  - Large λ (low complexity): $\mathbf{w} \approx \mathbf{0}$. The model is too simple $\rightarrow$ underfitting.
  - Small λ (high complexity): The model fits the training data perfectly, including noise $\rightarrow$ overfitting.
- **Training iterations t (early stopping):** Complexity is directly related to t.
  - Small t: The model has not learned the signal yet $\rightarrow$ underfitting.
  - Large t: The model has learned the signal and has now begun to memorize the noise $\rightarrow$ overfitting.

8.3 The Problem of Model Selection

We have just argued that, in practice, we control complexity using a wide variety of hyperparameters. A *hyperparameter* is a setting for our learning algorithm that is not learned from the data itself, but must be chosen beforehand. Finding

the "sweet spot" on the complexity curve means finding the optimal value of these hyperparameters.
Common examples include:

- The regularization parameter λ in regularized ERM.
- The choice of regularizer (e.g., L_1 vs. L_2).
- The degree Q of a polynomial feature map or kernel parameters (e.g., γ for an RBF kernel).
- The number of iterations t for early stopping.
- The weight-decay factor (which acts as a λ).
- The dropout rate π in a neural network.
- The data augmentation strategy (i.e., whether to use it and which transformations to apply).
- Optimization parameters such as the learning rate η, which indirectly affect the final model by interacting with other regularizers such as early stopping.

We need a way to choose the best combination of these hyperparameters. As we have established, we cannot use the training error $\hat{R}_n(\hat{h})$, because it will always prefer the most complex model (e.g., $\lambda = 0$, $t \to \infty$, or dropout rate $\pi = 0$), which leads directly to overfitting.

8.4 The Validation Set Protocol

Key Result. Given m candidate hyperparameter settings:

1. **Split:** Randomly split the original dataset D (of size n) into two disjoint sets:
   - A training set D_{train} (e.g., $n - k$ examples).
   - A validation set D_{val} (e.g., k examples).
2. **Train:** For each $j \in [m]$ of the m candidate settings, train a hypothesis $\hat{h}_j^-$ using only the training set D_{train}.
3. **Select:** Evaluate all m hypotheses on the validation set D_{val}. Compute the validation error for each:

$$\hat{R}_{\text{val}}(\hat{h}_j^-) = \frac{1}{k} \sum_{(x_i, y_i) \in D_{\text{val}}} e\left(\hat{h}_j^-(x_i), y_i\right). \tag{8.4}$$

Select the hypothesis $\hat{h}^*$ (and its corresponding hyperparameters λ^*) that achieves the lowest validation error:

$$\hat{j}^* = \arg\min_{j \in \{1,\ldots,m\}} \hat{R}_{\text{val}}(\hat{h}_j^-). \tag{8.5}$$

4. **Retrain (Heuristic):** Take the winning hyperparameters λ^* from the selected model and retrain a new, final hypothesis on the entire original dataset
$$D = D_{\text{train}} \cup D_{\text{val}}.$$
This final model is the one that is deployed.

This retraining step is a heuristic: we hope that the hyperparameters that were optimal for $n - k$ data points are also (nearly) optimal for n data points, and that the final model will be better since it is trained on more data.

8.5 Why Does Validation Work?

The Key Insight. When we evaluate $\hat{h}_j^-$ on D_{val}, that hypothesis is fixed with respect to the validation data. It was trained only on D_{train}, and $D_{\text{train}} \cap D_{\text{val}} = \varnothing$. The validation set is truly "unseen" data for $\hat{h}_j^-$.

Therefore, for any single candidate hypothesis $\hat{h}_j^-$, its validation error $\hat{R}_{\text{val}}(\hat{h}_j^-)$ is an unbiased estimator of its test error $R(\hat{h}_j^-)$:

$$\mathbb{E}_{D_{\text{val}}}\left[\hat{R}_{\text{val}}(\hat{h}_j^-)\right] = R(\hat{h}_j^-). \tag{8.6}$$

This is exactly analogous to what we did in Eq. 8.1. Even more strongly, we can apply a concentration inequality just as in Scenario 1, but this time using the k validation points. With high probability,

$$R(\hat{h}_j^-) \leq \hat{R}_{\text{val}}(\hat{h}_j^-) + O\left(\frac{1}{\sqrt{k}}\right). \tag{8.7}$$

This provides a computable and reliable estimate of the true generalization error for each candidate model. When we select the best of m models, we can apply a union bound, which yields

$$R(\hat{h}^*) \leq \hat{R}_{\text{val}}(\hat{h}^*) + O\left(\sqrt{\frac{\log m}{k}}\right). \tag{8.8}$$

This reveals the fundamental tradeoff of validation:

- We need k to be large so that our estimate is reliable (the $O(\cdot)$ term is small).
- We need $n - k$ to be large so that the hypotheses $\hat{h}_j^-$ we train are good in the first place.

Practical Tip. A common rule of thumb is to split the data 80% for D_{train} ($n - k$) and 20% for D_{val} (k). However, if the total dataset size n is small, this tradeoff becomes difficult, and "wasting" 20% of the data is too costly. This motivates more advanced techniques such as cross-validation.

8.6 Practical Validation Strategies

8.6.1 K-Fold Cross-Validation

1. Split D into K folds.
2. For each hyperparameter setting:
    - Train on $K - 1$ folds.
    - Validate on the remaining fold.
    - Average the validation errors.
3. Select hyperparameters with lowest average error.

8.6.2 Leave-One-Out Cross-Validation

This is the case $K = n$.

- Pro: very accurate estimate.
- Con: computationally infeasible for large n.

Chapter 9

Regression and Regularization

9.1 From Classification to Regression

So far, our entire focus has been on classification: predicting a discrete label, $y \in \{-1, +1\}$. We now turn to regression, where our goal is to predict a continuous value, $y \in \mathbb{R}$. Think of predicting a house price, tomorrow's temperature, or a stock's value. Let's follow the same formal pipeline we developed for classification. Our goal is to find a hypothesis $h : \mathbb{R}^d \to \mathbb{R}$ that makes accurate predictions.

9.1.1 Risk, Empirical Risk, and Parameterization

The Goal: Minimize Test Error (True Risk). Our "holy grail" is to find the hypothesis h from a hypothesis class $\mathcal{H}$ that minimizes the true risk, which for regression is the expected squared error:

$$R(h) = \mathbb{E}_{(x,y)\sim P}(h(x) - y)^2.$$

The Proxy: Minimize Training Error (Empirical Risk). Since we don't know the true distribution P, we minimize the empirical risk over the dataset $D = \{(x^i, y^i)\}_{i=1}^n$:

$$\min_{h\in\mathcal{H}} \left\{ \widehat{R}_n(h) := \frac{1}{n} \sum_{i=1}^{n} (h(x^i) - y^i)^2 \right\}$$

The Strategy: Parameterize Hypotheses. We parameterize the hypothesis class using a parameter vector $\theta \in \mathbb{R}^p$:

$$\min_{\theta\in\mathbb{R}^p} \left\{ L(\theta) := \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x^i) - y^i)^2 \right\}$$

9.1.2 Linear Regression: Model and Objective

We start with the simplest parameterization:

$$h_{w,b}(x) = w^\top x + b.$$

Absorbing the bias into w using augmented features, we write:

$$h_w(x) = w^\top x.$$

The ERM objective becomes:

$$\min_{w \in \mathbb{R}^d} L_{\text{mse}}(w) := \frac{1}{n} \sum_{i=1}^{n} (w^\top x^i - y^i)^2$$

This problem is known as Linear Regression or Ordinary Least Squares (OLS). The loss is quadratic in w, hence convex.

9.2 Solving Linear Regression

Linear Regression has been around for a long time, and over the years people have developed several well-known and elegant ways to solve it. These methods are the building blocks behind many modern machine learning techniques. Now's your chance to dive in, understand how they work, and appreciate why they've stood the test of time. Let's get started and have some fun with it!

9.2.1 Method 1: Gradient Descent

We have seen that gradient descent(GD) is a very general optimization algorithm. To use it, we just need the gradient of the loss function. For a single data point (x^i, y^i):

$$\nabla_w \ell_i(w) = 2(w^\top x^i - y^i)x^i.$$

The full gradient is:

$$\nabla_w L_{\text{mse}}(w) = \frac{2}{n} \sum_{i=1}^{n} (w^\top x^i - y^i)x^i.$$

Gradient descent update:

$$w_{t+1} = w_t - \eta \frac{2}{n} \sum_{i=1}^{n} (w_t^\top x^i - y^i)x^i.$$

Stochastic gradient descent:

$$w_{t+1} = w_t - \eta_t 2(w_t^\top x_{i_t} - y_{i_t})x_{i_t}.$$

This is a perfectly valid and very common way to solve linear regression, especially when η is very large.

9.2.2 Method 2: The Normal Equations

Unlike Logistic Regression (which had no closed-form solution), Linear Regression does. Because the objective is convex and differentiable, we know the global minimum occurs at the point where the gradient is zero. Let's just solve for it directly!
Recall the mean squared error loss:

$$L_{\text{mse}}(w) = \frac{1}{n}\sum_{i=1}^{n}(w^\top x^i - y^i)^2.$$

Taking the gradient with respect to w, we obtain:

$$\nabla_w \widehat{L}_{\text{mse}}(w) = \frac{2}{n}\sum_{i=1}^{n}(w^\top x^i - y^i)x^i$$

Setting the gradient equal to zero yields:

$$\frac{2}{n}\sum_{i=1}^{n}(w^\top x^i - y^i)x^i = 0.$$

To simplify the expression, we introduce matrix notation:

- Let $X \in \mathbb{R}^{n\times d}$ be the data matrix whose i-th row is $(x^i)^\top$.
- Let $y \in \mathbb{R}^n$ be the vector of labels.

The term $\sum_{i=1}^{n} w^\top x^i x^i$ can be written as $(X^\top X)w$. The term $\sum_{i=1}^{n} y^i x^i$ can be written as $X^\top y$. (Check the dimensions: $X^\top$ is $d \times n$, X is $n \times d$, so $X^\top X$ is $d \times d$, w is $d \times 1$, $X^\top y$ is $d \times 1$) Using this notation, we can rewrite the condition above as:

$$(X^\top X)w - X^\top y = 0.$$

This leads to the *normal equations*:

$$X^\top X\, w = X^\top y.$$

If the matrix $X^\top X$ is invertible, the solution is unique and given by:

$$w^* = (X^\top X)^{-1}X^\top y.$$

This closed-form solution is computationally efficient when the number of features d is small, since computing the inverse of a $d \times d$ matrix requires $\mathcal{O}(d^3)$ operations.

9.2.3 Overparameterized Learning: $d > n$

But what if d is large, in particular $d > n$? Since we have more features than number of examples in the training set ($d > n$), we say that the learning system is overparameterized. In terms of the underlying linear algebra, the linear system of equations $Xw = y$ is underdetermined. Specifically, the $d \times d$ matrix

$$X^\top X = \sum_{i \in [n]} x_i x_i^\top$$

is rank-deficient and not invertible. What this means is that there isn't one unique w^* that solves the problem. In fact, there are infinitely many solutions w that can get a perfect training error of zero. All these solutions satisfy $Xw = y$. In ML lingo, we say that the solution interpolates or fits the (training) data. So, which of these infinitely many solutions should we choose?

Remember, that in the regime $d > n$, there is the risk of overfitting, i.e. suffering from bad generalization (large train-test error gap) because of a large complexity term (that scales as d/n^2). Thus, we need a way to control complexity. Intuitively, we look for parameter vector w's that minimizes the training error while at the same time being "simple" enough. But what does it mean for a vector w to be simple?

One intuition is as follows: suppose w is sparse, that is it has only very few non-zero entries. Say only $k \ll d$ of its entries are non-zero while the rest are zero. Then, the prediction

$$h_w(x) = w^\top x = \sum_{j=1}^{d} w_j x_j$$

is effectively only taking linear combination of a small k-subset of the features $x_1, x_2, \ldots, x_d$. Thus, the effective dimension of the parameter vector is not d, but k.
With this intuition, a sensible goal becomes that of computing the parameter vector that has the smallest number of non-zero entries subject to perfectly fitting (interpolating) the training data. As an optimization problem this can be expressed as

$$\min_{w \in \mathbb{R}^d} \mathbf{nnz}(w) \quad \text{subject to} \quad Xw = y$$

where $\mathbf{nnz}(w)$ returns the number of non-zeros in w.
Unfortunately, this objective function is combinatorial, highly non-convex and hard to optimize efficiently (without having to do a brute-force search over say all k-subsets of coefficients that might fit the data for all small values of k).

To the rescue comes again our technique of convex relaxation. We can relax the optimization by substituting the non-convex objective $\mathbf{nnz}(w)$ with a convex surrogate. A popular choice of such a surrogate is the ℓ_2-norm $\|w\|_2$. This is

called the minimum-norm solution or minimum-norm interpolator w_{mm}. Our new, now convex surrogate problem of finding a simple parameter vector that fits the data is:

$$\min_{w \in \mathbb{R}^d} \|w\|_2^2 \quad \text{subject to} \quad Xw = y$$

> What is convexity ? We will discuss this soon, in the next chapter. Let's put a bookmark on this term for now.

This is a convex optimization problem (a quadratic objective with linear constraints). Conveniently, this optimization has a simple unique solution that is given in closed form:

$$w_{mm} = X^\top (XX^\top)^{-1} y \tag{9.1}$$

A few things to note:

1. **Invertibility of the Gram Matrix:** This solution assumes that the $n \times n$ matrix $K = XX^\top$ is invertible. This matrix, known as the Gram matrix, has entries $(K)_{ij} = x_i^\top x_j$, representing the inner products between all pairs of training samples. In the overparameterized regime where $d > n$, this matrix is typically full rank (rank n) and thus invertible, provided the n data points are linearly independent.

2. **Characterizing the Solution Space:** The vector w_{mm} is an interpolating solution, meaning it achieves zero training error:

   $$Xw_{\text{mm}} = X(X^\top (XX^\top)^{-1} y) = (XX^\top)(XX^\top)^{-1} y = y$$

   However, it is not the only one. Any vector w of the form $w = w_{\text{mm}} + v$ is also an interpolating solution, provided v is in the nullspace of X (i.e., $Xv = 0$), since $Xw = X(w_{\text{mm}} + v) = Xw_{\text{mm}} + Xv = y + 0 = y$. Given $\text{rank}(X) \leq n < d$, the nullspace is non-trivial, with $\dim(\text{null}(X)) = d - \text{rank}(X) \geq d - n$, implying an infinite set of solutions.

3. **Uniqueness of the Minimum-Norm Solution:** Points (1) and (2) together prove that w_{mm} is the unique solution to the minimum-norm optimization program. From (3), we know w_{mm} lies in the range space of $X^\top$. From (2), we know all other solutions are $w = w_{\text{mm}} + v$, where v is in the nullspace of X. A fundamental theorem of linear algebra states that the range space of $X^\top$ and the nullspace of X are orthogonal complements. Therefore, $w_{\text{mm}} \perp v$. By the Pythagorean theorem, the norm of any solution w is:

   $$\|w\|_2^2 = \|w_{\text{mm}} + v\|_2^2 = \|w_{\text{mm}}\|_2^2 + \|v\|_2^2$$

   This quantity is clearly minimized when $\|v\|_2^2 = 0$, which implies $v = 0$. Thus, the unique minimum-norm solution is precisely $w = w_{\text{mm}}$.

Derivation of the Exact Solution

Why does the solution take the unique form in Equation 9.1 ? We explain why replacing the non-convex sparsity objective with an ℓ_2-norm leads to a convex problem with a unique closed-form solution.

From Sparsity to Convex Relaxation. The original goal is to find a simple interpolating solution:

$$\min_w \mathbf{nnz}(w) \quad \text{subject to } Xw = y,$$

where $\mathbf{nnz}(w)$ denotes the number of non-zero entries of w.

This objective directly encodes sparsity but is combinatorial, non-convex, and NP-hard to optimize. To obtain a tractable problem, we use *convex relaxation*, replacing $\mathbf{nnz}(w)$ with a convex surrogate. A common hierarchy of relaxations is:

$$\|w\|_0 \;\approx\; \text{nnz}(w) \quad \longrightarrow \quad \|w\|_1 \quad \longrightarrow \quad \|w\|_2^2.$$

While the ℓ_2-norm does not promote sparsity directly, it penalizes large coefficients and controls the overall complexity of the solution.

Convexity of the Relaxed Problem

The relaxed optimization problem is:

$$\min_{w \in \mathbb{R}^d} \|w\|_2^2 \quad \text{subject to} \quad Xw = y.$$

Objective. The function $\|w\|_2^2 = w^\top w$ is quadratic with Hessian $2I \succ 0$, hence strictly convex.

Constraints. The constraint $Xw = y$ is linear and defines an affine subspace, which is a convex set.

Conclusion. Minimizing a strictly convex function over a convex set yields a unique global minimizer. Therefore, the problem is a convex optimization problem with a unique solution.

To solve the constrained problem, we form the Lagrangian:

$$\mathcal{L}(w, \alpha) = \frac{1}{2} w^\top w + \alpha^\top (y - Xw),$$

where $\alpha \in \mathbb{R}^n$ is the vector of Lagrange multipliers.

Taking the gradient with respect to w and setting it to zero yields:

$$\nabla_w \mathcal{L}(w, \alpha) = w - X^\top \alpha = 0,$$

which implies

$$w = X^\top \alpha.$$

Substituting this expression into the constraint $Xw = y$ gives:

$$X(X^\top \alpha) = y \quad \Longrightarrow \quad (XX^\top)\alpha = y.$$

Assuming X has full row rank, the matrix $XX^\top \in \mathbb{R}^{n\times n}$ is invertible, and therefore

$$\alpha = (XX^\top)^{-1}y.$$

Substituting back, we obtain the closed-form solution:

$$\boxed{w_{\text{mm}} = X^\top (XX^\top)^{-1}y.}$$

9.3 Loss Functions for Regression

MSE loss. If the labels are real-valued $y^i \in \mathbb{R}$, e.g., we are predicting the price of housing in Boston given features of the houses , we are solving a regression problem and the loss function to use for a deep network is also simply the regression loss.

$$\ell_{\text{mse}}(w) := \frac{1}{2}\big(f(x;w) - y\big)^2 \tag{9.2}$$

If you think about it carefully, it seems silly to add different dimensions of the input x using the weights w. Consider the case of

$$x = [\text{miles/gallon}, \text{number of other people with the same car}, \text{price of the car}].$$

The three elements of x are in totally different units and totally different scales. A popular trick to make things a bit more uniform for regression is take a logarithmic transformation of the input, i.e., fit a model to $\log x$ using the loss

$$\frac{1}{2}\big(f(\log x;w) - y\big)^2;$$

we can compute the logarithm element-wise for vector valued inputs.

Huber loss. The square-residual loss in Eq. 9.2 works in most cases but it does not work well if there are outliers in the data. Outliers are data in the training set that are noisy or did not come from the true model. In such cases, we can use the Huber loss. If the residual is $r = f(x;w) - y$, the Huber loss is

$$\ell_{\text{huber}}(w;\delta) = \begin{cases} \frac{1}{2}|r|^2 & \text{if } |r| \le \delta, \\ \delta\big(|r| - \frac{1}{2}\delta\big) & \text{else.} \end{cases}$$

Observe that this does not penalize the model egregiously if the predictions are bad ($|r| \ge \delta$) for a particular datum. Doing so prevents the outliers from

biasing the loss towards themselves and ruining the residuals for the other data.

> We can perform regression in a clever way: first set all weights $w_i = 0$ and iteratively allow a subset of the weights (say the ones that improve the residuals the most) to become non-zero; non-zero weights are fitted using ℓ_{mse}. This is known as forward selection. Backward selection starts with weights w^* which minimize ℓ_{mse} and iteratively prune the weights. Both forward and backward selection are techniques to fit a model w with sparse weights.

MAE loss. The absolute-error loss (or ℓ_1)

$$\ell_{\text{mae}}(w) = |f(x; w) - y|$$

has a similar motivation: it does not penalize the residual on the outliers.

> Using a subset-selection technique or the ℓ_{mse} loss with ℓ_1 regularization on the weights
>
> $$\frac{1}{2n} \sum_{i=1}^{n} \left(f(\log x^i; w) - y^i \right)^2 + \lambda \|w\|_1$$
>
> leads to sparse weights w. This makes the model more interpretable than a model fitted using ℓ_{mse} loss.

Variable importance. For linear models, another way to answer the same question is to fit two models, one with w_i fixed to zero and all other weights fitted using the MSE loss Eq. 9.2 and another model without fixing w_i; the difference between the average square residuals in the two cases is a measure of how important the feature x^i is for the prediction. These techniques are called variable importance methods. We can also undertake the same program for nonlinear models on non-image based data.

Quantile loss. The quantile loss is another simple trick to make the model more robust to outliers and get more information from the model than simply the prediction $f(x; w)$. Observe that if we have targets Y that are random variables with cumulative distribution function $F(y) = \mathbb{P}(Y \leq y)$, the τth quantile of Y is given by

$$Q_Y(\tau) = F^{-1}(\tau) = \inf\{y : F(y) \geq \tau\}, \quad \tau \in (0, 1).$$

We now learn a predictor for $Q_Y(\tau) = f(x; w)$. It turns out (you can try to

prove this) that this corresponds to the loss function

$$\ell_{\text{quantile}}(w;\tau) = \begin{cases} r(\tau-1) & \text{if } r<0, \\ r\tau & \text{else,} \end{cases} \tag{9.3}$$

$$= r\big(\tau - \mathbf{1}_{\{r<0\}}\big), \tag{9.4}$$

where $r = y - f(x;w)$ is the residual.

> The quantile loss is also called the pinball loss. Unlike the regression loss, it is highly asymmetric around the origin.

A standard technique is to fit multiple models using the quantile loss for different quantiles, say $\tau = 0.25, 0.5, 0.75$ and give multiple predictions of the target $f(x;w_\tau)$.

Chapter 10

Optimization and Gradient Descent

Reading

1. Bishop DL Chapter 7
2. The blog-post titled "Why momentum really works?" at https://distill.pub/2017/momentum

We have covered the cliff-notes of the practice of deep learning in the previous eight chapters. It is by no means a complete overview. The practice of deep learning is an enticing, mysterious, and sometimes frustrating enterprise. The more time you spend playing with code, the more you will learn about deep learning. New ideas are routinely discovered using very simple experiments that each of you is capable of running now.
As we discussed, there three main concepts in machine learning. First, the class of functions $f(x; w)$ that you use to make predictions, this is called the hypothesis class or the architecture. Second, the algorithm you use to find the best model in this class of functions that fits your data; this uses tools from optimization theory. Third is the generalization performance of your classifier. Machine Learning is about picking a good hypothesis class, finding the best model within this class and making sure that the model generalizes.

Goal of this chapter is to develop an understanding of optimization and generalization for more generic machine learning models first. It will end with an insight into understanding their interplay for deep networks. This has a different flavor, it is more theoretical. Our goal is to grasp the general concepts behind these theoretical results and understand the training process of deep networks better. This will also help us train deep networks much better in practice.

10.1 Convexity

Consider a function $\ell : \mathbb{R}^p \to \mathbb{R}$ that is convex, i.e., for any w, w' that lie in the domain (which is assumed to be a convex set) of ℓ and any $\lambda \in [0, 1]$ we have

$$\ell(\lambda w + (1-\lambda)w') \quad \leq \quad \lambda\ell(w) + (1-\lambda)\ell(w') \tag{10.1}$$

A function $\ell(w)$ is concave if $-\ell(w)$ is convex. If the function ℓ is continuous, it is enough to check this definition for a particular value of λ, say $\lambda = 1/2$ if you need to prove that a function is convex. Some examples of convex functions are

- powers w^α for $w > 0$ and $\alpha \geq 1$,
- powers of absolute values $|w|^\alpha$ for $w \in \mathbb{R}$ and $\alpha \geq 1$,
- exponential $\exp(w)$, negative logarithm $-\log(w)$ for $w \in \mathbb{R}$,
- affine functions $Aw + b$,
- quadratics $w^\top Aw + b^\top w + c$,
- norms $\|w\|_p = (\sum_i^p |w_i|^p)^{1/p}$ for $p \geq 1$, or $\|w\|_\infty = \max_k |w_k|$,
- log-sum-exp $f(w)_i = \log \sum_i \exp(w_i)$ for $w \in \mathbb{R}$.

The Cauchy-Schwarz inequality states that

$$\Big(\sum_i a_i^2\Big)\Big(\sum_i b_i^2\Big) \quad \geq \quad \Big(\sum_i a_i b_i\Big)^2.$$

A generalization of this is Holder's inequality which states that

$$\Big(\sum_i a_i^p\Big)^{1/p}\Big(\sum_i b_i^q\Big)^{1/q} \quad \geq \quad \Big(\sum_i |a_i b_i|\Big)$$

for any $1/p + 1/q = 1$

A couple of standard tricks that help prove that a function is convex. The first one is called "midpoint convexity". If a function is continuous, then showing that the definition of convexity is satisfied for $\lambda = 1/2$ is sufficient to prove that the function is convex. The proof is as follows.
Suppose we have

$$\ell\Big(\frac{w+w'}{2}\Big) \quad \leq \quad \frac{1}{2}\ell(w) + \frac{1}{2}\ell(w').$$

This can be used iteratively to show that

$$\ell\Big(\frac{w_1 + w_2 + \ldots w_{2^n}}{2^n}\Big) \quad \leq \quad \frac{1}{2^n}\sum_{k=1}^{2^n} \ell(w_k)$$

for n arguments $w_1, \ldots, w_n \in \mathbb{R}$. Now set $w_1 = \cdots = w_m = w$ and $w_{m+1} = \cdots = w_{2^n} = w'$. This gives

$$\ell\Big(\frac{m}{2^n}w + \Big(1 - \frac{m}{2^n}\Big)w'\Big) \quad \leq \quad \frac{m}{2^n}\ell(w) + \Big(1 - \frac{m}{2^n}\Big)\ell(w').$$

This proves the definition of convexity for ℓ for all $\lambda = m/(2^n)$. Now, dyadic rationals, i.e., numbers of the form $m/(2^n)$ are dense in the unit interval, i.e., any real number λ is arbitrarily close to a rational of the form $m/(2^n)$ for some m and n. Therefore, we can take the limit as $m/(2^n)' \lambda$ to see that the definition of convexity holds for any value of λ.
The second important trick is to observe that if a function $\ell : \mathbb{R}^p \to \mathbb{R}$ is convex, then it is convex "in any direction". Consider a function $g : \mathbb{R} \to \mathbb{R}$

$$g(\lambda) = \ell(w + \lambda(w' - w))$$

for any two points $w, w' \in \mathbb{R}^p$. Now see that, for all $\lambda \in [0, 1]$:

$$\begin{aligned} \ell((1-\lambda)w + \lambda w') \quad &\leq \quad (1-\lambda)\ell(w) + \lambda\ell(w') \\ g(\lambda) \quad &\leq \quad (1-\lambda)g(0) + \lambda g(1). \end{aligned}$$

Strictly convex functions Strictly convex functions have the property that for all $w \neq w'$ in the domain (which is assumed to be convex) and $\lambda \in (0, 1)$

$$\ell(\lambda w + (1-\lambda)w') < \lambda\ell(w) + (1-\lambda)\ell(w').$$

First-order condition for convexity If ℓ is differentiable, the definition of convexity in Eq. 10.1 is equivalent to the following first-order condition. A differentiable function ℓ with convex domain is convex iff

$$\ell(w') \quad \geq \quad \ell(w) + \langle \nabla\ell(w),\, w' - w \rangle \tag{10.2}$$

for all w, w' in the domain. Note that the first-order condition is equivalent to the definition of convexity in Eq. 10.1 for differentiable functions. The proof is long but easy; you can see for the proof. For strictly convex functions the inequality is strict

$$\ell(w') > \ell(w) + \langle \nabla\ell(w),\, w' - w \rangle.$$

Monotonicity of the gradient for convex functions The first-order condition for convexity gives a useful, and equivalent, characterization of the gradient. Write Eq. 10.2 for w, w' in two opposite directions

$$\begin{aligned} \ell(w) &\geq \ell(w') + \langle \nabla \ell(w'),\, w - w' \rangle \\ \ell(w') &\geq \ell(w) + \langle \nabla \ell(w),\, w' - w \rangle \end{aligned}$$

and add them to get

$$\langle \nabla \ell(w) - \nabla \ell(w'),\, w - w' \rangle \geq 0 \tag{10.3}$$

This is called the "monotonicity of the gradient" condition for convexity. In words, it says that the change in the gradient $\nabla \ell(w) - \nabla \ell(w')$ and the change in the weights $w - w'$ are aligned, i.e., their inner product is non-negative.

Second-order condition for convexity If ℓ is twice-differentiable and the domain is convex, then ℓ is convex iff

$$\nabla^2 \ell(w) \succeq 0 \tag{10.4}$$

for all w in the domain. The symbol $\succeq$ denotes positive semi-definiteness of the Hessian matrix $\nabla^2 \ell(w)$ whose entries are given by

$$(\nabla^2 \ell(w))_{ij} = \frac{\partial^2 \ell(w)}{\partial w_i \partial w_j}.$$

For strictly convex functions, the inequality in Eq. 10.4 is strict, i.e., the Hessian is positive definite. As an example using the second-order condition of convexity to show that a function is convex, note that the least squares objective

$$\ell(w) = \frac{1}{2} \|y - Xw\|_2^2$$

is convex because

$$\nabla^2 \ell(w) = X^\top X \succeq 0$$

which is positive semi-definite for any X.

Strongly convex functions A function is strongly convex if there exists an $m > 0$ such that

$$\ell(w) - \frac{m}{2} \|w\|_2^2 \text{ is convex.} \tag{10.5}$$

It is easy to see that strong convexity implies strict convexity. Since the function $\ell(w) - m/2 \|w\|_2$ is convex, it satisfies:

$$\ell(\lambda w + (1-\lambda) w') - \frac{m}{2} \|\lambda w + (1-\lambda) \|w'\|_2^2 \tag{10.6}$$

$$\leq \lambda \Big(\ell(w) - \frac{m}{2} \|w\|_2^2 \Big) + (1-\lambda) \Big(\ell(w') - \frac{m}{2} \|w'\|_2^2 \Big). \tag{10.7}$$

But

$$\lambda \frac{m}{2}\|w\|_2^2 + (1-\lambda)\frac{m}{2}\|w'\|_2^2 - \frac{m}{2}\|\lambda w + (1-\lambda)w'\|_2^2 > 0$$

for $\lambda \in (0,1)$ for all $w \neq w'$ because $\|w\|_2^2$ is strictly convex. This shows that if we have a strongly convex function ℓ it also satisfies

$$\ell(\lambda w + (1-\lambda)w') < \lambda \ell(w) + (1-\lambda)\ell(w').$$

In other words, we have

$$\text{strong convexity} \implies \text{strict convexity} \implies \text{convexity}.$$

We will see that strongly convex functions are easier to optimize for our algorithms. It will also always be much easier to prove a result, e.g., the number of iterations that we should run gradient descent for, for strongly convex functions. In your homework, you will show that the second-order condition for strongly convex functions reads as

$$\nabla^2 \ell(w) \quad \succeq \quad mI_{p\times p}.$$

We will use the following first-order condition for strongly convex functions often. A function is m-strongly convex if and only if

$$\ell(w') \quad \geq \quad \ell(w) + \langle \nabla \ell(w),\, w' - w\rangle + \frac{m}{2}\|w' - w\|_2^2 \tag{10.8}$$

for any w, w' in the domain. This is easy to show by observing that the function of $v = w' - w$

$$g(v) \quad \equiv \quad \ell(v+w) - \ell(w) - \langle \nabla \ell(w), v\rangle,$$

where w is fixed, is also m-strongly convex if ℓ is and therefore $g(v) - m/2\|v\|_2^2$ is convex.

The Polyak-Lojasiewicz (PL) inequality says that

$$\frac{1}{2m}\|\nabla \ell(w)\|_2^2 \quad \geq \quad \ell(w) - \ell(w^\star).$$

Functions that satisfy the PL inequality need not be convex (it simply says that the magnitude of the gradient at a point w should be small if $w \approx w^\star$) but it has been studied that such functions are also easy to optimize using first-order optimization methods. It has been empirically found that the PL inequality holds for many deep networks. Prove that a strongly convex function satisfies PL inequality, you can first prove Eq. 10.8 and minimize both sides of this inequality over w'.

10.2 Introduction to Gradient Descent

In this chapter, we will write $\ell(w)$ to denote the training objective, i.e., if we have a classifier $f(x; w)$ and a dataset $D = \{(x^i, y^i)\}_{i=1,\ldots,n}$ of n samples we will denote

$$\ell(w) := \frac{1}{n} \sum_{i=1}^{n} \ell(w; x^i, y^i).$$

The objective ℓ will always be a function of the entire dataset but we will keep the dependence implicit. Note that the number of samples n is usually quite large in deep learning, so the summation above has a large number of terms on the right-hand side. Gradient descent is a simple algorithm to minimize $\ell(w)$. Before we study its properties, it will help to refresh the following few facts.

10.2.1 Conditions for optimality

Local and global minima A point w is a local minimum of the function $\ell(w)$ for all w' in a neighborhood of w we have $\ell(w) \leq \ell(w')$. The point is a global minimum of the function ℓ if this condition is true for all w' in the domain, not just the ones in the neighborhood.

Local minima are global minima for convex functions This is easy to see using an argument by contradiction. If w is a local minimum that is not the global minimum, there exists a point w' in the domain such that $\ell(w') < \ell(w)$. The function is convex, so pick a point $v = \lambda w' + (1 - \lambda)w$ and see that

$$\ell(v) - \ell(w) \quad \leq \quad \lambda(\ell(w') - \ell(w))$$

using the definition of convexity (see Eq 10.1). Since w is only a local minimum, we can pick λ to be small enough that the left hand side is non-negative. This shows that $\ell(w') \geq \ell(w)$ but this means that w is a global minimum and we have a contradiction.

Global minimum is unique for strictly convex functions If a function is strictly convex on a convex domain the optimal solution (if it exists) must be unique. Indeed, if there were two solutions w, w' that were both minimizers we would have

$$\ell(w) = \ell(w') \leq \ell(w'') \ \ \forall w'' \tag{10.9}$$

We can now apply the definition of convexity to the point $v = (w + w')/2$ to get

$$\ell(v) < \frac{1}{2}\ell(w) + \frac{1}{2}\ell(w') = \ell(w),$$

which contradicts Eq. 10.9. The least-squares objective is strictly convex, so the solution is unique global minimizer of the objective.

First-order optimality condition If w is a local minimum of a continuously differentiable function ℓ, then it satisfies

$$\nabla \ell(w) = 0$$

If further ℓ is convex, then $\nabla \ell(w) = 0$ is a sufficient condition for global optimality from the above discussion.

10.2.2 Different types of convergence

Let us assume that we have a continuously differentiable convex function ℓ and let

$$w^\star = \arg\min_w \ell(w)$$

be the global minimizer of this function.
We would like to develop an iterative scheme that takes in the initialization of the weights w_0 and updates them to obtain a sequence

$$w^{(0)}, w^{(1)}, \ldots, w^{(t)}, \ldots$$

Along this sequence we are interested in understanding the

1. convergence of the function value $\ell(w^{(t)})$ to the minimal value $\ell(w^\star)$, and
2. convergence of the iterates $\|w^{(t)} - w^\star\|$.

Descent direction We are going to perform a sequence of updates given by

$$w^{(t+1)} = w^{(t)} + \rho d^{(t)}$$

where $d^{(t)}$ is called the descent direction and the scalar parameter $\rho > 0$ is called the step-size and determines how far we travel using this descent direction. Any direction such that

$$\langle \nabla \ell(w^{(t)}), d^{(t)} \rangle < 0$$

is a good descent direction because this leads to a reduction in the value of the function $\ell(w^{(t+1)})$ after the weight update locally. There are numerous ways to pick a good descent direction. Among the simplest ones is gradient descent which descends along the direction of the negative gradient and thereby performs the following set of updates

$$w^{(t+1)} = w^{(t)} - \rho \nabla \ell(w^{(t)}). \tag{9.12}$$

given an initial value $w^{(0)}$. The step-size (also called the learning rate) is chosen by the user. The step-size need not always be fixed, for instance you can chose

it to be a function of the number of weight updates t. A good step-size is one that does not overshoot the minimum $w^\star$.

> Draw a picture of overshooting using a large step-size.

For instance, after having chosen a particular descent direction $d^{(t)}$ we can compute the best step-size to use at time t by solving

$$\rho^{(t)} = \arg\min_{\rho \geq 0} \ell(w^{(t)} + \rho d^{(t)}).$$

This is known as line-search in the optimization literature. You may have seen Newton's method

$$w^{(t+1)} = w^{(t)} - (\nabla^2 \ell(w^{(t)}))^{-1} \nabla \ell(w^{(t)})$$

which does not have a user-tuned step-size and further modifies the descent direction to be the product of the inverse Hessian with the gradient.

10.3 Convergence rate for gradient descent

We will next understand how quickly gradient descent converges to the global minimum. There are two concrete goals of this analysis

1. to be able to pick the step-size to avoid overshooting without doing line-search, and
2. characterize how many iterations of gradient descent to run until we are guaranteed to be within some distance of the global minimum.

10.3.1 Some assumptions

Before we begin, we will make a few simplifying assumptions on the function $\ell(w)$. These are quite typical in optimization and ensure that we are not dealing with functions that are arbitrarily di"cult to optimize.

1. **Lipschitz continuity/bounded gradients** We will assume that ℓ is uniformly Lipschitz continuous over the entire domain, i.e.,

   $$|\ell(w) - \ell(w')| \leq B\|w - w'\|_2$$

   for some $B > 0$. You might also see this condition written as

   $$\|\nabla \ell(w)\| \leq B$$

   for differentiable functions; show that the second one is an implication of the first.

2. **Smoothness** We will always consider "smooth" functions with gradients that are L-Lipschitz, i.e.,

$$\|\nabla \ell(w) - \nabla \ell(w')\|_2 \leq L\|w - w'\|_2$$

If ℓ is twice-differentiable, this is equivalent to assuming

$$\nabla^2 \ell(w) \quad \preceq \quad L\, I_{p\times p}.$$

From the Cauchy-Schwarz inequality which states that

$$\langle u, v\rangle \leq \|u\|\|v\|$$

for two vectors u, v, we have the following implication of smoothness:

$$\langle \nabla \ell(w) - \nabla \ell(w'),\, w - w'\rangle \leq L\|w - w'\|_2^2. \tag{10.10}$$

10.3.2 GD for convex functions

We begin with the so-called Descent Lemma.

Lemma 10.3.1 (Descent Lemma)**.** For an L-smooth function, we have

$$\ell(w') \leq \ell(w) + \langle \nabla \ell(w),\, w' - w\rangle + \frac{L}{2}\|w' - w\|_2^2$$

for any two w, w' in the domain.

Proof. First, you should compare this with the first-order characterization of convexity

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w),\, w' - w\rangle.$$

The two conditions can be used to sandwich the value of $\ell(w^{(t+1)})$ given the value of $\ell(w^{(t)})$ in gradient descent with room for a quadratic term $\frac{L}{2}\|w' - w\|_2^2$. This marshals the intuition as to what L-smooth really means; a large value of L means that the function ℓ has a large curvature.
Let $v = w + \lambda(w' - w)$ and use Taylor's theorem to see that

$$\ell(w') = \ell(w) + \int_0^1 \langle \nabla \ell(v),\, w' - w\rangle\, d\lambda$$

Further explanation. Now, let's parameterize the line segment. Let $w, w' \in \mathbb{R}^d$ and define a path between them by

$$v(\lambda) = w + \lambda(w' - w), \quad \lambda \in [0, 1].$$

Note that $v(0) = w$ and $v(1) = w'$. Define the scalar-valued function

$$\phi(\lambda) := \ell(v(\lambda)) = \ell\big(w + \lambda(w' - w)\big).$$

This reduces the problem to a one-dimensional setting.
Next, let's take derivative along the path. By the chain rule,

$$\phi'(\lambda) = \left\langle \nabla \ell(v(\lambda)), \frac{dv(\lambda)}{d\lambda} \right\rangle = \langle \nabla \ell(v(\lambda)),\, w' - w \rangle.$$

Next, using the Fundamental Theorem of Calculus,

$$\phi(1) - \phi(0) = \int_0^1 \phi'(\lambda)\, d\lambda.$$

Substituting back, we obtain the exact identity

$$\ell(w') = \ell(w) + \int_0^1 \langle \nabla \ell(v(\lambda)),\, w' - w \rangle\, d\lambda.$$

Subtract $\langle \nabla \ell(w),\, w' - w \rangle$ from both sides to get

$$\ell(w') - \ell(w) - \langle \nabla \ell(w),\, w' - w \rangle = \int_0^1 \langle \nabla \ell(v) - \nabla \ell(w),\, w' - w \rangle\, d\lambda.$$

Observe that

$$\begin{aligned}
|\ell(w') - \ell(w) - \langle \nabla \ell(w),\, w' - w \rangle| &= \left| \int_0^1 \langle \nabla \ell(v) - \nabla \ell(w),\, w' - w \rangle\, d\lambda \right| \\
&\leq \int_0^1 |\langle \nabla \ell(v) - \nabla \ell(w),\, w' - w \rangle|\, d\lambda \\
&\leq \int_0^1 \|\nabla \ell(v) - \nabla \ell(w)\|\, \|w' - w\|\, d\lambda \\
&\leq L \int_0^1 \lambda \|w' - w\|_2^2\, d\lambda \\
&= \frac{L}{2} \|w' - w\|_2^2.
\end{aligned}$$

This completes the proof after removing the absolute value on the left-hand side. □

We can use the Descent Lemma twice on two points to w, w' to get Eq. 10.10. Another direct consequence of the Descent Lemma is the following corollary

that relates the value $\ell(w)$ at any point w in the domain to that of the global minimum.

Corollary 10.3.1. For L-smooth convex function ℓ, if $w^\star$ is the global minimizer, then

$$\frac{1}{2L}\|\nabla\ell(w)\|_2^2 \leq \ell(w) - \ell(w^\star) \leq \frac{L}{2}\|w - w^\star\|_2^2$$

Proof. Since $\nabla\ell(w^\star) = 0$, the right-hand side follows directly from the Descent Lemma. To get the left-hand size, let us optimize the upper bound in the Descent Lemma using $w' = w + \lambda v$ with $\|v\| = 1$ as follows

$$\begin{aligned}
\ell(w^\star) &= \inf_{w'} \ell(w') \\
&\leq \inf_{w'} \left[\ell(w) + \langle\nabla\ell(w),\, w' - w\rangle + \frac{L}{2}\|w' - w\|_2^2\right] \\
&= \inf_{\|v\|=1} \inf_{\lambda} \left[\ell(w) + \lambda\langle\nabla\ell(w), v\rangle + \frac{L}{2}\lambda^2\right] \\
&= \inf_{\|v\|=1} \left[\ell(w) - \frac{1}{2L}(\langle\nabla\ell(w), v\rangle)^2\right] \\
&= \ell(w) - \frac{1}{2L}\|\nabla\ell(w)\|_2^2.
\end{aligned}$$

□

In other words, the gap between the function values $\ell(w) - \ell(w^\star)$ is upper-bounded by the gap to the minimizer $\frac{L}{2}\|w - w^\star\|_2^2$ and lower-bounded by the norm of the gradient $\frac{1}{2L}\|\nabla\ell(w)\|_2^2$.

Co-coercivity of the gradient The gradient being L-Lipschitz is equivalent to co-coercivity of the gradient with parameter $1/L$

$$\langle\nabla\ell(w) - \nabla\ell(w'),\, w - w'\rangle \geq \frac{1}{L}\|\nabla\ell(w) - \nabla\ell(w')\|_2^2 \tag{10.11}$$

> The condition in Eq. 10.11 is called co-coercivity because there is a related condition called coercivity for m-strongly convex functions
>
> $$\langle\nabla\ell(w) - \nabla\ell(w'),\, w - w'\rangle \geq m\|w - w'\|_2^2,$$
>
> for all w, w'. Try to prove it. Note that this condition boils down to simple monotonicity of the gradient for $m = 0$ (which is just a convex function). This is why it is also called strong monotonicity of $\nabla\ell$.

We can see that co-coercivity implies Lipschitz continuity of the gradients $\nabla\ell(w)$ using Eqs. 10.10 and 10.11 .

Note that Lipschitz-continuity of the gradient implies the Descent Lemma (Lemma 10.3.1). Now define two functions

$$g(u) = \ell(u) - \langle \nabla \ell(w), u \rangle$$

$$h(u) = \ell(u) - \langle \nabla \ell(w'), u \rangle.$$

Both of these have L-Lipschitz gradients. Note that $u = w$ minimizes $g(u)$ (the minimum is zero). Observe that

$$\begin{aligned} \ell(w') - \ell(w) - \langle \nabla \ell(w), w' - w \rangle &= g(w') - g(w) && (10.12) \\ &\geq \frac{1}{2L} \|\nabla g(w')\|_2^2 \text{ from corollary 10.3.1} && (10.13) \\ &= \frac{1}{2L} \|\nabla \ell(w') - \nabla \ell(w)\|_2^2. && (10.14) \end{aligned}$$

Apply the same again to h to get

$$\ell(w) - \ell(w') - \langle \nabla \ell(w'), w - w' \rangle \geq \frac{1}{2L} \|\nabla \ell(w') - \nabla \ell(w)\|_2^2$$

and add the two inequalities, we arrive at Equation 10.11.

We can now get our first result on how gradient descent makes monotonic progress towards the solution.

Lemma 10.3.2 (Monotonic progress for gradient descent)**.** For gradient descent $w^{(t+1)} = w^{(t)} - \rho \nabla \ell(w^{(t)})$, if we pick the step-size

$$\rho \leq \frac{1}{L} \tag{10.15}$$

We have

$$\ell(w^{(t+1)}) \leq \ell(w^{(t)}) - \frac{\rho}{2} \|\nabla \ell(w^{(t)})\|_2^2 \quad \forall t. \tag{10.16}$$

Further,

$$\ell(w^{(t+1)}) - \ell(w^\star) \leq \frac{1}{2\rho} \left(\|w^{(t)} - w^\star\|_2^2 - \|w^{(t+1)} - w^\star\|_2^2 \right) \tag{10.17}$$

which implies

$$\|w^{(t+1)} - w^\star\|_2^2 \leq \|w^{(t)} - w^\star\|_2^2 \tag{10.18}$$

Proof. Substitute $\rho \leq 1/L$ in the Descent Lemma and simplify to get Eq. 10.16. The second result is obtained by

$$0 \leq \ell(w^{(t+1)}) - \ell(w^\star) \leq \ell(w^{(t)}) - \ell(w^\star) - \frac{\rho}{2}\|\nabla \ell(w^{(t)})\|_2^2 \tag{10.19}$$

$$\text{Set } w = w^{(t)} \text{ and } w' = w^* \text{ in 10.14} \tag{10.20}$$

$$\leq \langle \nabla \ell(w^{(t)}),\, w^{(t)} - w^\star \rangle - \frac{\rho}{2}\|\nabla \ell(w^{(t)})\|_2^2 \tag{10.21}$$

Now, expanding the squared distance, we get the following :

$$\|w^{(t+1)} - w^*\|^2 = \|w^{(t)} - w^* - \rho \nabla \ell(w^{(t)})\|^2 \tag{10.22}$$

$$= \|w^{(t)} - w^*\|^2 - 2\rho \langle \nabla \ell(w^{(t)}, w^{(t)} - w^* \rangle + \rho^2 \|\nabla \ell(w^{(t)}\|_2^2 \tag{10.23}$$

$$\langle \nabla \ell(w^{(t)}, w^{(t)} - w^* \rangle = \frac{1}{2\rho}\left(\|w^{(t)} - w^*\|^2 - \|w^{(t+1)} - w^*\|^2 \right) + \frac{\rho}{2}\|\nabla \ell(w^{(t)}\|_2^2 \tag{10.24}$$

Substituting this back in Equation 10.21, we get :

$$0 \leq \ell(w^{(t+1)}) - \ell(w^\star) \leq \frac{1}{2\rho}\left(\|w^{(t)} - w^*\|^2 - \|w^{(t+1)} - w^*\|^2 \right) \tag{10.25}$$

Observe that since the left-hand side is positive, the claim in Eq. 10.18 is true. □

We have therefore shown that if the step-size is not too large (the smoothness parameter of the function determines how large the step-size can be) then gradient descent always improves the value of the function with each iteration Eq. 10.17. It also improves the distance of the weights to the global minimum at each iteration Eq. 10.18.

Lemma 10.3.3 (Convergence rate for gradient descent, convex function)**.** For gradient descent $w^{(t+1)} = w^{(t)} - \rho \nabla \ell(w^{(t)})$ with step-size $\rho < 1/L$, we have

$$\ell(w^{(t+1)}) - \ell(w^\star) \leq \frac{1}{2t\rho}\|w^{(0)} - w^\star\|_2^2.$$

Proof. We sum up the expression in Eq. 10.17 for all times t to get

$$\sum_{s=1}^{t} \left(\ell(w^{(s)}) - \ell(w^\star) \right) \leq \frac{1}{2\rho} \sum_{s=1}^{t} \left(\|w^{(s-1)} - w^\star\|_2^2 - \|w^{(s)} - w^\star\|_2^2 \right) \tag{10.26}$$

$$= \frac{1}{2\rho}\left(\|w^{(0)} - w^\star\|_2^2 - \|w^{(t)} - w^\star\|_2^2 \right) \tag{10.27}$$

$$\leq \frac{1}{2\rho}\|w^{(0)} - w^*\|^2 \tag{10.28}$$

We know from Eq. 10.18 that $\ell(w^{(t)})$ is non-increasing, so we can write

$$\ell(w^{(t)}) - \ell(w^\star) \leq \frac{1}{t}\sum_{s=1}^{t}\Big(\ell(w^{(s)}) - \ell(w^\star)\Big) \leq \frac{1}{2t\rho}\|w^{(0)} - w^\star\|_2^2$$

If we want to find weights with

$$\ell(w^{(t)}) - \ell(w^\star) \leq \epsilon$$

for a convex function, we need to run gradient descent for at least

$$t = O(1/\epsilon)$$

iterations. This is an important result to remember.

10.3.3 Limits on convergence rate of first-order methods

It is a powerful and deep result that we cannot do better than a linear convergence rate for optimization methods that only use the gradient of the function $\ell(w)$. More precisely, for any first-order method, i.e., any method where the iterate at step t given by $w^{(t)}$ is chosen to be

$$w^{(t)} \in w_0 + \text{span}\{\nabla\ell(w_0), \ldots, \nabla\ell(w_t)\},$$

we have the following theorem by Yurii Nesterov.

Theorem 10.3.1 (Nesterov's lower bound)**.** If $w \in \mathbb{R}^p$, for any $t \leq (p-1)/2$ and every initialization of weights w_0 there exist functions $\ell(w)$ that are convex, differentiable, L-smooth with finite optimal value $\ell(w^\star)$ such that any first-order method has

$$\ell(w^{(t)}) - \ell(w^\star) \leq \frac{3}{32}\frac{L\|w_0 - w^\star\|_2^2}{(t+1)^2}.$$

Let us read the statement of the theorem carefully. It states that fix a time t and initial condition w_0, we can find a convex function $\ell(w)$ such that it takes any first order method at least $O(1/\sqrt{\epsilon})$ to reach an ϵ-neighborhood of the optimal value $\ell(w^\star)$. The implication of this theorem is as follows. The convergence rate $O(1/\epsilon)$ we obtained for convex functions is not the best rate we can get. Nesterov's lower bound suggests that there should be gradient-based algorithms that only require $O(1/\sqrt{\epsilon})$ iterations. Such methods will be the topic of the next section.

10.4 Accelerated Gradient Descent

In the previous section we saw two results that characterize how many iterations gradient descent requires to reach within an ϵ-neighborhood of the global

optimum for convex functions. If the function $\ell(w)$ is convex, GD with a step-size at most $1/L$ requires $O(1/\epsilon)$ iterations. We will study two algorithms in this section which accelerates the progress of gradient descent.

10.4.1 Polyak's Heavy Ball method

The most natural place to begin is to imagine gradient descent as a kinematic equation. Let $w^{(t)}$ be the iterate of GD at time t, let us associate to it an auxiliary variable called the "velocity"

$$v^{(t)} := w^{(t+1)} - w^{(t)}. \tag{10.29}$$

Gradient descent can then be written as

$$v^{(t)} = -\eta \nabla \ell(w^{(t)}), \tag{10.30}$$

which allows us to think of the term $\nabla \ell(w^{(t)})$ as some kind of force that acts on a particle to update its position from $w^{(t)}$ to $w^{(t+1)}$. This particle has no inertia, so we will say that the applied force directly affects its position. If the magnitude of the gradient is small in a certain direction, the velocity is also small in that direction.

We now give our particle some inertia. Instead of the force directly affecting the position we will write down Newton's second law of motion ($F = ma$) for a particle with unit mass $m = 1$ and time discretization η (or equivalently, with a time discretization of 1 and a mass of η^{-1}) as

$$-\nabla \ell(w^{(t)}) =: \frac{v^{(t+1)} - v^{(t)}}{\eta} = \frac{1}{\eta}\big(w^{(t+1)} - 2w^{(t)} + w(t-1)\big) \tag{10.31}$$

$$\implies \quad w^{(t+1)} = w^{(t)} - \eta \nabla \ell(w^{(t)}) + \big(w^{(t)} - w^{(t-1)}\big). \tag{10.32}$$

Notice the third term on the right-hand side above, it is the gap between the current weights $w^{(t)}$ and the previous weights $w^{(t-1)}$, if we have

$$\langle w^{(t)} - w^{(t-1)},\ \nabla \ell(w^{(t)}) \rangle < 0,$$

i.e., the change from the previous time-step is along the descent direction, then the weights $w^{(t+1)}$ get an extra boost. If instead, the change from the previous direction is not along the gradient descent direction, then the third term reduces the magnitude of the gradient. The third term is effectively the inertia of gradient updates. This method is therefore called Polyak's Heavy Ball method.

We give ourselves some more control over how inertia enters the update equation using a hyper-parameter η (which is akin to mass)

$$w^{(t+1)} = w^{(t)} - \rho \nabla \ell(w^{(t)}) + \eta \left(w^{(t)} - w^{(t-1)}\right). \tag{10.33}$$

If $\eta = 0$, we do not use any inertia and Polyak's method boils down to gradient descent. Typically, we choose $\eta \in (0, 1)$. This inertia is called momentum in the optimization literature and ρ is called the momentum coefficient.

Polyak's method is simple yet very powerful. In the previous chapter, we showed a lower-bound of Nesterov which indicates that first-order optimization algorithm (that only depends on the gradient of the objective) cannot be faster than $O(1/\sqrt{\epsilon})$. It turns out that Polyak's method converges at this rate, i.e., if we want

$$\|w^{(t)} - w^\star\| \leq \ell$$

we need to run Polyak's Heavy Ball method for $O(1/\sqrt{\epsilon})$ iterations for convex functions. These improvements are also quite a lot, we need quadratically fewer iterations than gradient descent in Polyak's method and the only incremental cost of doing so is that we have to maintain a copy of the weights $w^{(t+1)}$ while implementing the updates in Eq. 10.33.

An alternative way to write Polyak's updates. We can rewrite the updates in Eq. 10.33 using a dummy variable $u(t)$ as

$$u^{(t)} = (1 + \eta) w^{(t)} - \eta w^{(t-1)} \tag{10.34}$$

$$w^{(t+1)} = u^{(t)} - \rho \nabla \ell(w^{(t)}) \tag{10.35}$$

This is how these updates are implemented in PyTorch. This is convenient: effectively, the code needs to maintain only the difference $u(t) = (1 + \eta) w^{(t)} - \eta w^{(t-1)}$ in a buffer $u(t)$ and subtract the gradient $\nabla \ell(w^{(t)})$ from this buffer to result in the new updates. GD can be implemented with a simple change by setting $u(t) := w^{(t)}$ which corresponds to $\eta = 0$. The dummy variable is initialized to $u_0 = w_0$.

A yet another way to write Polyak's updates. We can also rewrite the updates in Eq.10.35 as

$$u^{(t+1)} = \eta\, u^{(t)} - \nabla \ell(w^{(t)}) \tag{10.36}$$

$$w^{(t+1)} = w^{(t)} + \rho\, u^{(t+1)}. \tag{10.37}$$

This set of updates brings out idea of momentum more clearly. The variable $u^{(t)}$ in this case is exactly the velocity $v^{(t)}$ that we have seen above except that it is updated slightly different than our expression ($F = ma$) in the first equation. The first term

$$u^{(t+1)} = \eta\, u^{(t)} - \nabla \ell(w^{(t)})$$

reduces the velocity $u(t)$ by a factor η before adding the gradient to it. To derive the above, observe that $\rho\, u^{(t)} = (w^{(t)} - w^{(t-1)})$, and plug this into Equation 10.33. Now, define $u^{(t+1)}$ as in 10.36 to establish 10.37.

10.4.2 Polyak's method can fail to converge

The caveat with relying on the inertia of the particle to make progress is that near the global minimum, when the iterates overshoot the global minimum, the inertia is often very different from the gradient. Polyak's method can become unstable and can result in oscillations under such conditions, e.g.,

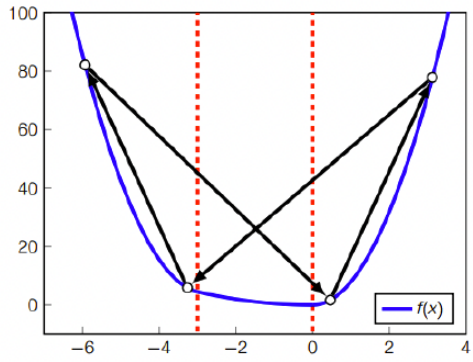


Figure 10.1

However it is a very simple method to accelerate gradient descent and works great in practice.

For the theoretical reason of faster convergence of the heavy ball method I would encourage you to check out the proof here.

Chapter 11

Clustering

In the problem of clustering, we are given a dataset comprised only of input features without labels. We wish to assign to each data point a discrete label indicating which "cluster" it belongs to, in such a way that the resulting cluster assignment "fits" the data. We are given flexibility to choose our notion of goodness of fit for cluster assignments.
Clustering is an example of unsupervised learning, where we are not given labels and desire to infer something about the underlying structure of the data. Another example of unsupervised learning is dimensionality reduction, where we desire to learn important features from the data. Clustering is most often used in exploratory data visualization, as it allows us to see the different groups of similar data points within the data. Combined with domain knowledge, these clusters can have a physical interpretation - for example, different clusters can represent different species of plant in the biological setting, or types of consumers in a business setting. If desired, these clusters can be used as pre-processing to make the data more compact. Clustering is also used for outlier detection, as in Figure 11.2: data points that do not seem to belong in their assigned cluster may be flagged as outliers.
In order to create an algorithm for clustering, we first must determine what makes a good clustering assignment. Here are some possible desired proper-

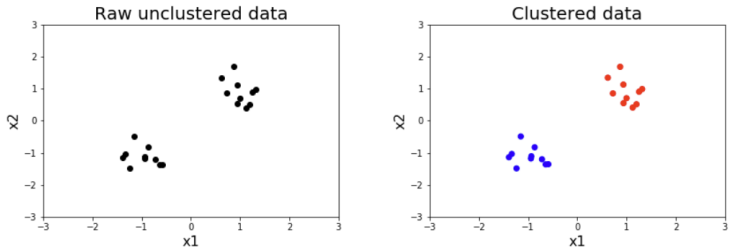


Figure 11.1: Left: unclustered raw data; Right: clustered data

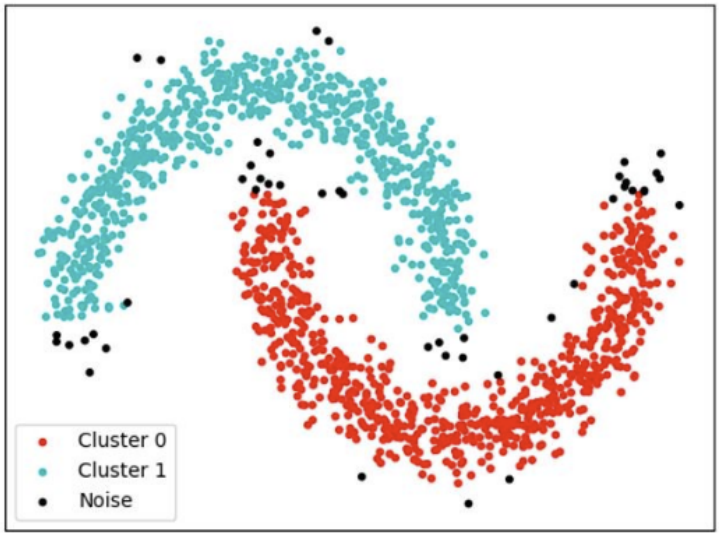


Figure 11.2: A nonspherical clustering assignment. Possible outliers are shown in black. Click here for details

ties:

1. High intra-cluster similarity - points within a given cluster are very similar.

2. Low inter-cluster similarity - points in different clusters are not very similar.

Of course, this depends on our notion of similarity. For now, we will say that points in $\mathbb{R}^d$ are similar if their L_2 distance is small, and dissimilar otherwise. A generalization of this notion is provided in the appendix.

11.1 K-means Clustering

Let X denote the set of N data points $x_i \in \mathbb{R}^d$. A cluster assignment is a partition $C_1, \ldots, C_K \subseteq X$ such that the sets C_k are disjoint and $X = C_1 \cup \cdots \cup C_K$. A data point $x \in X$ is said to belong to cluster k if it is in C_k.
One approach to the clustering problem is to represent each cluster C_k by a single point $c_k \in \mathbb{R}^d$ in the input space - this is called the centroid approach. K-means is an example of centroid-based clustering where we choose centroids and a cluster assignment such that the total distance of each point to its assigned centroid is minimized. In this regard, K-means optimizes for high intra-cluster similarity, but the clusters do not necessarily need to be far apart, so we may also have high inter-cluster similarity.

Formally, K-means solves the following problem:

$$\arg\min_{\{C_k\}_{k=1}^K, \{c_k\}_{k=1}^K : X = C_1 \cup \cdots \cup C_K} \sum_{k=1}^{K} \sum_{x \in C_k} \|x - c_k\|^2 \tag{11.1}$$

It has been shown that this problem is NP hard, so solving it exactly is intractable. However, we can come up with a simple algorithm to compute a candidate solution. If we knew the cluster assignment $C_1, \ldots, C_K$, then we would only need to determine the centroid locations. Since the choice of centroid location c_i does not affect the distances of points in C_j to c_j for $i \neq j$, we can consider each cluster separately and choose the centroid that minimizes the sum of squared distances to points in that cluster. The centroid we compute, $\hat{c}_k$, is

$$\hat{c}_k = \arg\min_{c_k} \sum_{x \in C_k} \|x - c_k\|^2$$

But this is simply the mean of the data in C_k, that is,

$$\hat{c}_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$

Similarly, if we knew the centroids c_k, in order to choose the cluster assignment $C_1, \ldots, C_K$ that minimizes the sum of squared distances to the centroids, we simply assign each data point x to the cluster represented by its closest centroid, that is, we assign x to

$$\arg\min_k \|x - c_k\|^2$$

Now we can perform alternating minimization - on each iteration of our algorithm, we update the clusters using the current centroids, and then update the centroids using the new clusters. This algorithm is sometimes called Lloyd's Algorithm.

Algorithm 1: K-means Algorithm

Initialize $c_k, k = 1, \ldots, K$
while K-means objective has not converged do

- Update partition $C_1 \cup \cdots \cup C_K$ given the c_k by assigning each $x \in X$ to the cluster represented by its nearest centroid
- Update centroids c_k given $C_1 \cup \cdots \cup C_K$ as

$$c_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$

This algorithm will always converge to some value. To show this, note the following facts:

1. There are only finitely many (say, M) possible partition/centroid pairs that can be produced by the algorithm.

2. Each update of the cluster assignment and centroids does not increase the value of the objective.

If the value of the objective has not converged after M iterations, then we have cycled through all the possible partition/centroid pairs attainable by the algorithm. On the next iteration, we would obtain a partition and centroid assignment that we have already seen. In practice, it is common to run the K-means algorithm multiple times with different initialization points, and the cluster corresponding to the minimum objective value is chosen. There are also ways to choose a smarter initialization than a random seed, which can improve the quality of the local optimum found by the algorithm.[1] It should be emphasized that no efficient algorithm for solving the K-means optimization is guaranteed to give a good cluster assignment, as the problem is NP hard and there are local optima. Choosing the number of clusters k is similar to choosing the number of principal components for PCA - we can compute the value of the objective for multiple values of k and find the "elbow" in the curve.

11.2 Practical Use of K-means

K-means is one of the most well-known and widely used clustering algorithms in practice. Despite the fact that the underlying optimization problem is NP-hard and the algorithm only guarantees convergence to a local optimum, K-means remains popular due to its conceptual simplicity, ease of implementation, scalability to large datasets, and strong empirical performance in many real-world applications. It is commonly used in exploratory data analysis, data compression, vector quantization, image segmentation, and as a preprocessing step in larger machine learning pipelines.
Because of its practical importance, K-means has been implemented in essentially all major scientific computing and machine learning libraries. In Python, commonly used implementations include:

- **scikit-learn**: The `sklearn.cluster.KMeans` class provides a highly optimized implementation with support for multiple random initializations, K-means++ initialization, and efficient convergence checks. SciKit
- **SciPy**: The `scipy.cluster.vq.kmeans` and `kmeans2` functions provide classical implementations closely aligned with Lloyd's algorithm. SciPy
- **PyTorch (community implementations)**: While PyTorch does not include K-means in its core library, a number of GPU-accelerated implementations are available and commonly used when clustering large, high-dimensional datasets. Pytorch

[1] For example, K-means++.

- **FAISS**: Facebook AI Similarity Search provides highly optimized CPU and GPU implementations of K-means designed for very large-scale clustering problems. FAISS

These implementations reflect the continued relevance of K-means as a practical tool, even as more expressive probabilistic and nonparametric clustering models have been developed.

Bibliography

Aizerman, M. A. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837.

Amari, S. (1967). A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, EC-16(3):299–307.

Balasubramanian, V. (2015). Heterogeneity and efficiency in the brain. *Proceedings of the IEEE*, 103(8):1346–1358.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1(2):119–130.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

Minsky, M. and Papert, S. A. (2017). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA.

Pickering, A. (2010). *The Cybernetic Brain: Sketches of Another Future*. University of Chicago Press, Chicago.

Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, pages 1177–1184.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, Institute for Cognitive Science, University of California, San Diego, La Jolla, CA.

Salakhutdinov, R. and Larochelle, H. (2010). Efficient learning of deep boltzmann machines. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, pages 693–700.

Schölkopf, B. and Smola, A. J. (2018). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Turing, A. M. (2009). Computing machinery and intelligence. In Epstein, R., Roberts, G., and Beber, G., editors, *Parsing the Turing Test*, pages 23–65. Springer, Dordrecht.

Vapnik, V. N. (2013). *The Nature of Statistical Learning Theory*. Statistics for Engineering and Information Science. Springer, 2 edition.

Wiener, N. (1965). *Cybernetics: Or Control and Communication in the Animal and the Machine*, volume 25. MIT Press, Cambridge, MA.