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

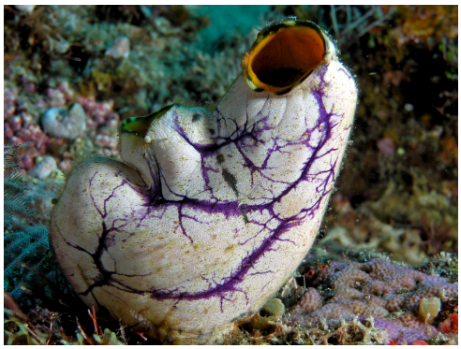


Figure 1.1: A Tunicate on the ocean floor

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.

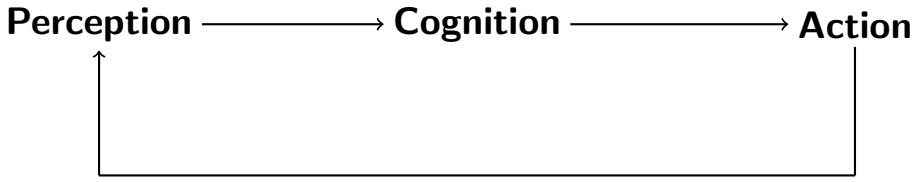
Is a program like AlphaGo intelligent? There is a very nice movie on Netflix on the development of AlphaGo and here's an excerpt from the movie. The commentator in this video is wondering how Lee Se-dol, who was one of the most accomplished Go players in the world then, might defeat this very powerful 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.

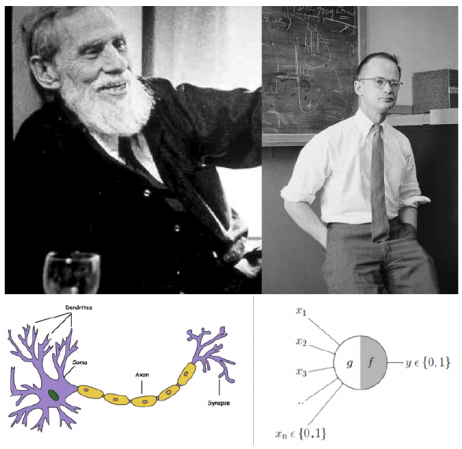
Cognition also involves handling situations when the current data does not match past data, etc. To give you an example, arithmetic problems you solved in elementary school are akin to learning. Whereas figuring out that taking a standard deduction when you file your income tax versus itemized deduction 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.

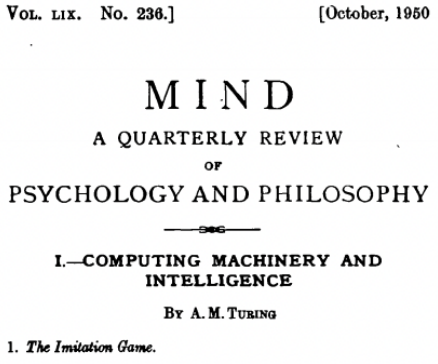


A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

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.

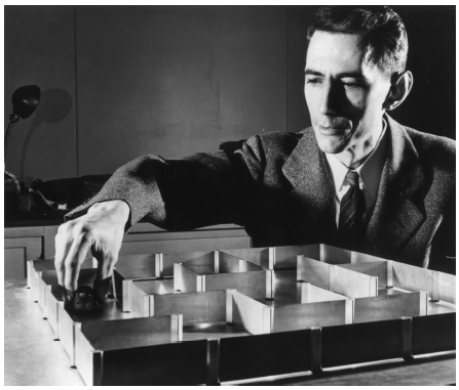


Figure 1.2: Claude Shannon studied information theory. This is a picture of a maze solving mouse that he made around 1950, among the world's first examples of machine learning; read more here

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)

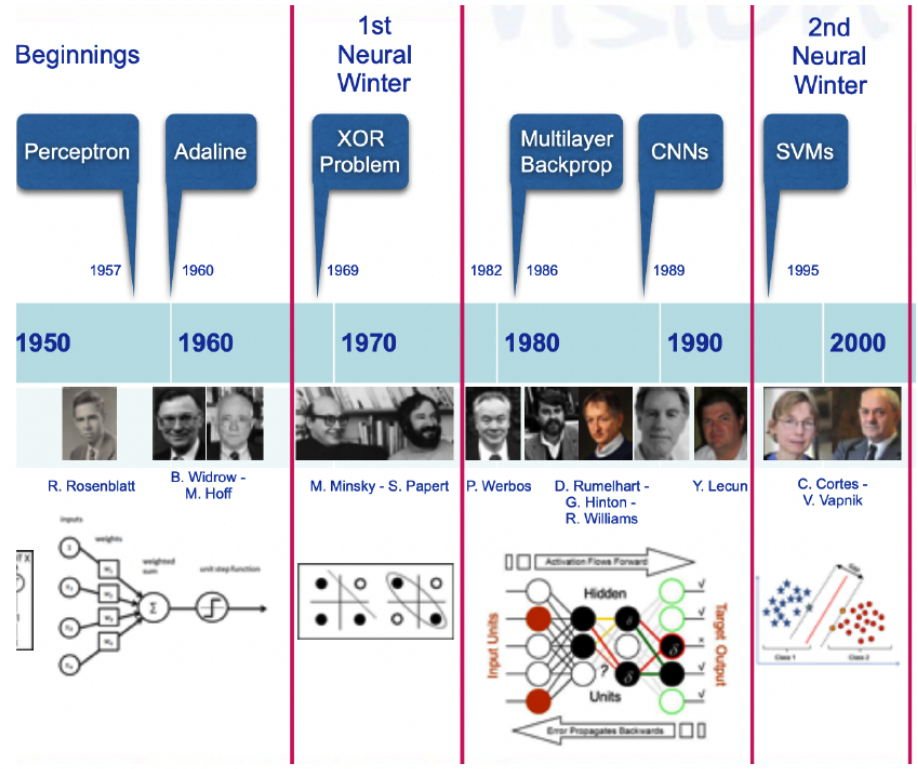
The early period created interest in intelligence and developed some basic ideas. The first major progress of what one would call the second era was made by Frank Rosenblatt in 1957 at Cornell University. Rosenblatt's model called the perceptron is a model with a single binary neuron. It was a machine 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 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****.*

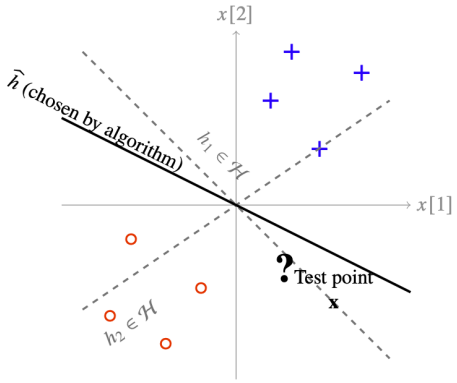


Figure 2.1: Visualizing the learning setup. The algorithm sees the blue '+' and red 'o' training points and selects the solid black line/hypothesis ($\hat{h}$) from the set of all possible lines/hypotheses (a few are shown as dashed). The goal is to perform well (see Sec. 2.2) on a new test point (the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

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**:

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

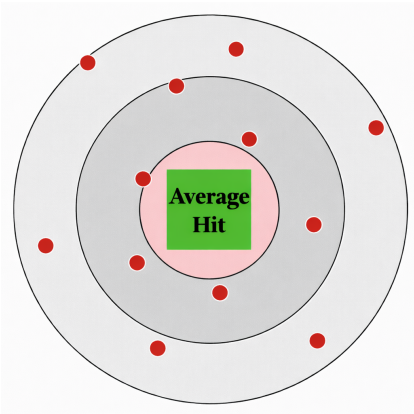
What is the probability over in the above expression? Remember, we took h to be fixed and also $R(h)$ already involves an expectation, so is itself a deterministic quantity (convince yourself!). The only random variable is $\hat{R}_n(h)$ which depends on the training set $\mathcal{D}$. The training set $\mathcal{D}$ 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

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

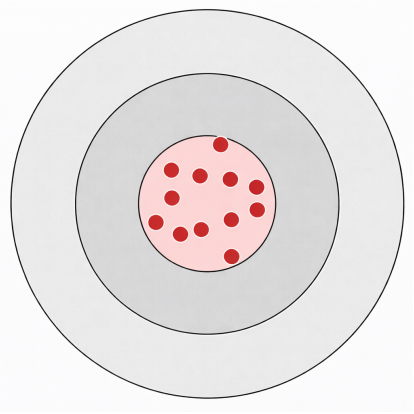
Then,

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

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 $n \geq \frac{2}{\epsilon^2}\log(m/\delta)$

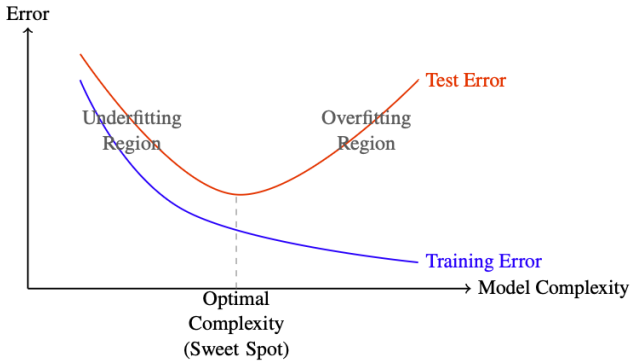


Figure 2.3: A typical U-shaped curve showing the relationship between model complexity and error. As complexity increases, training error decreases, but test error initially falls and then rises.

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

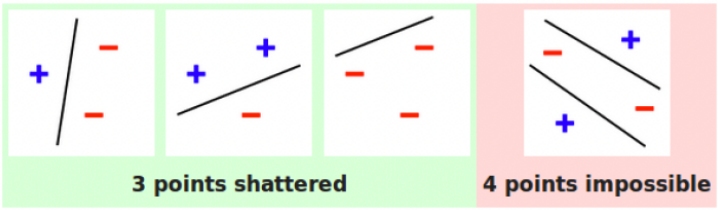


Figure 2.4: d=2: See that for the lower bound, we found some configuration of the 3 points, such that a linear threshold function always separates the points consistently with the labels; for any possible labeling. 3 such labellings are shown, convince yourselves that it can be done for all 8 cases. 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 4.1 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.

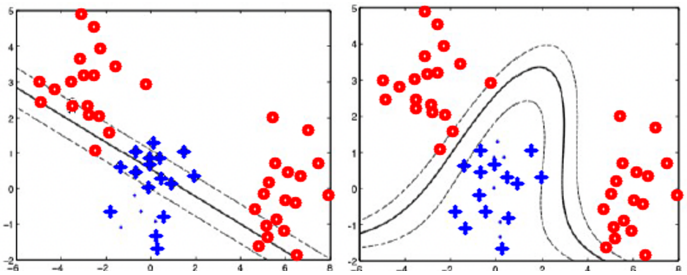


Figure 4.1: Feature vectors can be used to induce non-linear features in the classification space.

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

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

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.

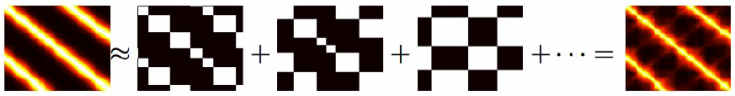


Figure 4.2

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. 4.11 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. 4.11 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. 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.

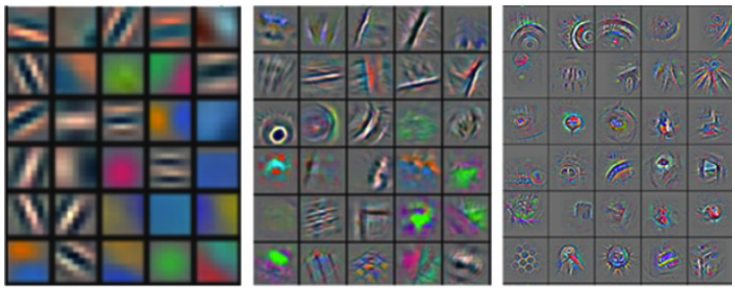


Figure 5.1

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 (Section 1.2): 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 param-

eters. The set

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

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 $\overline{x}$.

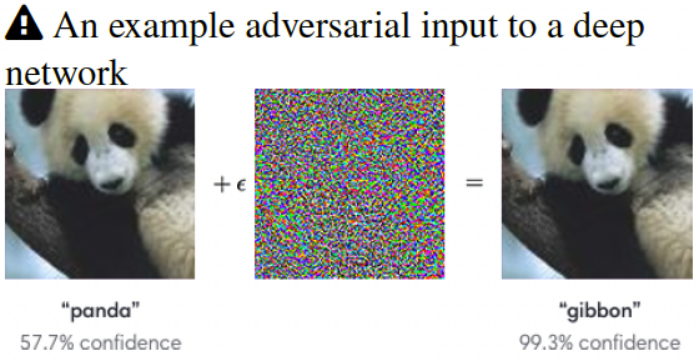


Figure 5.2

5.2.2 Implementation of backpropagation

Consider our neural network classifier given by

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

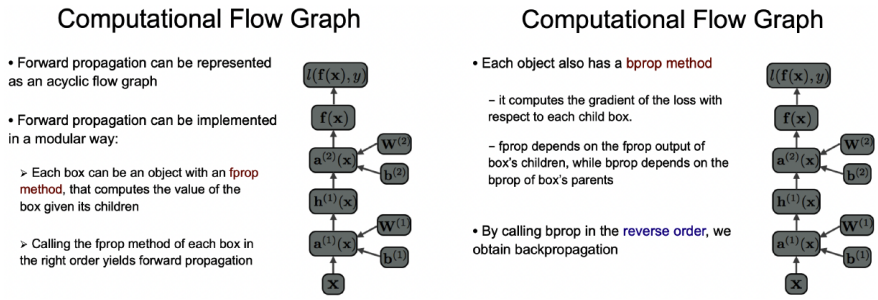


Figure 5.3: Schematic of forward and backward computations in backpropagation

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 nonlinearities. 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

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 6.2: 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-

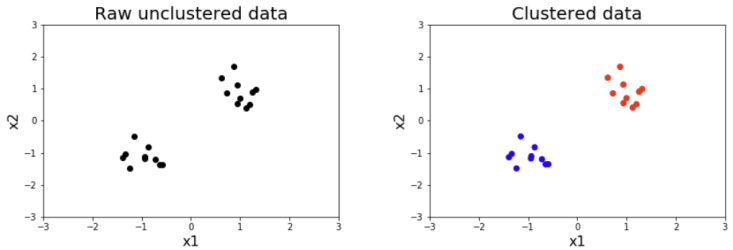


Figure 6.1: Left: unclustered raw data; Right: clustered data

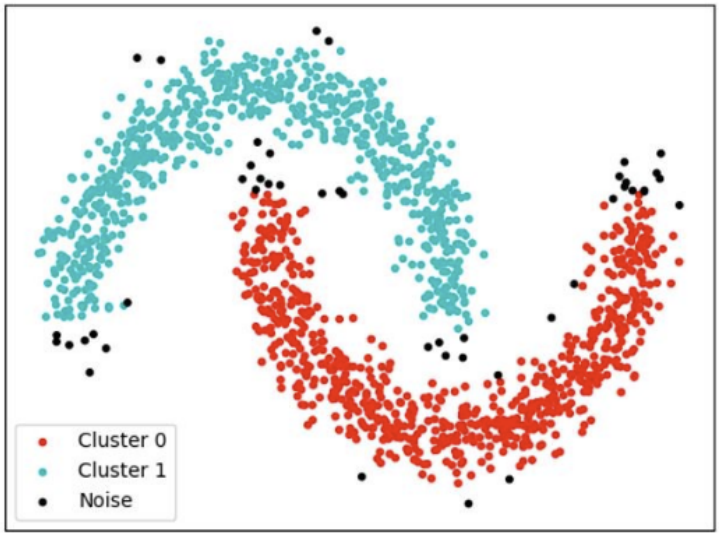


Figure 6.2: A nonspherical clustering assignment. Possible outliers are shown in black. Click here for details

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.

6.1 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:

$$\underset{\{C_k\}_{k=1}^K, \{c_k\}_{k=1}^K : X = C_1 \cup \cdots \cup C_K}{\arg\min} \sum_{k=1}^{K} \sum_{x \in C_k} \|x - c_k\|^2 \tag{6.1}$$

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.

6.2 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.

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.

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.