ESE 546: Principles of Deep Learning
Fall 2020

Instructor

Pratik Chaudhari pratikac@seas.upenn.edu

Teaching Assistants

Evangelos Chatzipantazis vaghat@seas
Rahul Ramesh rahulram@seas
Dushyant Sahoo sadu@seas
Rongguang Wang rgw@seas
Shiyun Xu shiyunxu@sas

December 10, 2020

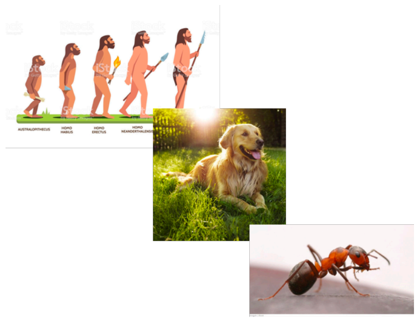
Contents

Chapter 1

What is intelligence?

Reading

1. Bishop 1.1-1.5
2. Goodfellow Chapter 1
3. “A logical calculus of the ideas immanent in nervous activity” by Warren McCulloch and Walter Pitts (McCulloch and Pitts, 1943).
4. “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.

Let us ask a different question. 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 Figure 1.1 is an interesting plant however. Tunicates are invertebrates. When they are young they roam around the ocean floor in search of nutrients, and they also have a nervous system (ganglion cells) at this point

Figure 1.1: A Tunicate on the ocean floor

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 (https://youtu.be/YrTRKh4FPio). 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 move around in the physical 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 a process that takes a dataset stored on your hard-disk and makes some predictions of its labels. It is much richer than that. 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, gathering more data, manipulating objects etc. The ability to do so is the hallmark of intelligence.

The purpose of intelligence is really just survival.

1.1 Key components of intelligence

If you agree with my definition, we can write down the three key parts that an intelligent, autonomous agent possesses as follows.

$$\begin{tikzcd}
\text{Perception} \arrow[r] & \text{Cognition} \arrow[r] & \text{Action} \arrow[ll, bend left]
\end{tikzcd}$$

Perception refers to the sensory mechanisms to gain information about the environment (eyes, ears, 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.

1.1.1 What is learning?

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 is much more than that, it also involves assimilating knowledge, handling situations when the current data does not match past data, e.g., arithmetic problems you solved in elementary school used your skills of algebraic manipulation to handle new problems.

Examples of other classes that address various aspects of intelligence are:

- Perception: CIS 580, CIS 581, CIS 680
- Learning: CIS 520, CIS 521, CIS 522, CIS 620, CIS 700, ESE 545
- Control: ESE 650, MEAM 620, ESE 505

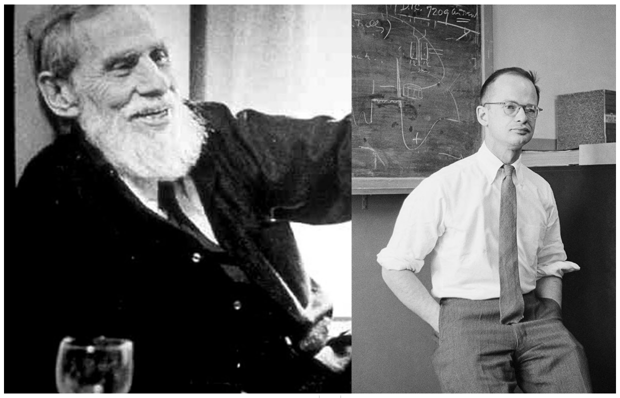
Imagine a supreme agent which is infinitely fast and clever can interpret its sensory data and compute the best actions for any task, say driving, it wishes. One would think learning on the past data is not essential to cognition, certainly not for this supreme agent. However, learning is essential to cognition. Priors help you if you are not as fast as the supreme agent or if you want to save some compute time/energy during decision making.

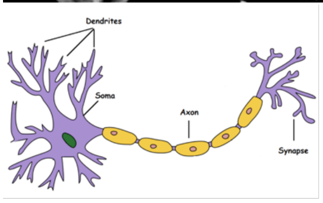
You should not think of a deep network or a machine learning model as a mechanism that directly undertakes the actions. It is better suited to provide a prior on the possible actions that an autonomous agent should take; other algorithms that rely on real-time sensory data will be in charge of picking one action out of these predictions. The objective of the learning process is really to crunch the data and learn a prior.

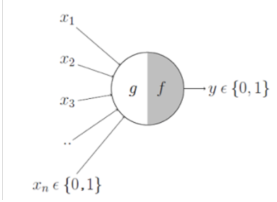
1.2 Intelligence: The Beginning (1942-50)

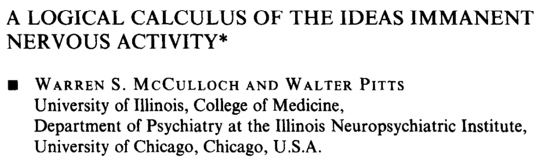
Let us give a short account of how our ideas about intelligence have evolved.

The story begins 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 perform complex functions. Their paper (McCulloch and Pitts, 1943) is an assigned reading for this lecture.





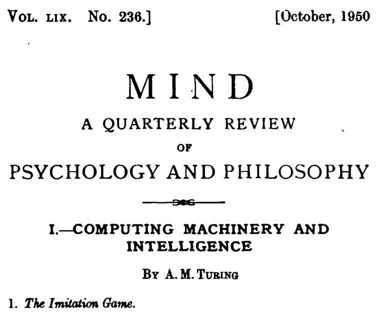




A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

■ 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.

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. [1]



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.*

McCulloch was inspired by Turing's idea that of building a machine that could compute any function in finitely-many steps was powerful. In his mind, the neuron in a human brain, which either fires or does not fire depending upon the stimuli of the neurons connected to it, was a binary object; rules of logic where a natural way to link such neurons, just like the Pitt's hero Bertrand Russell rebuilt modern mathematics using logic.

[1] If you need more inspiration to go and read it, the first section is titled "The Imitation Game".

Together, McCulloch & Pitts' and Turing's work already had all the germs of neural networks as we know them today: non-linearities, 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.

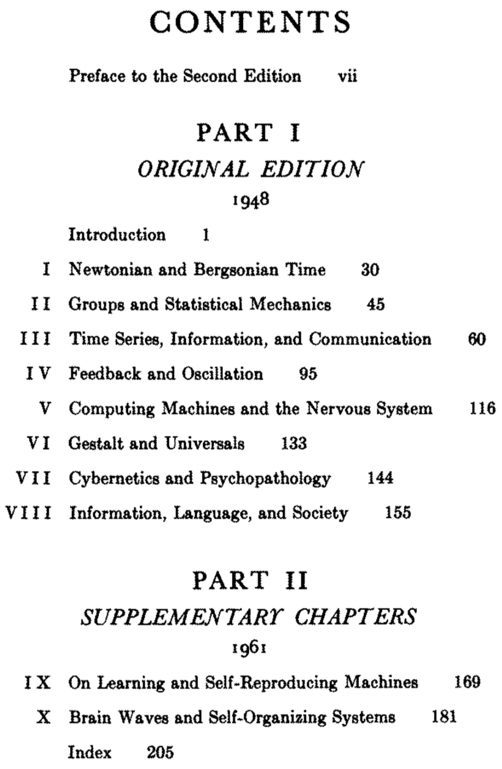
Figure 1.2. The four pioneers of cybernetics (*left to right*): Ross Ashby, Warren McCulloch, Grey Walter, and Norbert Wiener. Source: de Latil 1956, facing p. 53.

Figure 1.2: The famous four of the first era of intelligence. (From right to left) Norbert Wiener, Grey Walter, Warren McCulloch and Walter Pitts

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, "concepts, ideas, objects, categories" etc. These can be manipulated with tools in logic and inference. What is the information that is transferred from the perception system to the cognition system, or from cognition to control? An agent needs to maintain a notion of an internal representation that is the object being passed around.

We will often talk about Claude Shannon and information theory for studying these kind of ideas. Shannon devised one such representation learning scheme: that for compressing, coding, decoding and decompressing data. The key idea to grasp here is that the notion of information in information theory is slightly different from the one we need in machine learning. Compression, decompression etc. care about never losing information from the data; machine learning necessarily requires you forget parts of your 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.



CONTENTS

Figure 1.3: This course's content is (surprisingly) closely related to Wiener's book on Cybernetics.

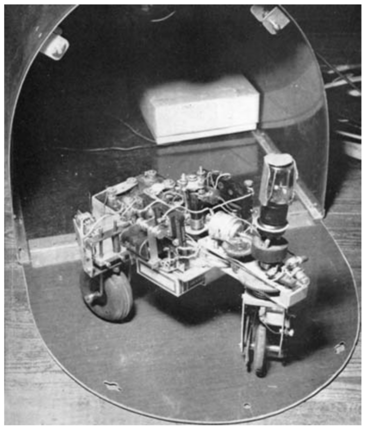


Figure 3.3. Anatomy of a tortoise. Source: de Latil 1956, facing p. 50.

Figure 1.4: Grey Walter's cybernetic tortoises named Elmer and Elsie were one of the first electronic autonomous robots (https://youtu.be/lLULRlmXkKo). Walter wanted to create an artificial brain, he wanted to show how neuron-like components connected together can give to complex behaviors, in this case this is a light sensitive robot that tracks the source of light.

Figure 1.5: 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 at https://www.technologyreview.com/2018/12/19/138508/mighty-mouse.

The study of 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.

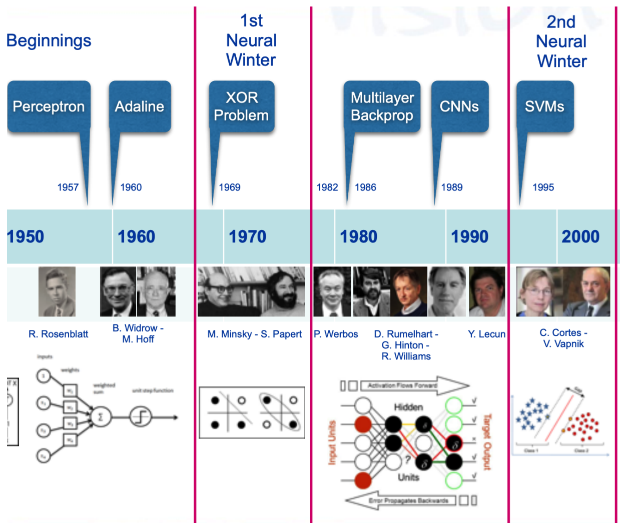
1.3 Intelligence: Reloaded (1960-2000)

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. 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 (https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon). 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 addition 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

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

Rosenblatt's perceptron (Rosenblatt, 1958) had a single neuron, it cannot distinguish between complex data. This is what Marvin Minsky and Seymour Papert discussed in a famous book Minsky and Papert (2017). This book was widely perceived as a death knell for the perceptron and interest in neuron-based artificial intelligence (connectionist approach) waned.

This coupled with the rise of symbolic reasoning in the early 1970s and resulted in 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). Multi-layer networks were in vogue due to back-propagation working well. Convolutional neural networks built upon a large body of work starting from two neuro-scientists 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 research area as it is in 2020 today.

Support Vector Machines (SVMs) were invented in Cortes and Vapnik (1995). These were (are) brilliant machine learning models with extremely good performance, were much easier to train than neural networks because they had a strong foundation in theory and, in general were a delight to use as compared to neural networks. Kernel methods, although known much before in the context of the perceptron (Aizerman, 1964; Scholkopf 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 remained popular through this period.

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 Boltzmann 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 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 chemistry and biology, medical sciences, and many many others (LeCun et al., 2015).

This progress in deep learning has been driven by the availability of data and cheap computation. Most importantly, it is driven today by the intense curiosity of people from diverse fields of inquiry. Deep learning in its modern form is a very young field. As is typical in new research fields, consolidation of ideas is difficult to come by. The dramatic progress today is driven by ideas that are often-quixotic and a large number of open problems remain in how we may build a more sophisticated understanding of deep networks.

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 2020. Our goals are to

1. become good at using modern deep networks, i.e., implementing them, training them, modeling specific problems using ideas in deep learning;
2. understanding why techniques in deep networks work.

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

Chapter 2

Linear Regression, Perceptron, Stochastic Gradient Descent

Reading

1. Bishop 3.1, 4.1, 4.3
2. Goodfellow Chapter 5.1-5.4

2.1 Problem setup for machine learning

Nature gives us data X and targets Y for this data.

$$X \to Y.$$

Nature does not usually tell us what property of a datum $x \in X$ results in a particular prediction $y \in Y$. We would like to learn to imitate Nature, namely predict y given x.

What does such learning mean? It is simply a notion of being able to identify patterns in the input data without explicitly programming a computer for prediction. We are often happy with a learning process that identifies correlations: if we learn correlations on a few samples $(x^1, y^1), \ldots, (x^n, y^n)$, we may be able to predict the output for a new datum x^{n+1}. We may not need to know *why* the label of x^{n+1} was predicted to be so and so.

Let us say that Nature possesses a probability distribution P over (X, Y). We will formalize the problem of machine learning as Nature drawing n independent and identically distributed samples from this distribution. This is denoted by

$$D_{\text{train}} = \left\{(x^i, y^i) \sim P\right\}_{i=1}^n$$

is called the "training set". We use this data to identify patterns that help make predictions on some future data.

What is the task in machine learning?

Suppose D_{train} consists of $n = 50$ RGB images of size 100×100 of two kinds, ones with an orange inside them and ones without. 10^4 is a large number of pixels, each pixel taking any of the possible 255^3 values. Suppose we discover that one particular pixel, say at location $(25, 45)$, takes distinct values in all images inside our training set. We can then construct a predictor based on this pixel. This predictor, it is a binary classifier, perfectly maps the training images to their labels (orange: +1 or no orange: -1). If x^k_{ij} is the $(ij)^{\text{th}}$ pixel for image x^k, then we use the function

❓ How many such binary classifiers are there at most?

$$f(x) = \begin{cases} y^k & \text{if } x^k_{ij} = x_{ij} \text{ for some } k = 1, \ldots, n \\ -1 & \text{otherwise.} \end{cases}$$

This predictor certainly solves the task. It correctly works for all images in the training set. Does it work for images outside the training set?

Our task in machine learning is to learn a predictor that works *outside* the training set. The training set is only a source of information that Nature gives us to find such a predictor.

> Designing a predictor that is accurate on D_{train} is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside D_{train}.

2.1.1 Generalization

If we never see data from outside D_{train} why should we hope to do well on it? The key is the distribution P. Machine learning is formalized as constructing a predictor that works well on new data that is also drawn independently from the distribution P. We will call this set of data the "test set".

$$D_{\text{test}}.$$

This assumption is important. It provides coherence between past and future samples: past samples that were used to train and future samples that we will wish to predict upon.

How to find such predictors that work well on new data? The central idea in machine learning is to restrict the set of possible binary functions that we consider.

> We are searching for a predictor that generalizes well but only have the training to ascertain which predictor works well.

The *right* class of functions f cannot be too large, otherwise we will find our binary classifier above as the solution and that is not too useful. The class of functions cannot be too small either, otherwise we won't be able to predict difficult images. If the predictor does not even work well on the training set, why should we expect it to work on the test set!

❓ Can you now think how is machine learning different from other fields you might know such as statistics or optimization?

Finding this correct class of functions with the right balance is what machine learning is all about.

2.2 Linear regression

Let us focus on a simpler problem. We fix the class of functions, our predictors, to only have linear classifiers. We will consider that our data $X \subset \mathbb{R}^d$ and labels $Y \subset \mathbb{R}$. If the labels/targets are real-valued, we call it is a regression problem. Our predictor for any $x \in X$ is

$$f(x; w, b) = w^\top x + b. \tag{2.1}$$

This is a linear function in the data x with parameters $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$. Different settings of w and b give access to different functions f. Picking a particular function f is therefore akin to picking particular values of the parameters. Parameters are also called weights. We can visualize what this predictor does in two ways, consider the case of $d = 2$.

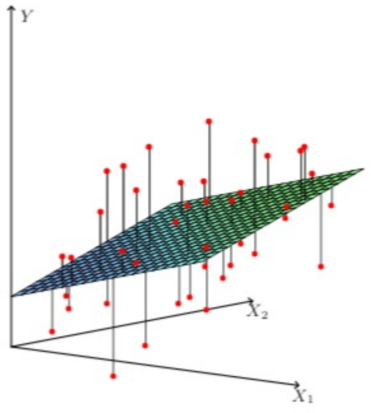


Figure 2.1: Linear least squares with $X \subset \mathbb{R}^2$.

Figure 2.1 shows the hyperplane corresponding to a particular (w, b) with the data x^i, y^i (in red). Each hyperplane is a particular predictor $f(x; w, b)$. You can also think of the function f as a point in three dimensional space $w \in \mathbb{R}^2$ and $b \in \mathbb{R}$.

Predicting the target accurately using this linear model would require us to find values (w, b) that minimize the average distance to the hyperplane of each sample in the training dataset. We write this as an *objective function*.

$$\begin{aligned}\ell(w, b) &:= \frac{1}{2n} \sum_{i=1}^{n} \left(y^i - \widehat{y^i}\right)^2 \\ &= \frac{1}{2n} \sum_{i=1}^{n} \left(y^i - w^\top x^i - b\right)^2\end{aligned} \tag{2.2}$$

where we have written the prediction as

$$\widehat{y^i} = w^\top x^i + b.$$

The quadratic term for each datum $\frac{1}{2}\left(y^i - \widehat{y^i}\right)^2$ is known as the *loss function*. The objective above is thus an average of the loss for each datum. Finding the

❓ Why use the average, as opposed to say the maximum value?

best weights w, b now boils down to solving the optimization problem

$$w^*, b^* = \underset{w\in\mathbb{R}^d, b\in\mathbb{R}}{\text{argmin}} \;\; \ell(w,b) \tag{2.3}$$

How to solve the optimization problem? We will learn many techniques to solve problems of the form (2.3). We have a simple case here and therefore can use what you did did in HW0. The solution is given by

$$w^* = (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^\top \mathbf{Y} \tag{2.4}$$

where we have denoted by $\tilde{\mathbf{X}} \in \mathbb{R}^{n\times(d+1)}$ the matrix whose i^{th} row is the datum with a constant entry 1 appended at the end $[x^i, 1]$. Similarly $\mathbf{Y} \in \mathbb{R}^n$ is a vector whose i^{th} entry is the target y^i.

❓ When is our solution to least squares regression in (2.4) not defined?

❓ What are we losing by fitting a linear predictor? Will this work if the true model from which Nature generates the data was different, say a polynomial?

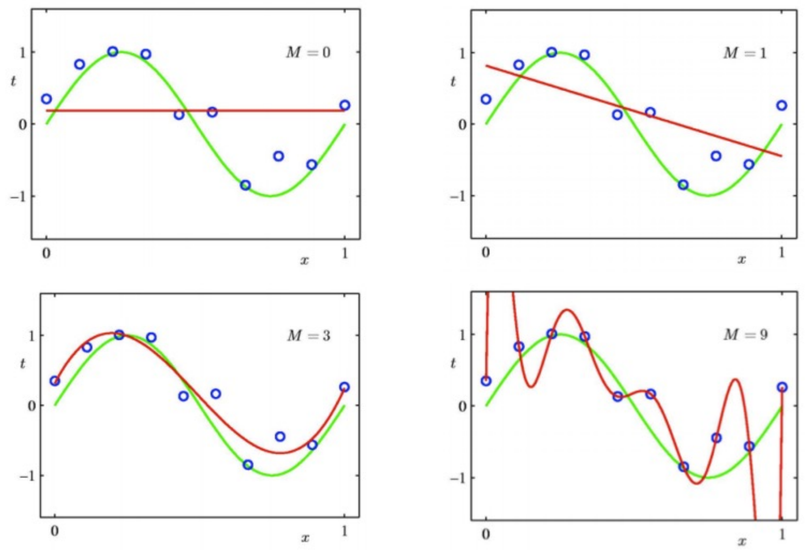


Figure 2.2: Least squares fitting using polynomials. As the degree of the polynomial M increases the predictor f fits the training data (in blue) better and better. But such a well-fitted predictor may be very different from the true model from which Nature generated the data (in green). The red curve in the fourth panel in these cases is said to have been *over-fitted*.

2.2.1 Maximum Likelihood Estimation

There is another perspective to fitting a machine learning model. We will *suppose* that our training data was created using a statistical model. We write this as

$$y = w^\top x + b + \epsilon \tag{2.5}$$

Of course we do not know whether Nature used this particular model $f(x; w, b) := w^\top + b$ to create the data, it might have created the data using some other model. This discrepancy between the models is *modeled* as noise ϵ. Noise in machine learning comes from the fact that we the user do not know Nature's model.

What model is appropriate for the noise ϵ? There can be many models depending upon your experiment (think of a model that predicts the arrival

❓ Can you think any other sources of noise? For instance, if you scraped some images from the Internet, how will you label them?

time of a bus at the bus stop, what noise would you use?). For our purpose we will use zero-mean Gaussian noise

$$\epsilon \sim N(0, \sigma_\epsilon^2)$$

that does not depend on the sample x. The probability that a sample (x^i, y^i) in our dataset D_{train} was created using our statistical model is then

$$p(y^i | x^i, w, b) = N(w^\top x^i + b,\ \sigma_\epsilon^2).$$

We have assumed that the data was drawn iid by Nature so the likelihood of our entire dataset is

$$p(D_{\text{train}} | w, b) = \prod_{i=1}^n p(y^i | x^i, w, b).$$

Finding good values of w, b can now be thought of as finding values that maximize the likelihood of our observed data

$$w^*, b^* = \underset{w,b}{\text{argmin}}\ -\log p(D_{\text{train}} | w, b). \tag{2.6}$$

Observe that our objective is written as the minimization of the *negative log-likelihood*. This is equivalent to maximizing the likelihood because logarithm is monotonic function. We can now rewrite the objective as

$$-\log p(D_{\text{train}} | w, b) = \frac{n}{2} \log(\sigma_\epsilon^2) + \frac{n}{2} \log(2\pi) + \frac{1}{2\sigma_\epsilon^2} \sum_{i=1}^n \left(y^i - w^\top x^i - b\right)^2.$$

Notice that only the third term depends on w, b. The first term is a function of our *chosen* value σ_ϵ^2, the second term is a constant. In other words, finding maximizing the likelihood boils down to solving the optimization problem

$$w^*, \beta^* = \underset{w,b}{\text{argmin}} \frac{1}{2\sigma_\epsilon^2} \sum_{i=1}^n \left(y^i - w^\top x^i - b\right)^2. \tag{2.7}$$

This objective is nothing other than our least squares regression objective with σ_ϵ^2 set to 1. This objective known as the maximum likelihood objective (MLE).

❓ How does using a different value of σ_ϵ in (2.7) change the least squares solution in (2.4)?

Maximum likelihood objective has an interesting offshoot. In the least squares case, given an input x all that our fitted model could predict was

$$\widehat{y} = {w^*}^\top x + b^*.$$

MLE has helped us fit a statistical model to the data. So we can now predict the entire distribution

$$p(y | x, w^*, b^*) = N({w^*}^\top x + b^*, \sigma_\epsilon^2).$$

The solution of least squares is the mean of the Gaussian random variable $y | x, w^*, b^*$, the variance of this random variance is σ_ϵ^2. So instead of just predicting $\widehat{y}$ the machine learning model can now give the probability distribution $p(y | x, w^*, b^*)$ as the output and the user is free to use it as they wish, e.g., compute the mean, the median, the 5% probability value of the right tail etc.

2.3 Perceptron

Let us now solve a classification problem. We will again go around the model selection problem and consider the class of linear classifiers. Assume binary labels $Y \in \{-1, 1\}$. To keep the notation clear, we will use the trick of appending a 1 to the data x and hide the bias term b in the linear classifier. The predictor is now given by

$$\begin{aligned} f(x; w) &= \text{sign}(w^\top x) \\ &= \begin{cases} +1 & \text{if } w^\top x \geq 0 \\ -1 & \text{else.} \end{cases} \end{aligned} \tag{2.8}$$

We have used the sign function denoted as sign to get binary $\{-1, +1\}$ outputs form our real-valued prediction $w^\top x$. This is the famous perceptron model of Frank Rosenblatt. We can visualize the perceptron the same way as we did for linear regression.

Let us now formulate an objective to fit/train the perceptron. As usual, we want the predictions of the model to match those in the training data.

$$\ell_{\text{zero-one}}(w) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{y^i \neq f(x^i; w)\}}. \tag{2.9}$$

The indicator function inside the summation measures the number of mistakes the perceptron makes on the training dataset. The objective here is designed to find weights w that minimizes the average number of mistakes, also known as the training error. Such a loss that measures the mistakes is called the zero-one loss, it incurs a penalty of 1 for a mistake and zero otherwise.

❓ Is a linear model appropriate if our data was natural images? What properties have we lost by restricting the classifier to be linear?

⚠ 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.

❓ Can you think of some quantity other than the zero-one error that we may wish to optimize?

2.3.1 Surrogate Losses

The zero-one loss is the clearest indication of whether the perceptron is working well. It is however non-differentiable, so we cannot use powerful ideas from optimization theory to minimize it. This is why surrogate losses are constructed in machine learning. These are proxies for the loss function, typically for the classification problems and look as follows.

The hinge loss is one such surrogate loss. It is given by

$$\ell_{\text{hinge}}(w) = \max(0, -y\, w^\top x).$$

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

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

or the logistic loss

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

are some other popular losses for classification.

❓ Draw the three losses to observe their differences.

2.4 Stochastic Gradient Descent

We will now fit a perceptron using the hinge loss using a very simple optimization technique. At each iteration, this algorithm updates the weights w in the direction of the negative gradient. So first, let us compute the gradient of the hinge loss. It is easily seen to be

$$\frac{\mathrm{d}\ell_{\text{hinge}}(w)}{\mathrm{d}w} = \begin{cases} -y\,x & \text{for incorrect prediction} \\ 0 & \text{else.} \end{cases} \tag{2.10}$$

We will use a naive algorithm to update the weights. Here is how it goes.

The Perceptron algorithm

Perform the following steps for iterations $t = 1, 2, \ldots$.

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

$$w^{t+1} = \begin{cases} w^t + y^{\omega_t} x^{\omega_t} & \text{if } \text{sign}({w^t}^\top x^{\omega_t}) \neq y^{\omega_t} \\ w^t & \text{else.} \end{cases} \tag{2.11}$$

In other words, the perceptron weights is changed only if it makes a mistake on the sample $(x^{\omega_t}, y^{\omega_t})$. The updated perceptron improves its mistake on this sample. Observe that a mistake happens if the sign of ${w^t}^\top x^{\omega_t}$ and y^{ω_t} are different, the product $y^{\omega_t} w^\top x^{\omega_t}$ is therefore negative. The updated weights of the perceptron now satisfy

$$\begin{aligned} y^{\omega_t}(w^t + y^{\omega_t} x^{\omega_t})^\top x^{\omega_t} &= y^{\omega_t} \left\langle w^t, x^{\omega_t} \right\rangle + (y^{\omega_t})^2 \left\langle x^{\omega_t}, x^{\omega_t} \right\rangle \\ &= y^{\omega_t} \left\langle w^t, x^{\omega_t} \right\rangle + \|x^{\omega_t}\|_2^2. \end{aligned}$$

In simple worlds, the value of $y^{\omega_t} \langle w, x^{\omega_t} \rangle$ increases as a result of the update, it becomes more positive. If the perceptron makes mistakes on the same datum repeatedly, this value is eventually going to become positive. Of course, mistakes on other data in the training set may steer the perceptron towards other directions and it may continue to cycle ad infinitum, but it is easy to show that it ceases its updates when all data are correctly classified. More precisely, if the training data are such that they can be correctly classified using a linear predictor, then the perceptron will find this predictor after a finite number of iterations.

We have seen a powerful algorithm for machine learning. This algorithm is called stochastic gradient descent (SGD) and it is very general: so long as you can take the gradient of the objective you can execute SGD. The algorithm for fitting the perceptron above was given by Rosenblatt in 1957 and is popularly known as the "perceptron algorithm". It is interesting to note that it is simply the instantiation of SGD which was known before Robbins and Monro (1951) for the hinge loss.

❓ You may have seen the hinge loss written as $\ell_{\text{hinge}}(w) = \max(0, 1 - y\, w^\top x)$. Why the difference?

2.4.1 The general form of SGD

SGD is a very general algorithm. We can use it so long as you have a dataset and an objective that is differentiable. Consider an optimization problem that looks like

$$w^* = \underset{w}{\text{argmin}}\ \frac{1}{n}\sum_{i=1}^{n} \ell^i(w)$$

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

$$w^{t+1} = w^t - \eta \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w}\Big|_{w=w^t}.$$

We have chosen to be a bit more precise and the sample over which we compute the gradient is ω_t. This is a random variable with domain and we will use ω_t to denote its index.

$$\omega_t \in \{1, \dots, n\}.$$

The gradient of the loss $\ell^{\omega_t}(w)$ with respect to w is denoted by

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

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

$$\nabla_{w_1} \ell^{\omega_t}(w^t) = \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w_1}\Big|_{w=w^t}$$

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

$$w^{t+1} = w^t - \eta\, \nabla\, \ell^{\omega_t}(w^t). \tag{2.12}$$

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 $-\nabla\, \ell^{\omega_t}(w^t)$ at each iteration.

Chapter 3

Kernels, Beginning of neural networks

Reading

1. Bishop 6.1-6.3
2. Goodfellow 6.1-6.4
3. "Random features for large-scale kernel machines" by Rahimi and Recht (2008).

3.1 Digging deeper into the perceptron

3.1.1 Convergence rate

How many iterations does a perceptron need to fit on a given dataset? We will assume that the training data are bounded, i.e., $\|x^i\| \leq R$ for some R and for all $i \in \{1, \ldots, n\}$. Let us also assume that the training dataset is indeed linearly separable, i.e., a solution w^* exists for the perceptron weights with training error exactly zero. This means

$$y^i {w^*}^\top x^i > 0 \quad \forall i.$$

We will also assume that this classifier *separates the data well*. Note that the distance of each input x^i from the decision boundary (i.e., all x such that ${w^*}^\top x = 0$) is given by the component of x^i in the direction of w^* if the label is $y^* = +1$ and in the direction $-w^*$ if the label is negative. In other words,

$$\frac{y^i {w^*}^\top x^i}{\|w^*\|} = \rho^i$$

gives the distance to the decision boundary. The quantity on the right hand side is called the *margin*, it is simply the distance of the sample i from the decision boundary. If w^* is the classifier with the largest average margin,

$$\rho = \min_{i \in \{1,\dots,n\}} \rho^i$$

is a good measure of how hard a particular machine learning problem is.

You can now try to prove that after each update of the perceptron the inner product of the current weights with the try 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. Together the two will give you a result that after t weight updates

$$t \leq \frac{R^2}{\rho^2} \tag{3.1}$$

all training data are classified correctly. Notice a few things about this 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 rare.

3. The number of updates scales with the hardness of the problem; if the margin ρ was small, we need lots of updates to drive the training error to zero.

3.1.2 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 (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 a linear combination

$$w^* = \sum_{i=1}^n \alpha^i y^i x^i. \tag{3.2}$$

where $\alpha^i \in \{0, 1, \dots, \}$. The perceptron therefore using the classifier

$$\begin{aligned} f(x, w) &= \text{sign}(\hat{y}) \\ \text{where} \quad \hat{y} &= \left(\sum_{i=1}^n \alpha^i y^i x^i \right)^\top x \\ &= \sum_{i=1}^n \alpha^i y^i {x^i}^\top x. \end{aligned} \tag{3.3}$$

⚠ As you see in (3.3), computing the prediction for a new input x involves, either remembering all the weights w at the end of training, or storing all the $\{\alpha^i\}_{i=1,\dots,n}$ along with the training dataset. The latter is called the dual representation of a perceptron and the scalars $\{\alpha^i\}$ are called the dual parameters.

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 is a measure of how many tries it took the perceptron to fit that sample during training.

3.2 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 saw 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.

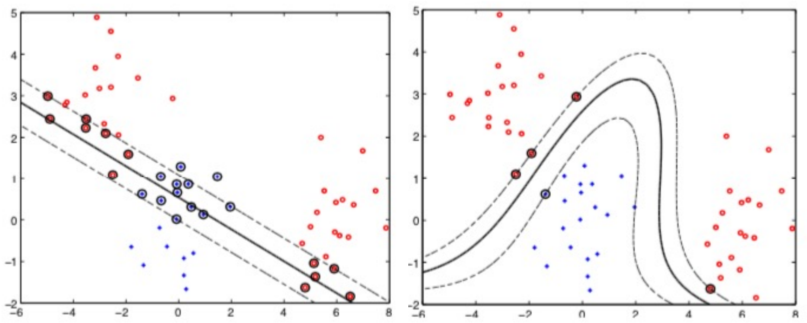


Figure 3.1

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 polynomial expansion 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 $\left\{\phi(x^i), y^i\right\}$. This involves updating the weights at each iteration as

$$w_{t+1} = \begin{cases} w_t + y^t\phi(x^t) & \text{if } \operatorname{sign}(w_t^\top \phi(x^t)) \neq y^t \\ w_t & \text{else.} \end{cases} \tag{3.4}$$

At the end of such training, the perceptron is

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

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

$$f(x; w) = \operatorname{sign}\left(\sum_{i=1}^n \alpha^i y^i \phi(x^i)^\top \phi(x)\right). \tag{3.5}$$

Notice that we now have a linear combination of the *features* not the data directly.

❓ 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 data 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?

3.3 Kernels

Observe the expression of the classifier in (3.5). Each time we make predictions on the 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 the 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 the inner product

⚠ Feature spaces can become large very quickly. What is the dimensionality of $\phi(x)$ for a tenth-order polynomial with a three-dimensional input data?

$$\begin{aligned}
\phi(x) &= \left[1, \sqrt{2}x, x^2\right] \\
\phi(x') &= \left[1, \sqrt{2}x', x'^2\right] \\
\phi(x)^\top \phi(x') = 1 + 2xx' + (xx')^2 &= (1 + xx')^2 .
\end{aligned}$$

for input $x \in \mathbb{R}$. Kernels are a formalization of exactly this idea. A kernel

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

is a symmetric, positive semi-definite function of its two arguments for which it holds that

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

for some feature ϕ. Few examples of kernels are

$$\begin{aligned}
k(x, x') &= \left(x^\top x' + c\right)^2 , \\
k(x, x') &= \exp\left(-\|x - x'\|^2/(2\sigma^2)\right) .
\end{aligned}$$

3.3.1 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

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

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

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

then update

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

Notice that we do not ever compute $\phi(x)$ so it does not matter what the dimensionality of the feature vector is. It can even be infinite, e.g., for the radial basis function kernel. Observe also that we do not maintain weights w. We instead maintain the dual variables $\{\alpha^1, \ldots, \alpha^n\}$ while running the algorithm.

Note that the kernel perceptron computes the kernel over *all* data samples in the training set at each iteration. It is expensive and seems wasteful. The Gram matrix denoted by $G \in \mathbb{R}^{n\times n}$

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

helps address this problem by computing the kernel on all pairs in the training dataset. With this in hand, we can modify step 2 in the kernel perceptron using

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

where $e_t = [0, \ldots, 0, 1, 0, \ldots]$ with a 1 on the t^{th} element, $\alpha = \left[\alpha^1, \ldots, \alpha^n\right]$ denotes the vector of all the dual variables, $Y = \left[y^1, \ldots, y^n\right]$ is a vector of all the labels, and the notation $\alpha \odot Y = \left[\alpha^1 y^1, \ldots, \alpha^n y^n\right]$ denotes the element-wise (Hadamard) product. This expression now only involves a matrix-vector multiplication, which is much easier than computing the kernel at each iteration. Gram matrices can become very big. If the number of samples is $n = 10^6$, not an unusual number today, the Gram matrix has 10^{12} elements. The big failing of kernel methods is that they require a large amount of memory at training time. Nystrom methods compute low-rank approximations of the Gram matrix which makes operations with kernels easier.

❓ Kernels look great, you can fit perceptrons in powerful feature spaces using essentially the same algorithm. How expensive is each iteration of the perceptron?

⚠ When ML algorithms are implemented in a system, there exist tradeoffs between the feature-space version and the Gram matrix version of linear classifiers. The former is preferable if the number of samples in the dataset is large, while the latter is used when the dimensionality of features is large.

3.3.2 Mercer's theorem

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

Theorem 3.1 (Mercer's Theorem). For any symmetric function $k : X \times X \to \mathbb{R}$ which is square integrable in $X \times X$ and satisfies

$$\int_{X\times X} k(x, x')\; f(x)\; f(x')\; \mathrm{d}x\; \mathrm{d}x' \geq 0 \tag{3.7}$$

for all square integrable functions $f \in L_2(X)$, there exist functions $\phi_i : X \to \mathbb{R}$ and numbers $\lambda_i \geq 0$ where

$$k(x, x') = \sum_i \lambda_i \phi_i^\top(x)\; \phi_i(x')$$

for all $x, x' \in X$. The condition in (3.7) is called Mercer's condition. You will also have seen it written as *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 λ_i in the summation.

❓ Logistic regression with a loss function

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

is also a linear classifier. Write down how you will fit a logistic regression using stochastic gradient descent; this is similar to the perceptron algorithm. Write down the feature-space version of the algorithm and a kernelized logistic regression that uses the Gram matrix.

💡 A function $f : X \to \mathbb{R}$ is square integrable iff

$$\int_{x\in X} |f(x)|^2 \;\mathrm{d}x < \infty.$$

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:

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

This is easy to show.

$$\begin{aligned}
u^\top G u &= \sum_{ij} u_i u_j G_{ij} \\
&= \sum_{ij} u_i u_j \phi(x_i)^\top \phi(x_j) \\
&= \left(\sum_i u_i \phi(x_i) \right)^\top \left(\sum_j u_j \phi(x_j) \right) \\
&= \|\sum_i u_i \phi(x_i)\|^2 \\
&\geq 0.
\end{aligned}$$

The integral in Theorem 3.1 in Mercer's condition is really just the continuous analogue of the vector-matrix-vector multiplication above. So 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.

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 feature vector to use is to consider the set of all length k subsequences. The number of components in this feature vector is exponential. However, as you can imagine, given two strings x, x'

```
this string is interesting
txws sbhtqg is iyubqtnhpqg
```

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

3.4 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 can pick all polynomials; we can pick all possible Gabor filters that you saw in HW 1; we can also pick all possible string kernels.

3.4.1 Random features

Suppose that we have a finite-dimensional feature $\phi(x) \in \mathbb{R}^p$. We saw in the perceptron that

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

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

$$\phi(x) = \sigma\left(S^\top x\right), \tag{3.8}$$

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

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

We will abuse notation that denote both the vector version of σ and the element-wise version of σ 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 $k(x, x') = k(x - x')$ one may use a matrix with random elements as our S

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

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

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

is a cosine function. 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

$$f(x;w) = \text{sign}\left(w^\top \sigma\left(S^\top x\right)\right) \tag{3.9}$$

and we can solve the optimization problem

$$w^* = \underset{w}{\text{argmin}}\, \frac{1}{n} \sum_{i=1}^n \ell_{\text{hinge}}(y^i, \hat{y}^i) \tag{3.10}$$

with $\hat{y^i} = w^\top \sigma\left(S^\top x^i\right)$ and fit the weights w using SGD as before.

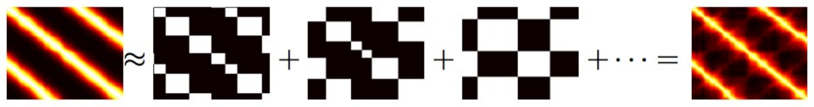


Figure 3.2

As an example consider the heatmap of Gabor-like kernel $k(x, x')$ in Figure 3.2 on the left. We can think of the decomposition

$$\begin{aligned}\text{left-most picture} = k(x, x') &= \phi(x)^\top \phi(x') \\ &= w_1 \sigma\left(\omega_1^\top x\right) + w_2 \sigma\left(\omega_2^\top x\right) + w_k \sigma\left(\omega_k^\top x\right) + \dots \\ &= \text{right-most picture}\end{aligned}$$

In other words, the random elements of the matrix S, namely ω_k can combine together *linearly* using the trained weights w_k to give us a kernel that looks like a useful kernel on the left. A large random matrix S allows us to learn may such kernels and combine their output linearly.

3.4.2 Learning the feature matrix as well

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

❓ What kind of data do you think random features will work well for?

we can design random features of the form

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

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/kernel.

We can now simply motivate deep learning as learning the matrix S in (3.9) in addition to the coefficients w. The classifier now is

$$f(x; w, S) = \text{sign}\left(w^\top \sigma\left(S^\top x\right)\right) \tag{3.11}$$

but we now solve the optimization problem

$$w^*, S^* = \underset{w,S}{\text{argmin}} \frac{1}{n} \sum_{i=1}^n \ell_{\text{hinge}}(y^i, \hat{y}^i) \tag{3.12}$$

with $\hat{y}^i = w^\top \sigma\left(S^\top x^i\right)$ as before. We have hereby seen our first *deep network*. The classifier in (3.11) is a two-layer neural network.

Moving from the problem in (3.10) to this new problem in (3.12) is a very big change.

1. **Nonlinearity.** The classifier in (3.11) is not linear anymore. It is a nonlinear function of its parameters w, 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 (3.12) much harder than the one in (3.10). The latter is a convex function (we will discuss this soon) which are easy to minimize. The former is a non-convex function in its parameters w, 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 4

Deep fully-connected networks, Backpropagation

Reading

1. Bishop 5.1, 5.3
2. Goodfellow 6.3-6.5
3. Notes at http://cs231n.github.io/optimization-2/

4.1 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

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

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

$$f(x; v, S_1, \ldots, S_L) = \text{sign}\left(v^\top \sigma\left(S_L^\top \ldots \sigma\left(S_2^\top\, \sigma(S_1^\top x)\right) \ldots\right)\right). \tag{4.1}$$

We will call each operation of the form $\sigma\left(S_k^\top \ldots\right)$, as a *layer*. Consider the second layer: it takes the features generated by the first layer, namely $\sigma(S_1^\top x)$, multiplies these features using its feature matrix $S_2^\top$ 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 Figure 4.1 shows the features (more precisely, the kernel) learnt by a deep neural network. The first layer of features are called Gabor-like, they are similar to ones you constructed in HW 1. These features are *combined* linearly along with a nonlinear operation to give richer features (spirals, right angles) in the middle panel. The third layer combines the lower features to get even more complex features, these look like patterns (notice a soccer ball in the bottom left), a box on the bottom right etc.

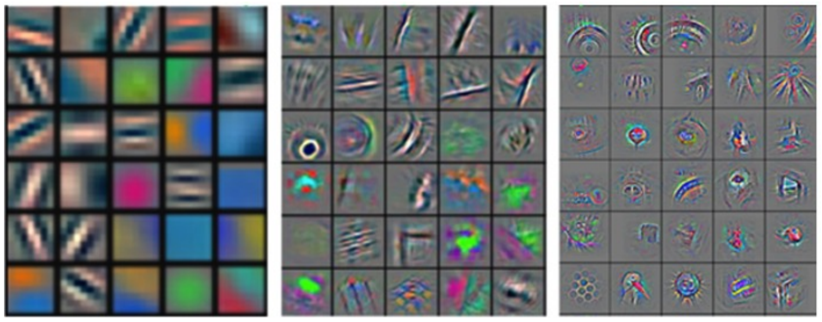


Figure 4.1

The optimization problem for fitting a deep network is written as

$$v^*, S_1^*, \ldots, S_L^* = \underset{v, S_1, \ldots, S_L}{\text{argmin}} \ \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(y^i, \hat{y}^i). \tag{4.2}$$

where the output prediction is now

$$\hat{y} = v^\top \sigma\left(S_L^\top \ldots \sigma\left(S_2^\top\, \sigma(S_1^\top x)\right) \ldots\right).$$

Notice that if fitting a two-layer network was difficult, then fitting a multi-layer neural network like (4.1) 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 (4.2) 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 (4.1) 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 field are adopting deep networks.

2. Before the resurgence of deep learning, each of these fields essentially had their own favorite kernels they preferred, 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, it means that 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.

4.1.1 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 (4.1) is called the activation function (motivated from the threshold-based activation of 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
$$\text{threshold}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{else.} \end{cases}$$
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)
$$\begin{aligned} \text{relu}(x) &= |x|_+ \\ &= \max(0, x). \end{aligned}$$
5. Leaky ReLUs
$$\sigma_c(x) = \begin{cases} x & \text{if } x > 0 \\ c\,x & \text{else.} \end{cases}$$
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/tanh nonlinearities in the backpropagation section.

❓ How would you use a binary classifier to classify 10 classes?

Logits for multi-class classification. The output
$$\hat{y} = v^\top \sigma\left(S_L^\top \ldots \sigma\left(S_2^\top\, \sigma(S_1^\top x)\right) \ldots\right)$$
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 the two classes, we would like to *interpret* the vector $\hat{y}$ as the log-probabilities of an input belonging to one of the classes.

❓ What would the shape of w be if you were performing regression using a deep network?

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

$$h^l = S_l^\top \dots \sigma\left(S_2^\top\, \sigma(S_1^\top x)\right) \dots \tag{4.3}$$

to get the *pre-activation* output of the l^{th} layer. The *post-activation* output is given by applying the nonlinearity

$$\sigma(h^l).$$

Sometimes people will call the $\sigma(h^L)$ as 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, e.g., you can use a pre-trained deep network on the ImageNet dataset in PyTorch within two lines of code and use these features to fit a linear classifier for classifying fruits. Training a deep network yourself on ImageNet is quite difficult.

Hidden layers/neurons. The intermediate layers that create the features $h^1, \dots, h^L$ are called the hidden layers. A feature is the same as a neuron; think of the McCulloch-Pitts picture, just like a neuron takes input from all the other neurons connected to it via some weights , a feature is computed using a weighted combination of the features at the lower layer. We will say that a neural network is *wide* if it has lots of features/neurons on each hidden layer. We will say that it is *thin* if it has few features/neurons on each hidden layer.

4.1.2 Weights

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

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

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

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

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

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

$$f(x, w) \tag{4.4}$$

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

$$w^* = \underset{w}{\text{argmin}} \ \frac{1}{n}\sum_{i=1}^{n} \ell(y^i, \hat{y}^i). \tag{4.5}$$

We will also sometimes denote the loss of the i^{th} sample as

$$\ell^i(w) := \ell(y^i, \hat{y}^i).$$

4.2 The backpropagation algorithm

We would like to using SGD to fit a deep network on a given dataset. As we saw in Chapter 2, if the loss function is denoted by $\ell^{\omega_t}(w)$ where ω_t was the index of the datum sampled at iteration t, we would like to update the weights using

$$w^{t+1} = w^t - \eta \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w}\Big|_{w=w^t}.$$

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

$$\frac{\mathrm{d}\ell(w)}{\mathrm{d}w}.$$

Backpropagation is an algorithm for computing the gradient of the loss function for a deep network.

4.2.1 One hidden layer with one neuron

Consider the linear regression problem with one layer and one datum:

$$\ell(w) = \frac{1}{2}(y - v\sigma(w^\top x))^2$$

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:

$$w, x \quad \underbrace{\longmapsto}_{\text{layer 1}} \quad z \quad \underbrace{\overset{\sigma}{\longmapsto}}_{\text{layer 2}} \quad h \quad \underbrace{\overset{v}{\longmapsto}}_{\text{layer 3}} \quad vh \quad \underbrace{\overset{y}{\longmapsto}}_{\text{layer 4}} \quad \ell. \tag{4.6}$$

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

$$\frac{\partial \ell}{\partial v} = (y - v\sigma(w^\top x)) \left(-\sigma(w^\top x)\right) \tag{4.7}$$

and

$$\frac{\partial \ell}{\partial w} = (y - v\sigma(w^\top x)) \left(-v\sigma'(w^\top x)\right) (x)\,. \tag{4.8}$$

1. **Caching computations for 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 (4.7) and (4.8). If we can cache these quantities we can compute the chain rule-based gradient for the different parameters quickly.

2. **Forward computation.** 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. **Backward computation.** The third observation is to see that the quantity $\sigma'(z)$ in (4.8) is the derivative of the output of the activation function, namely $h = \sigma(z)$ with respect to z, its input argument
$$\sigma'(z) = \frac{\mathrm{d}h}{\mathrm{d}z}.$$
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 (4.6) 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 all this process very mechanical and easy. Denote by

$$\overline{v} = \frac{\mathrm{d}\ell}{\mathrm{d}v} \tag{4.9}$$

the derivative of the loss ℓ with respect to a parameter v. Effectively, for our simple two layer (one neuron) neural network, we are interested in computing the quantities

$$\overline{w}, \quad \overline{v}.$$

Let us also denote the output of the second linear layer (layer 3)

$$e = vh.$$

Now observe the following "forward computation"

$$
\begin{aligned}
z &= w^\top x && (4.10)\\
h &= \sigma(z) && (4.11)\\
e &= vh && (4.12)\\
\ell &= \frac{1}{2}(y-e)^2. && (4.13)
\end{aligned}
$$

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.

$$
\begin{aligned}
\frac{\mathrm{d}\ell}{\mathrm{d}\ell} &= \overline{\ell} = 1.\\
\mathbb{R} \ni \overline{e} &= \overline{\ell}\,\frac{\mathrm{d}\ell}{\mathrm{d}e}\\
&= -1\,(y-e) = \overline{\ell}\,(-(y-e)). && \text{(from (4.13))}\\
\mathbb{R} \ni \overline{v} &= \overline{e}\,\frac{\mathrm{d}e}{\mathrm{d}v}\\
&= -\,(y-e)\ \ h = \overline{e}\,h. && \text{(from (4.12))}\\
\mathbb{R} \ni \overline{h} &= \overline{e}\,\frac{\mathrm{d}e}{\mathrm{d}h}\\
&= \overline{e}\,(v). && \text{(from (4.12))}\\
\mathbb{R} \ni \overline{z} &= \overline{h}\,\frac{\mathrm{d}h}{\mathrm{d}z}\\
&= \overline{h}\,\sigma'(z). && \text{(from (4.11))}\\
\mathbb{R}^d \ni \overline{w} &= \overline{z}\,\frac{\mathrm{d}z}{\mathrm{d}w}\\
&= \overline{z}\,x. && \text{(from (4.10))}\\
\mathbb{R}^d \ni \overline{x} &= \overline{z}\,\frac{\mathrm{d}z}{\mathrm{d}x}\\
&= \overline{z}\,w. && \text{(from (4.10))}
\end{aligned}
$$

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

$$z = w_1 x_1 + w_2 x_2$$

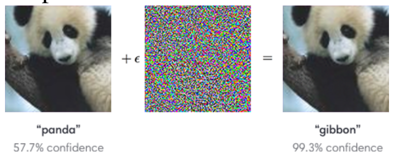
the backprop gradient is $\overline{w_1} = \overline{z}\,x_1$ and $\overline{w_2} = \overline{z}\,x_2$. If x_1 was large and dominated the computation of z during the forward propagation, then w_1 which is the multiplier of x_1 also gets a dominant share of the backprop gradient $\overline{z}$. The backprop 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

$$\frac{\mathrm{d}\ell}{\mathrm{d}x}$$

as a by-product of backpropagation. Backpropagation computes the gradient of the input activations to each layer $\overline{v}$ 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 $\overline{x}$.

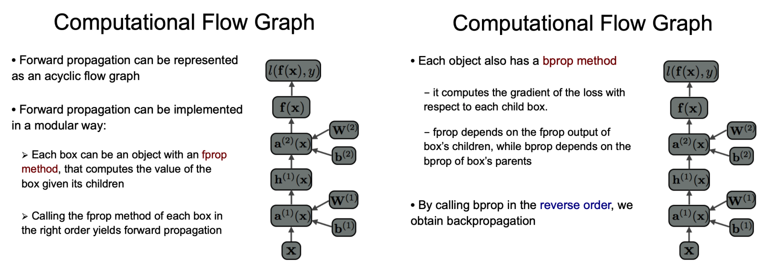
An example adversarial input to a deep network



4.2.2 Implementation of backpropagation

Consider our neural network classifier given by

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


Computational Flow Graph

- Forward propagation can be represented as an acyclic flow graph
- Forward propagation can be implemented in a modular way:
  - Each box can be an object with an fprop method, that computes the value of the box given its children
  - Calling the fprop method of each box in the right order yields forward propagation

Computational Flow Graph

- Each object also has a bprop method
  - it computes the gradient of the loss with respect to each child box.
  - fprop depends on the fprop output of box's children, while bprop depends on the bprop of box's parents
- By calling bprop in the reverse order, we obtain backpropagation

$l(\mathbf{f}(\mathbf{x}), y)$
$\mathbf{f}(\mathbf{x})$
$\mathbf{W}^{(2)}$
$\mathbf{a}^{(2)}(\mathbf{x})$
$\mathbf{b}^{(2)}$
$\mathbf{h}^{(1)}(\mathbf{x})$
$\mathbf{W}^{(1)}$
$\mathbf{a}^{(1)}(\mathbf{x})$
$\mathbf{b}^{(1)}$
$\mathbf{x}$

Figure 4.2: A schematic of 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 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 really bothering to compute the derivatives yourself.

Chapter 5

Convolutional Architectures

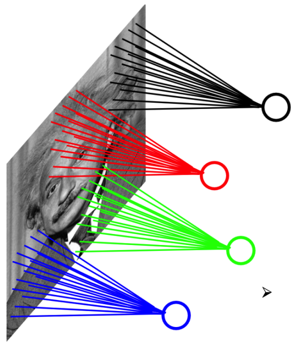
Reading

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

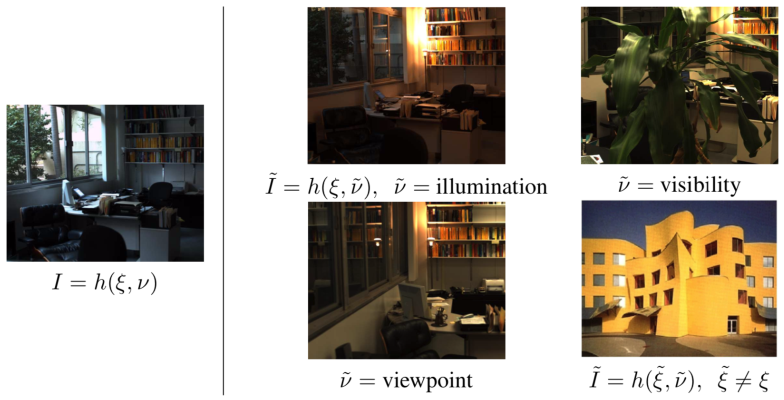
We have been talking about "fully-connected" neural networks till now. There are a few problems that are apparent even in our limited experience.

Fully-connected layers have lot of parameters. If an input image is of size $100 \times 100 = 10^4$ grayscale pixels and we would like to classify it as belonging to one out of 1000 classes, we need 10M parameters. It is difficult to perform so many add-multiply operations quickly even on sophisticated hardware. Further, the curse of dimensionality never goes away; we need lots of data to fit these many parameters.

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

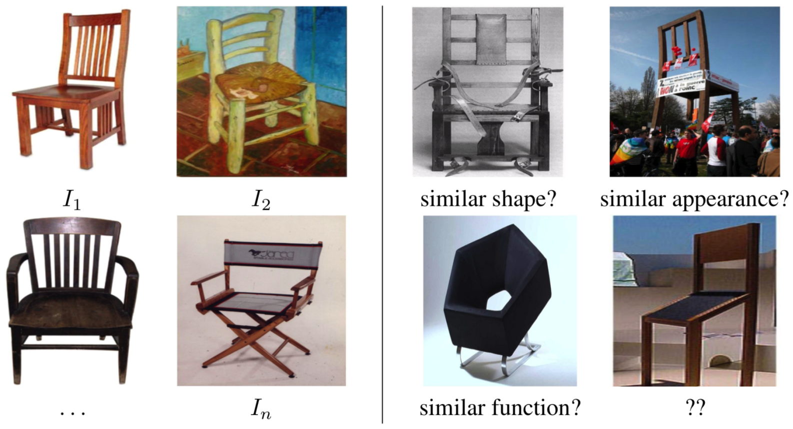


Natural data is full of "nuisances" that are not useful for tasks such as classification. E.g., illumination, viewpoint, and occlusions



$I = h(\xi, \nu)$ | $\tilde{I} = h(\xi, \tilde{\nu}),\ \ \tilde{\nu} =$ illumination | $\tilde{\nu} =$ visibility | $\tilde{\nu} =$ viewpoint | $\tilde{I} = h(\tilde{\xi}, \tilde{\nu}),\ \ \tilde{\xi} \neq \xi$

or even semantic ones shown below



Do fully connected networks work for such different images?

Nuisances can be defined as operations that act on the data before you get to see it (nature creates these nuisances). Some of them are special and they have a group structure, i.e., they satisfy certain algebraic conditions https://en.wikipedia.org/wiki/Group_(mathematics). For instance, images of the same chair taken from different vantage points are projections of different rigid body transformations of the camera. Some other nuisances such as occlusions do not have a group structure, e.g., there is no rigid body transformation that allows us to backcalculate the pixels belonging to a person standing behind a car. Convolutional layers are a simple way to tackle one particular kind of nuisance, that of translations.

5.1 Basics of the convolution operation

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

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

The basic unit of a convolutional neural network is

$$\sigma(x * w)$$

where the $*$ denotes a convolution operation. Consider two one-dimensional vectors $x \in \mathbb{R}^3$ and $w \in \mathbb{R}^3$; we will imagine these to be arrays of infinite length with all the entries at indices $[4, \infty)$ set to zero; this is known as zero-padding the input

$$\begin{aligned} x &= [2, -1, 1, 0, 0, \ldots] \\ w &= [1,\ 1, 2, 0, 0, \ldots]. \end{aligned}$$

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

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

$$(x * w)_k = \sum_{\tau=0}^{\infty} x_\tau \, w_{k-\tau}. \tag{5.1}$$

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

❓ Discuss the convolution of two square waves x, w.

$$(x * w)(t) = \int_0^t x(\tau)w(t-\tau)\,\mathrm{d}\tau.$$

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

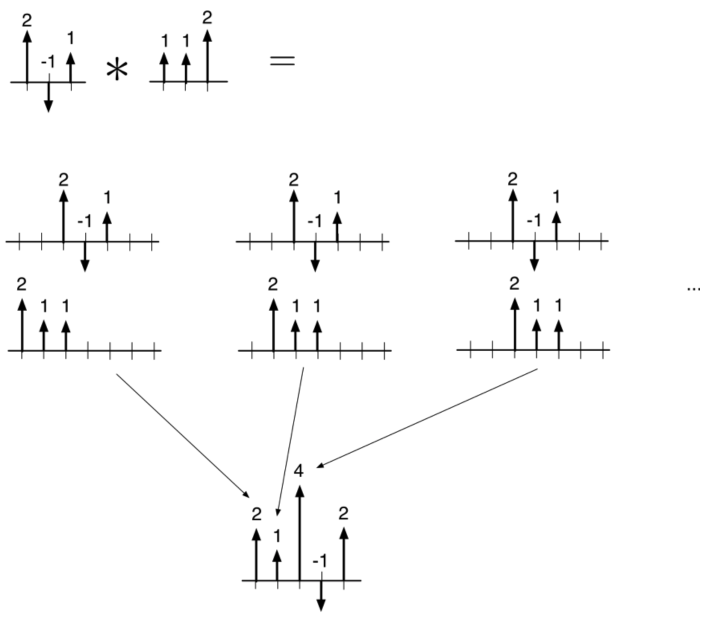


Figure 5.1: Flip and filter style computation of a convolution corresponding to the summation in (5.1).

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

$$\begin{aligned}(x * w)(t) &= \int_0^t x(\tau)w(t-\tau)\,\mathrm{d}\tau \\ &= -\int_t^0 x(t-s)\,w(s)\,\mathrm{d}s \\ &= \int_0^t w(s)\,x(t-s)\,\mathrm{d}s \\ &= (w * x)(t).\end{aligned}$$

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

$$(f + g) * h = (f * h) + (g * h)$$

for any integrable functions f, g, h.

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

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

$$(x * w)_k = \sum_{\tau=0}^{\infty} x_\tau \, w_{k+\tau}.$$

In simple words, the kernel w is not mirror flipped about the Y axis before computing the summation in (5.1). While such an operation is not strictly a convolution (you can see the difference if you consider an asymmetric kernel w, cross-correlation and convolution are the same for symmetric kernels), the difference does not matter for deep learning because the kernel w is learned during training. You can mirror flip the kernel after training and interpret the network as indeed performing a convolution with the flipped kernel.

5.1.1 Convolutions of 2D images

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

$$(x * w)_{i,j} = \sum_{s=0}^{\infty} \sum_{t=0}^{\infty} x_{s,t} \, w_{i-s,j-t}. \tag{5.2}$$

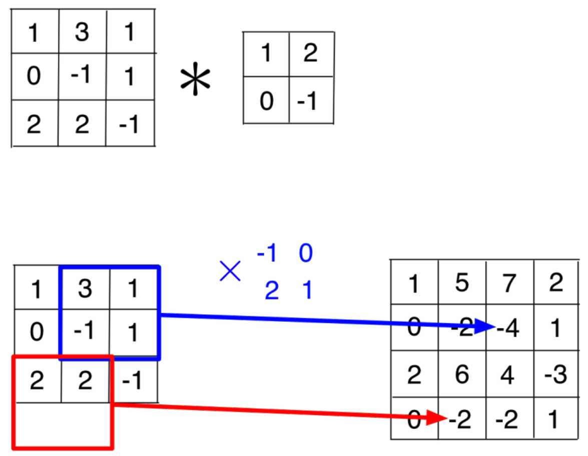


Figure 5.2: Flip and filter style computation of a convolution for a 2D input image corresponding to the summation in (5.2).

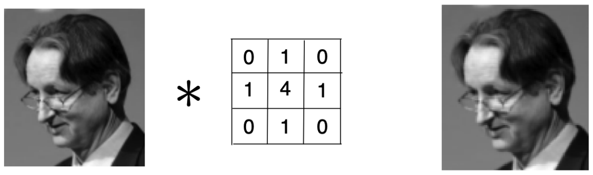
5.1.2 Some examples

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

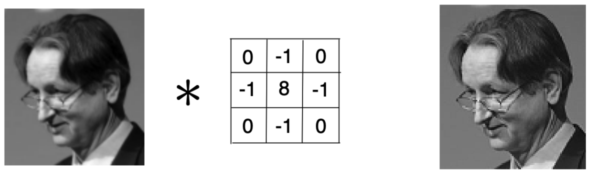
$$(2,-1,1) * (1,1,2) = \begin{bmatrix} 1 & & \\ 1 & 1 & \\ 2 & 1 & 1 \\ & 2 & 1 \\ & & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}.$$

The matrix is called a Toeplitz matrix https://en.wikipedia.org/wiki/Toeplitz_matrix. Two-dimensional convolutions can be written as a matrix-matrix multiplication using a similar construction; see https://stackoverflow.com/questions/16798888/2-d-convolution-as-a-matrix-matrix-multiplication.

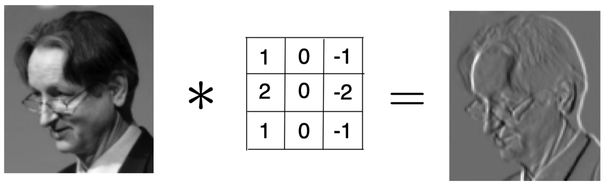
2. Lots of non-trivial transformations of the image are possible using slight changes in the weights. E.g., blurring



or sharpening using a slight change in the weights



We can also detect edges



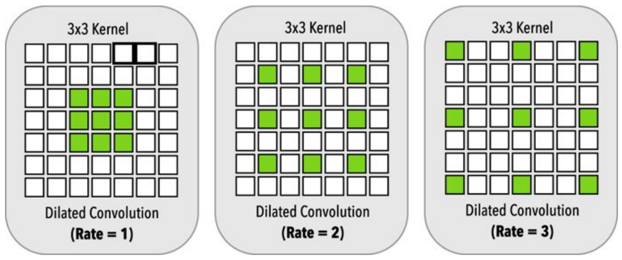
This filter is called the Sobel filter and is an integral part of image pre-processing pipelines in computer vision.

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

4. The operation $S^\top x$ has $S \in \mathbb{R}^{d \times p}$ weights and returns a vector in $\mathbb{R}^p$. A convolution operator returns a vector $(x * w) \in \mathbb{R}^d$ using K parameters in the kernel w. It is important to note that a lot of parameter sharing is happening while computing the values of the output neurons. You can find some animations at https://colah.github.io/posts/2014-07-Conv-Nets-Modular and https://colah.github.io/posts/2014-07-Understanding-Convolutions.

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

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



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

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

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

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

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

because it measures the gradient of the image intensity independently in the two directions. Separable convolutions are very useful when you can use high-dimensional data in deep learning, e.g., medical images out of MRI are 4-dimensional images (width, height, depth, channel).

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

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

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

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

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

5.2 How are convolutions implemented?

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

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

3. We can use the Fast Fourier Transform (FFT) to compute the convolution as
$$x * w = \mathcal{F}^{-1}\left[\mathcal{F}[x]\ \mathcal{F}[w]\right].$$
This is efficient for large kernels, say greater than 7×7.

Typically, deep learning libraries will choose an algorithm for convolution in *run-time* after looking at your neural architecture; you do not have to worry about the specific algorithm. A library called cuDNN from Nvidia implements a bunch of convolution algorithms on GPUs efficiently. PyTorch will pick one of these algorithms by checking how long it takes for the first forward-pass on your deep network. But the fact remains that large kernels which allow a larger receptive field (long-range correlations in the input image) are more expensive to compute than smaller kernels. Architectures such as Inception that we will see soon are an attempt to get a large receptive field while still keeping computations in the convolutional layer small.

⚠ You can set torch.cudnn.benchmark = False to stop this.

Remark 5.5 (Stride in convolutional layers). If you see the documentation for the convolutional layer in PyTorch at (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html) you will also see a parameter known as stride. Stride simply means that the output
$$(x * w)_k = \sum_{\tau=0}^{\infty} x_\tau w_{k-\tau}$$
is not computed at all values of k; if the stride is set to 2, the output is computed only at every alternate value of k. Note that the default stride as seen in the definition of convolution is 1. Since images change very little from pixel to pixel, this is a neat trick to reduce the redundancy of computing the convolution again and again over similar input. The important artifact of using a stride larger than 1 is that the output $(x * w)$ is no longer the same length (even after padding) as the input, is half the length if the stride is 2.

5.3 Convolutions for multi-channel images in a deep network

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

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

We will now study how the convolutional layer is implemented in a typical deep network. Let us denote the 2D convolution operation on a single-channel 2D image $A \in \mathbb{R}^{w \times h}$ by a kernel $w \in \mathbb{R}^{k \times k}$ by
$$A * w = B \in \mathbb{R}^{w \times h}.$$
Imagine that we have an RGB input image of size $w \times h$; the RGB indicates that there are three input channels, one for each color. The input to a convolutional layer in a deep network is therefore an array of size $3 \times w \times h$. Typical deep learning libraries, when they implement a convolutional layer with a kernel w of size $k \times k$, will output an image of size $c \times w \times h$ where c are the number of channels in the image at the output of the layer.

Effectively, a convolutional layer maps
$$\mathbb{R}^{3 \times w \times h} \ni A \mapsto B \in \mathbb{R}^{c \times w \times h}.$$

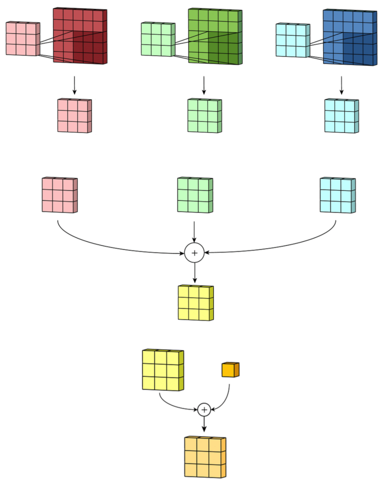


Figure 5.3: Convolutional layer in a typical deep network

The layer performs the operation

$$v_j + \sum_{i=1}^{3} A_i * w^{ij} = B_j$$

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

5.4 Translational equivariance using convolutions

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

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

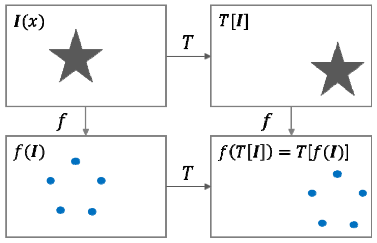
You will see from the definition of convolution in (5.1) that the convolution also gets translated

$$\begin{aligned}
(x' * w)_k &= \sum_{\tau=0}^{\infty} x'_\tau w_{k-\tau} \\
&= \sum_{\tau=0}^{\infty} x_{\tau-\Delta} w_{k-\tau} \\
&= \sum_{s=-\Delta}^{\infty} x_s w_{k-s-\Delta} \quad (s = \tau - \Delta) \\
&= (x * w)_{k-\Delta}.
\end{aligned} \tag{5.3}$$

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

$$(x' * w)_{k+\Delta} = (x * w)_k.$$

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



This property is called equivariance. Equivariance also holds for 2D convolutions. Equivariance to translations allows us to build an important property in a deep network. If we have a convolutional kernel that has weights such that the output is high for a certain object (star in adjoining picture, vertical/slanted strips in your Gabor filter homework), the output of a convolutional layer is such that the features also "move" if the input moves in the receptive field.

We can easily build a binary classifier using such equivariant features. If we want to build a star classifier, we simply check if some features in the output are large after convolution, e.g., we check if the largest feature in the 2D-feature map is greater than some pre-determined threshold

$$f(x, w) := \mathbf{1}_{\{\max_{ij}\{(x*w)_{ij}\} \geq \epsilon\}}. \tag{5.4}$$

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

5.5 Pooling to build translational invariance

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

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

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

In other words, we want features of a deep network to be *invariant* to translations in the input.

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

Pooling is an operation that smears out the features locally in the neighborhood of each pixel.

We can use our idea of setting all the weights to 1 to get what is called the average pooling operation. It is a linear operation and equivalent to convolving the input features using a kernel

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

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

Average pooling does not solve our problem of making the features invariant; the smeared out version simply moves less than Δ when the input translates by Δ. If we add many average pooling layers at various stages in a deep network, we make the features move even less and this may be sufficient to allow for weights v to be discriminative.

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

Max-pooling is another operation that builds invariance. It takes in an input $x \in \mathbb{R}^{w\times h}$ and computes

$$(\text{max-pool}(x))_{ij} = \max_{-k\leq s\leq k} \max_{-k\leq t\leq k} x_{i-s,j-t}. \tag{5.6}$$

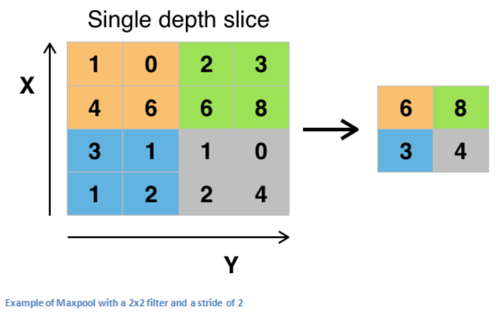


Figure 5.4: Max-pooling with a 2×2 kernel and a stride of 2 reduces the size of the input image by half. A stride of 1 would preserve the image size but would give less invariance.

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

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

Remark 5.6 (Max-pooling destroys information). As we see in Figure 5.4, max-pooling destroys a lot of information in the input image. The result of max-pooling is a much smaller feature map. This results in a large loss of information in the input data and often leads to a loss of discriminative power, i.e., accuracy, during training. This trade-off between building a classifier that is invariant to changes in the input and discriminative enough to distinguish between many different categories is fundamental.

❓ We have talked about invariance to translations in this lecture. Images taken from a fish-eye camera, or MRI images of the brain, are such that objects *rotate* in the field of view.

Can you think of a trick to build invariance to rotations?

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

Chapter 6

Data augmentation, Loss functions

Reading

1. Bishop Chapter 5.5.3, 4.3
2. Goodfellow Chapter 7.4

6.1 Data augmentation

In the previous chapter, we looked at convolutions as a way to reduce the parameters in a deep network, but more importantly as a way of building equivariance/invariance to translations. There are a lot of nuisances other than translation that do not have a group structure which precludes operations such as convolutions that we can perform to generate equivariance/invariance.

In this section, we will discuss techniques to build invariance to nuisances that are more complex than just translations, these techniques will seem brute-force but they also allow us to handle more complex nuisances. The main trick is to *augment* the data, i.e., create variants of each input datum in some simple way such that we *know* that its label is unchanged. If our original dataset is $D = \left\{(x^i, y^i)\right\}_{i=1,\ldots,n}$ we create an augmented dataset

$$D^T = \left\{(T(x^i), y^i)\right\}_{i=1,\ldots,n} \cup D. \tag{6.1}$$

where T is some operation of our choice. We have therefore expanded the number of samples in the training dataset to $2n$ instead of the original n. Effectively, data augmentation is a technique to create a dataset that is sampled from some other data distribution P than the original one.

6.1.1 Some basic data augmentation techniques

The most popular data augmentation techniques are setting T to be changes in brightness, contrast, cropping the image to simulate occlusions, flipping the image horizontally or vertically, jittering the pixels of the input image to simulate noise in the CCD of the camera/weather, padding the image which changes the borders of the input image, warping the image using a projection that simulates the same picture taken from a different viewpoint, thresholding the RGB color channels, zooming into an image to simulate changes in the scale etc.

You can see these operations at https://fastai1.fast.ai/vision.transform.html#List-of-transforms.

⚠ FastAI is a wrapper on top of PyTorch and is an excellent library to learn for doing your course projects.

6.1.2 How does augmentation help?

A number of such augmentations are applied to the input data while training a deep network. This increases the number of samples n we have for training but note that different samples share a lot of information, so the effective novel samples has not increased by much. Let us get an idea of when augmentation is useful and when it is not. Consider a regression and classification problem as shown below.

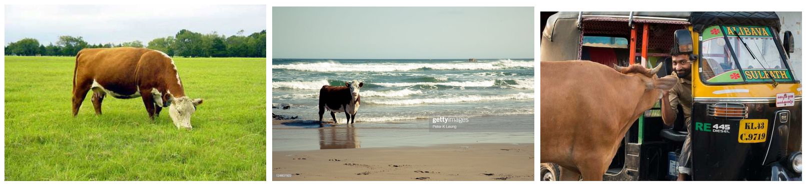


Figure 6.1: Cows live in many different parts of the world. A classifier that also uses background information to predict the category is likely to make mistakes when it is run in a different part of the world. Augmenting the input dataset on the left by replacing the background to include a mountain or a city is therefore a good idea if we want to run the classifier in a different part of the world. This will also force the classifier to *ignore* the background pixels when it classifies the cow, in other words the classifier is forced to become invariant to backgrounds by brute-force showing it different backgrounds.

In essence, data augmentation forces the model to tackle a larger dataset than our original dataset. The model is forced to learn what nuisances the designer would like it to be invariant to. Compare this to the previous chapter: by replacing fully-connected layers with convolutions and pooling we made the model invariant to translations. In principle, we could have trained a fully-connected deep network on a very large augmented dataset with translated objects. In principle, this would make the fully-connected network invariant to translations as well.

6.1.3 What kind of augmentation to use when?

In the example with regression, we saw that the regressor on the augmented data was essentially linear and had much less discriminative power than a polynomial regressor. This was of course by design, we chose to augment the data. If the test data for the problem came from the polynomial instead of our augmented distribution, the new classifier will perform poorly.

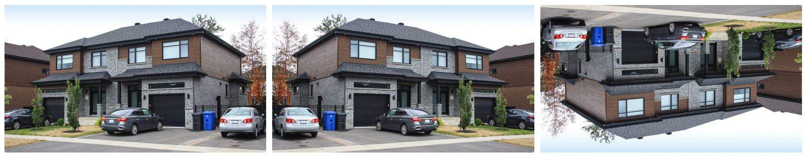


Figure 6.2: The second panel shows the original scene with a mirror flip (i.e., across the horizontal axis) while the third panel shows the original scene after a water reflection (i.e., flip across the vertical axis). The latter is an image that is very unlikely to occur in the real world, so it is not a good idea to use it for training the model.

> By being invariant to a larger set of nuisances than necessary, we are wasting the parameters of the model and risk getting a large error if the test data was not from the augmented distribution. By being invariant to a smaller set of nuisances than necessary, we are risking the situation that the test data will have some new nuisances which the classifier will perform poorly on. It is important to bear in mind that we do not always know what nuisances the model should be invariant to, the set of transformations in data augmentations necessarily depends—often critically—upon the application.

If you are building a classifier for detecting cars, motorbikes, people etc. for autonomous driving application, do you want to be the invariant to rotations?

Data augmentation requires a lot of domain expertise and often plays a huge role in the performance of a deep network. You should think about what kind of augmentations you will apply to data for speech processing, or for data from written text.

6.2 Loss functions

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

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

6.2.1 Regression

MSE loss. If the labels are real-valued $y^i \in \mathbb{R}$, e.g., we are predicting the price of housing in Boston given features of the houses (like you did in HW 0), we are solving a regression problem and the loss function to use for a deep network is also simply the regression loss.

$$\ell_{\text{mse}}(w) := \frac{1}{2} \left(f(x; w) - y \right)^2 \tag{6.2}$$

Recall, that we assume in machine learning that the training dataset contains independent and identically drawn samples. Real data often does not satisfy this iid assumption and a model fitted via regression may not work well if the data are correlated. A popular trick to handle such situations in regression to take a logarithmic transformation of the input, i.e., fit a model to $\log x$ using the loss

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

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

Huber loss. The square-residual loss in (6.2) works in most cases but it does not work well if there are outliers in the data. Outliers are data in the training set that are noisy or did not come from the true model. In such cases, we can use the Huber loss. If the residual is $r = f(x; w) - y$, the Huber loss is

$$\ell_{\text{huber}}(w; \delta) = \begin{cases} \frac{1}{2} \left| r \right|^2 & \text{if } |r| \leq \delta \\ \delta \left(|r| - \frac{1}{2}\delta \right) & \text{else.} \end{cases} \tag{6.3}$$

Observe that this does not penalize the model egregiously if the predictions are bad ($|r| \geq \delta$) for a particular datum. Doing so prevents the outliers from biasing the loss towards themselves and ruining the residuals for the other data.

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

$$\ell_{\text{mae}}(w) = |f(x; w) - y| \tag{6.4}$$

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

Using a subset-selection technique or the ℓ_{mae} loss leads to sparse weights w^*. This makes the model more interpretable than a model fitted using ℓ_{mse} loss. This is easy to understand for linear models: input dimensions corresponding to weights w_i^* that are zero do not take part in making predictions. So one may answer questions of the form "is variable x_i a relevant predictor of the target y".

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

Variable importance. For linear models, another way to answer the same question is to fit two models, one with w_i fixed to zero and all other weights fitted using the MSE loss (6.2) and another model without fixing w_i; the difference between the average square residuals in the two cases is a measure of how important the feature x_i is for the prediction. These techniques are called variable importance methods. We can also undertake the same program for nonlinear models on non-image based data.

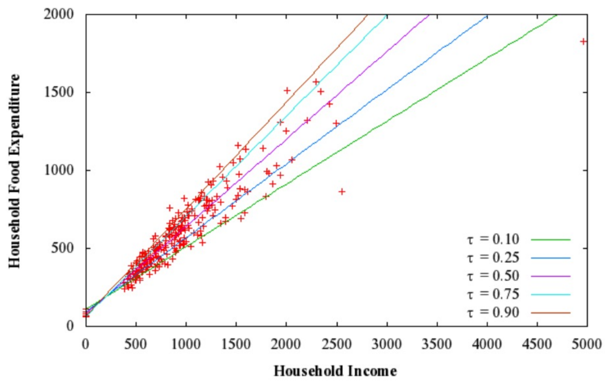
Quantile loss. The quantile loss is another simple trick to make the model more robust to outliers and get more information from the model than simply the prediction $f(x; w)$. Observe that if we have targets Y that are random variables with cumulative distribution function $F(y) = \mathbb{P}(Y \leq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

$$Q_Y(\tau) = F^{-1}(\tau) = \inf\{y : F(y) \geq \tau\}$$

for $\tau\ in(0,1)$. We now learn a predictor for $Q_Y(\tau) = f(x; w)$. It turns out (you can try to prove this) that this corresponds to the loss function

$$\begin{aligned}\ell_{\text{quantile}}(w; \tau) &= \begin{cases} r(\tau - 1) & \text{if } r < 0 \\ r\tau & \text{else.} \end{cases} \\ &= r\left(\tau - \mathbf{1}_{\{r<0\}}\right). \end{aligned} \tag{6.5}$$

where $r = y - f(x; w)$ is the residual. A standard technique is to fit multiple models using the quantile loss for different quantiles, say $\tau = 0.25, 0.5, 0.75$ and give multiple predictions of the target $f(x; w^\tau)$. A typical example of quantile linear regression looks as follows.



6.2.2 Classification: Cross-Entropy loss

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

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

One hot encoding.

An alternative representation of the targets in classification is so-called the *one-hot* encoding where y is transformed to

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

the vector e_y has a 1 at the y^{th} element and zeros everywhere else. The notation e_y denotes the y^{th} row of the identity matrix $I_{m\times m}$.

Predicting class probabilities.

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

$$f(x; w) = \underset{k}{\text{argmax}}\ \log p_w(k|x). \tag{6.6}$$

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

$$\mathbb{R}^m \ni \hat{y} = v^\top \sigma\left(S_L^\top \dots \sigma\left(S_2^\top\ \sigma(S_1^\top x)\right) \dots\right) \tag{6.7}$$

where $v \in \mathbb{R}^{p \times m}$. As we saw in Chapter 4, $\hat{y}$ are also called logits. Observe that the logits $\hat{y}$ are simply vectors in $\mathbb{R}^m$. How can we transform these logits to get $\log p_w(k|x)$ for all $k \in \{1, \dots, m\}$ as the output of the model?

Logistic loss.

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

$$\log \frac{p(1|x)}{p(0|x)} = \hat{y} = w^\top x. \tag{6.8}$$

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

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

Maximizing this probability (MLE) is the same as minimizing the log-probability

$$\begin{aligned}\ell_{\text{logistic}}(w) &:= -\log p_w(\{(x^1, y^1), \dots, (x^n, y^n)\}) \\ &= -\sum_{i=1}^n y^i p_w(1|x^i) + (1 - y^i) p_w(0|x^i)\end{aligned} \tag{6.9}$$

❓ We saw a different expression for the logistic loss in Chapter 3

$$\ell_{\text{logistic}}(w) = \log\left(1 + e^{-y\hat{y}}\right).$$

What is the difference?

In other words, the logistic loss is simply maximum-likelihood estimation for the model (6.8).

Binary Cross-Entropy loss.

Let us turn back to neural networks and multi-class classification. Imagine if each logit of a neural network in (6.7) acts independently, i.e., it predicts whether there is class k in this input or not without paying heed to what the other logits predict. This is not very prudent, for instance, if we know beforehand that there is only one object in the input image, then such a classifier is likely to have lots of false positives. Nevertheless, observe that this is exactly like running m independent binary logistic classifiers with the same feature $h^L \in \mathbb{R}^p$. We can write the loss for such a classifier succinctly as

$$\ell_{\text{bce}}(w) = -\sum_{k=1}^m \text{one-hot}(y)_k \log p_w(k|x). \tag{6.10}$$

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

6.2.3 Softmax Layer

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

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

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

$$\begin{aligned} \log p_w(k|x) &\propto \hat{y}_k \\ \Rightarrow p_w(k|x) &= \frac{e^{\hat{y}_k/T}}{\sum_{k'=1}^{m} e^{\hat{y}_{k'}/T}}. \end{aligned} \tag{6.11}$$

The result $p_w(k|x)$ is a valid distribution on k because it sums up to 1. This operation, namely taking the logits $\hat{y}$ and constructing a probabilities out of them is called as a softmax operator. The constant T in (6.11) is called the temperature. A large value of T results in a smoother probability distribution $p_w(k|x)$ because the individual values of the logits matter less. A small value of T results in a very large weight due to the exponent on the largest logit and the distribution $p_w(k|x)$ is therefore highly spiked. The temperature is set to 1 by default in PyTorch.

⚠ You will often see people calling

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

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

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

if one of the entires of $\hat{y}$ is much larger than the others, or if $T \to 0$. We will however use the word "softmax" to refer to the operation of transforming $\hat{y}$ into $p_w(k|x)$ because we do not have any need for this softened version of the max operator.

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

$$\begin{aligned} \ell_{\text{ce}}(w) &= -\sum_{k=1}^{m} \text{one-hot}(y)_k \log p_w(k|x) \\ &= -\frac{\hat{y}_y}{T} + \log\left(\sum_{k'=1}^{m} e^{\hat{y}_{k'}/T}\right). \end{aligned} \tag{6.12}$$

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

6.2.4 Label smoothing

The correct logit in (6.12) is encouraged to go to $+\infty$ while the incorrect logits are encouraged to go to $-\infty$. This can lead to dramatic over-fitting when the number of classes m is very large. Label smoothing is a trick that alleviates the problem: instead of using a one-hot encoding of the true label y, it uses the encoding

$$\text{label-smoothing}(y)_k = \begin{cases} 1-\epsilon & \text{if } k = y, \\ \frac{\epsilon}{m-1} & \text{else.} \end{cases} \tag{6.13}$$

The cross-entropy loss with this new encoding is now

$$\begin{aligned} \ell_{\text{label-smoothing-ce}}(w) &= -\sum_{k=1}^{m} \text{label-smoothing}(y)_k \log p_w(k|x) \\ &= -(1-\epsilon) \log p_w(y|x) - \frac{\epsilon}{m-1} \sum_{k \neq y} \log p_w(k|x) \end{aligned} \tag{6.14}$$

If you take the derivative of this loss with respect to $\hat{y}$ you will see that the value of $\hat{y}$ that minimizes the loss is

$$\hat{y}_k^* = \begin{cases} \log\left((m-1)(1-\epsilon)/\epsilon\right) + \alpha & \text{if } k = y \\ \alpha & \text{else.} \end{cases} \tag{6.15}$$

where α is an arbitrary real number. Notice that logits for both the correct and the incorrect classes are finite in this case, they no longer blow up to infinity.

6.2.5 Multiple ground-truth classes

If there are multiple classes that are all present in the input image, i.e., if the ground truth data has multiple labels, we can easily use the vector

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

for all the present classes k and set

$$\ell_{\text{bce}}(w) = -\sum_{k=1}^{m} \text{multi-hot}(y)_k \log p_w(k|x) \tag{6.16}$$

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

Chapter 7

Bias-Variance Trade-off, Dropout, Batch-Normalization

Reading

1. Bishop 1.3, 3.2, 14.2-14.3
2. Goodfellow 5.1-5.4, 7.1-7.3
3. Dropout Srivastava et al. (2014)
4. Batch-Normalization Ioffe and Szegedy (2015)

In this chapter, we will take our first look at how machine learning classifiers generalize to new data. We will first discuss the so-called Bias-Variance Tradeoff which indicates that the variance of the predictions of a model can be reduced by increasing its bias. Regularization is a technique to give us control over this tradeoff. We will then see a few popular regularization techniques, in particular two that are important in deep learning called Dropout and Batch-Normalization.

7.1 Bias-Variance Decomposition

Ideally, we want a classifier that accurately captures the regularity in the data but also works well for unseen data. Turns out this is typically impossible to both simultaneously. We will introduce this using regression.

Given our dataset $D = \{(x^i, y^i)\}_{i=1,\ldots,n}$ we fit a model $f(x; w) \in \mathcal{F}$ where $\mathcal{F}$ is some class of models, say all neural networks with a given architecture; we will keep the dependence of f on w implicit in this section because we don't need it. We use a loss $\ell(f(x), y) = |f(x) - y|^2$ to fit this model by minimizing

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n} |f(x^i) - y^i|^2 \tag{7.1}$$

This is of course the training loss, also called the empirical risk. A classifier that minimizes $\hat{R}(f)$ works well on the training data. If we want to measure how well a model works on new data from the distribution P we are interested in the the *population risk*

$$\begin{aligned} R(f) &= \int |f(x) - y|^2 \, P(x, y) \, \mathrm{d}x \, \mathrm{d}y \\ &= \mathop{\mathbb{E}}_{x}\left[\int |f(x) - y|^2 \, P(y|x) \, \mathrm{d}y\right]. \end{aligned} \tag{7.2}$$

It turns out that because the loss is quadratic, we can write down the minimizer of the population risk, formally, as

$$f^*(x) = \mathop{\mathbb{E}}_{y}[y|x]. \tag{7.3}$$

In other words, the optimal regressor is the conditional expectation of the targets y given a datum x. Since we do not know the data distribution P, we cannot compute the model f^*. We now compare some regression f that we may have obtained by minimizing (7.1) with the optimal f^*.

Observe that

$$\begin{aligned} (f(x) - y)^2 &= (f(x) - f^*(x) + f^*(x) - y)^2 \\ &= (f(x) - f^*(x))^2 + 2\,(f(x) - f^*(x))\,(f^*(x) - y) + (f^*(x) - y)^2. \end{aligned}$$

Substitute this expression in (7.2) to get

$$R(f) = \mathop{\mathbb{E}}_{x}\left[\int (f(x) - f^*(x))^2\right] + \mathop{\mathbb{E}}_{(x,y)\sim P}\left[(f^*(x) - y)^2\right] \tag{7.4}$$

Observe that the cross-term

$$\mathop{\mathbb{E}}_{x}\left[\int 2(f - f^*)(f^* - y)P(y|x) \, \mathrm{d}y\right] = 0$$

vanishes because $f^*(x) = \mathbb{E}[y|x] = \int yP(y|x)\mathrm{d}y$. In the first term, there is no y because the distribution $P(y|x)$ when integrated with respect to y is 1. The decomposition in (7.4) is insightful. The first term tells us how far our model $f(x)$ is from the optimal $f^*(x)$. The second term tells us how much the optimal model itself is from the data (x, y). The second term is not under our control because it does not depend on $f(x)$ at all. This term

$$\text{Bayes error} = \mathop{\mathbb{E}}_{(x,y)\sim P}\left[(f^*(x) - y)^2\right]. \tag{7.5}$$

⚠ You can think of the Bayes error as being non-zero if the sensor used to measure y is noisy, there is no way we can get deterministic data in that case. If on the other hand the sensor is perfect, e.g., a large number of humans are annotating data very carefully like we often do in modern machine learning, the Bayes error is essentially zero.

is irreducible error of any classifier f. It is only zero if the data (x, y) is coming from a deterministic source, i.e., there is no noise in the true targets y created by Nature and Nature's model (it is important to realize that this model is *not* f^*) is deterministic.

We will now investigate the first term better. Notice that the model f is created using a finite dataset. Let us emphasize it as

$$f(x; D)$$

and rewrite the first term in (7.4) as

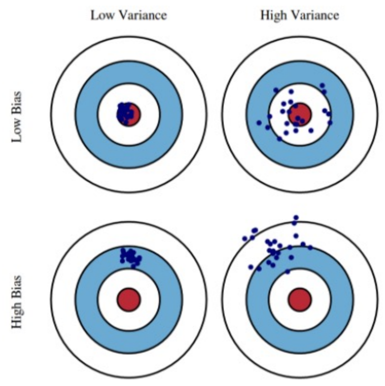
$$\begin{aligned}(f(x;D) - f^*(x))^2 &= \Big(f(x;D) - \underset{D}{\mathbb{E}}[f(x;D)] + \underset{D}{\mathbb{E}}[f(x;D)] - f^*(x)\Big)^2 \\ &= \Big(f(x;D) - \underset{D}{\mathbb{E}}[f(x;D)]\Big)^2 \\ &+ \Big(\underset{D}{\mathbb{E}}[f(x;D)] - f^*(x)\Big)^2 \\ &+ 2\Big(f(x;D) - \underset{D}{\mathbb{E}}[f(x;D)]\Big)\ \Big(\underset{D}{\mathbb{E}}[f(x;D)] - f^*(x)\Big).\end{aligned}$$

Recall that the dataset is a random variable as well, it is a bunch of draws from the joint distribution P. Effectively, $f(x;D)$ which is our fitted model is a random variable that depends on the randomness of D. We now take the expectation over the *dataset* D on both sides of this equation.

$$\underset{D}{\mathbb{E}}\Big[(f(x;D) - f^*(x))^2\Big] = \underbrace{\underset{D}{\mathbb{E}}\Big[\Big(\underset{D}{\mathbb{E}}[f(x;D)] - f^*(x)\Big)^2\Big]}_{(\text{bias})^2} + \underbrace{\underset{D}{\mathbb{E}}\Big[\Big(f(x;D) - \underset{D}{\mathbb{E}}[f(x;D)]\Big)^2\Big]}_{\text{variance}}. \tag{7.6}$$

The cross-term again vanishes when we take the expectation over the dataset. The first term is called the squared bias: it is the gap between the predictions of our model compared to the optimal model f^* created across many experiments each with a different dataset D. The second term is the variance and it measures how sensitive the model $f(x;D)$ to getting a particular dataset D to train on, if it is very sensitive a model fitted on D does not work well on most others datasets and consequently the variance is large. We will parse these quantities further soon.

⚠ Here is a good mnemonic to remember. Imagine the center of the bull's eye as the optimal classifier f^* and our darts as the model $f(x;D)$.



We have therefore shown that

$$R(f) = (\text{bias})^2 + \text{variance} + \text{Bayes error} \tag{7.7}$$

Recall that we want to minimize the population risk $R(f)$. We cannot do much about the Bayes error. If the model $f(x;D)$ is large and is fitted very well

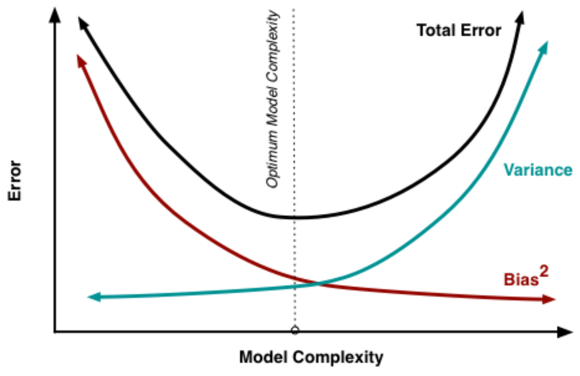


Figure 7.1: Population risk as a function of model capacity

on the dataset D, i.e., if its predictions match true y (notice that the optimal models predictions f^* are also close to y), the gap between the predictions of the fitted model and the optimal model is small on the dataset D. In other words, if our model is large we will have a small bias. The bias of a model decreases as we consider larger models $f(x; D)$. If our dataset is small, the model $f(x; D)$ is likely to have a large variance because it has not seen a large amount of data. The effect increases for larger models because they may use a larger number of nuisances i.e., features that are not relevant to prediction of targets. We call this over-fitting.

If we plot a picture of how the bias and variance change as model capacity (you can think of capacity simply as the number of parameters in a model for now) increases, we see a famous U-shaped curve for the sum of squared bias and variance shown in Figure 7.1. Given a dataset D we should pick a model that lies at the bottom of this curve to get a good population risk; this model makes a good tradeoff between bias and variance.

The caveat is that we do not have access to a lot of different datasets to measure the bias or the variance. This is why the bias-variance trade-off, although fundamental in machine learning/statistics and a great thinking tool, is of limited direct practical value.

Bias-variance tradeoff for classification

We have only talked about the bias-variance trade-off for regression. The development for classification is not very different and same principles hold. We first define an optimal classifier

$$f^*(x) = \operatorname*{argmin}_{f \in \mathcal{F}} \mathop{\mathbb{E}}_{(x,y)\sim P} [\ell(y, f(x))]$$

for a loss function ℓ. The bias, variance of a given classifier $f(x; D)$ relative to this optimal classifier and the Bayes error are given by

$$\begin{aligned}
\text{bias} &= \mathop{\mathbb{E}}_{x} [\ell(f^*(x), f(x; D))] \\
\text{variance} &= \mathop{\mathbb{E}}_{D} [\ell(f(x; D), f^{\text{avg}}(x))] \\
\text{Bayes error} &= \mathop{\mathbb{E}}_{(x,y)\sim P} [\ell(y, f^*(x))] .
\end{aligned} \tag{7.8}$$

where $f^{\text{avg}}(x) = \operatorname{argmin}_f \mathbb{E}_D [\ell(y, f(x))]$; under the MSE loss this is the average of predictions of regressors on different datasets, for the MAE loss this is the median of the predictions of models trained on different datasets, for the zero-one loss it is the most frequent prediction of models trained on different datasets. We again have a trade-off that is obtained by decomposing the population risk

$$\mathop{\mathbb{E}}_{(x,y)\sim P} \left[\mathop{\mathbb{E}}_{D} [\ell(y, f(x; D))] \right] = \text{bias } + c_2 \text{variance } + c_1 \text{Bayes error.}$$

where c_1, c_2 are constants. You can read more about this in Pedro (2000).

Double-descent

The surprising thing is that for deep networks, we do not see this classical bias-variance trade-off. The population risk looks like

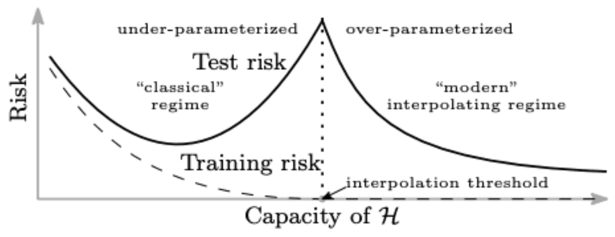


Figure 7.2: Double-descent curve: the validation error of deep networks decreases even if more and more complex models are fitted on the same data; there is no apparent over-fitting and growth in the variance of the classifier.

in what is now called the "double-descent" curve. The population risk of deep networks keeps decreasing even if we fit very large models on relatively small datasets, e.g., CIFAR-10 has 50,000 images, the model you will fit in HW 2 has about 1.6M weights and is considered a very small model by today's standards. We will see some heuristic derivation into why the population risk may look like this for deep networks but understanding this phenomenon which goes flat against established knowledge in machine learning is one of the big open problems in the study of deep networks today.

7.1.1 Cross-Validation

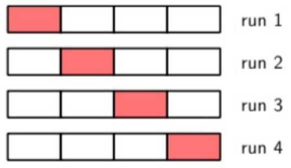
We have seen that the bias-variance trade-off requires us to consider multiple datasets. In practice, we only have *one* dataset that we collected by running an experiment. If this data is large, we can split it into two three parts

$$\text{data } = \text{training set } \cup \text{validation set } \cup \text{test set}.$$

The validation set is used to compare multiple models that we fit on the training set and pick the best performing one. This model is then run on the test set to demonstrate how well we have learned the data. The test set is necessary because across your design efforts to fit different models, you will evaluate on the validation set multiple times and this may lead to over-fitting on the validation set.

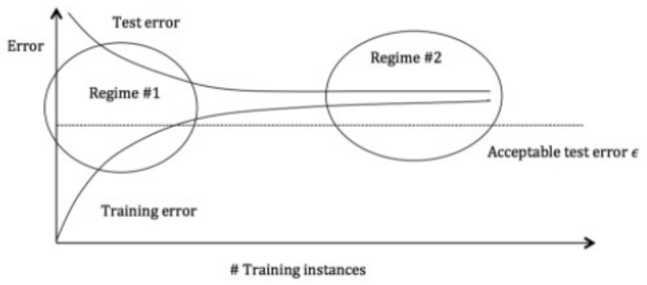
If the available data is not a lot, we want to use as much of the data as possible for training. If however only use a small fixed validation set for comparing models, we risk making mistakes in our choices. Cross-validation is a solution to this problem: it trains k different models, each time a fraction $(k-1)/k$ of the data is used as the training set and the remainder is used as the validation set. The validation performance of k models obtained by this process is averaged and used as a score to evaluate a particular model design (architecture, hyper-parameters etc).

⚠ 4-fold cross-validation.



Some practical tips

It is useful to think of the bias-variance trade-off when you fit deep networks in practice. If the training or test error is high, there are a number of ways to improve performance using the bias-variance tradeoff as a thinking tool.



In the first regime on the left, we have high validation error across cross-validation folds and low training error. This indicates that we have a high variance in the bias-variance trade-off. Typical techniques to counter this is to use a smaller model, get more data, or bagging a set of models together (will cover this in Section 7.3). In the second regime on the right, if the test error *and* the training error are close to each other but both are large, the model is likely to have high bias. In these cases, we should fit a more complex model (say increase the number of weights, or pick a different architecture), add more features to the training data (in the non-deep-learning setting) to give our model more discriminative features to use, or use boosting (we will cover this in Section 7.3).

Cautionary Tale

You will however notice that a lot of research papers in deep learning simply use validation data as test data. Their reasons for doing so are as follows. All researchers have the same large dataset from which they would create a potential test set, the researchers therefore also know the ground-truth labels of test images and it is difficult to trust them not to peek at the ground-truth labels to choose between models. If the test data is hidden from everyone, we need a centralized server for evaluating everyone's results. This is difficult because research is fundamentally about discovering new knowledge. Kaggle competitions or the ImageNet Challenge http://image-net.org/challenges/LSVRC are few instances where such a centralized server is available.

It is therefore debatable whether the current practice of using validation set as the test set should be considered valid. On the positive side, it makes results across different publications comparable to each other; if everyone reports the error of their model on the same validation set, it is easy to compare Algorithm A versus Algorithm B. On the negative side, this incentivizes extensive hyper-parameter tuning and risks results that are over-fitted on the validation data, e.g., new fields such as neural architecture search are particularly problematic in this context. This is also the main reason for the current "style of research" where folks judges the merit of machine learning research simply by checking whether Algorithm A gets better validation error than Algorithm B on standard datasets. This is not the correct way to do scientific research. The more appropriate way to instantiate the scientific method is to first formulate a hypothesis, e.g., is gene X correlated with cancer Y, then collect data that allows us to evaluate such an hypothesis and undertake appropriate statistical precautions report whether the hypothesis stands/does not stand.

That said, there are researchers who have evaluated others' claims (obtained using validation data, namely A better than B) on independent test data and reached similar conclusions, see for example https://arxiv.org/abs/1902.10811, so the evaluation methodology is broken but the progress is real.

7.2 Weight Decay

The set of models with smaller complexity are a subset of the set of models with larger complexity, e.g., if you are fitting a polynomial regression, you can consider the subset of models with coefficients of the higher-order terms equal to zero and have thus created the set linear regressors. Effectively, the space of *models* looks as follows.

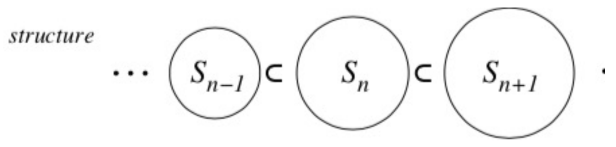


Figure 7.3: A cartoon of the space of models. The n in the picture refers to number of parameters in the model, not the number of data.

Let's say we are fitting a class of models with large complexity and are unsure whether the variance in the bias-variance trade-off will be large. We can either collect more data, or we can modify the loss function to encourage the training process to pick models of lower complexity.

Restricting the space of models that the training process searchers over to fit the data is called *regularization*. We will denote regularizers by

$$\text{regularizer} = \Omega(w)$$

and modify our loss function for fitting data to be

$$\ell'(w; x, y) := \ell(w; x, y) + \Omega(w).$$

Weight decay is one of the simplest regularization techniques and uses

$$\Omega(w) = \frac{\alpha}{2} \|w\|_2^2. \tag{7.9}$$

This is more widely known as ℓ_2 regularization because we use the ℓ_2 norm of the weights as the regularizer. It is also called Tikonov regularization, a name that comes from the literature on partial differential equations. The name weight decay comes from the neural networks literature of the 1980s. The gradient of the modified loss is

$$\nabla \ell'(w; x, y) = \nabla \ell(w; x, y) + \alpha\, w,$$

which gives

$$w^{t+1} = (1 - \eta\, \alpha) w^t - \eta \nabla \ell(w^t; x, y);$$

where η is the learning rate. In other words the weights w are encouraged to become smaller in magnitude when SGD takes a step using the negative gradient.

If we have a linear regression problem with $f(x;w) = w^\top x$ and X, Y are the matrices for the data and targets respectively, the regularized objective is

$$\frac{1}{2}\|Y - Xw\|_2^2 + \frac{\alpha}{2}\|w\|^2$$

and you can compute the minimizer by taking derivatives and setting them to zero to be

$$w^* = \left(X^\top X + \alpha I\right)^{-1} X^\top Y.$$

In other words, weight decay for linear regression adds elements to the diagonal of the data covariance matrix $X^\top X$. This results in a smaller inverse and thereby a smaller magnitude of w^*. Notice that if the covariance matrix is rank deficient, the regularized matrix is no longer rank deficient. If the covariance matrix has a large condition number (ratio of the largest and smaller eigenvalue), which makes taking the inverse numerically difficult, the regularized matrix has a better condition number.

7.2.1 Do not do weight decay on biases

If the input data and targets in linear regression are centered we do not need a bias parameter in our model. Notice however that if the dataset is not centered, the bias parameter is essential. Should we perform weight decay on the bias parameter in this case? The weight decay penalty prevents the bias parameter to adapt to the non-zero mean of the data. This is also important to keep in mind while training neural networks. We should not impose weight decay regularization on the bias parameters of the convolutional and fully-connected layers.

⚠ Weight decay is closely related to other norm-based penalties, e.g., ℓ_1 regularization sets

$$\Omega_{\ell_1}(w) = \alpha\|w\|_1.$$

As we discussed briefly in Chapter 6, such a regularizer encourages the weights to become sparse. Sparsity penalties are very common in the signal processing literature (e.g., compressed sensing, phase retrieval problems) but they are less common in the deep learning literature.

7.2.2 Maximum a posteriori (MAP) Estimation

MAP estimation gives a Bayesian perspective to regularization in machine learning. In maximum likelihood (ML) estimation, we were interested in solving for weights that maximize the likelihood of the observed data:

$$w^*_{\text{MLE}} = \underset{w}{\text{argmin}} -\frac{1}{n}\sum_{i=1}^{n} \log p_w(y^i|x^i; w).$$

MAP estimation enforces some prior knowledge we may have about the weights w. In Bayesian statistics, such prior knowledge is represented as a probability distribution, known as the *prior*, on the parameters w *before we see any data in the training process*, i.e., *a priori probability*

$$\text{prior} = p(w)$$

MAP estimation is regularized ML estimation. Given a prior distribution, we can use Bayes law to find the *posterior distribution* on the weights after observing the data

$$p(w|D) = \frac{p(D|w)\, p(w)}{p(D)} \tag{7.10}$$

Remember that the left hand side is a legitimate probability distribution with the denominator given by

$$Z := p(D) = \int p(D|w)\, p(w)\, \mathrm{d}w.$$

The denominator Z called the "evidence" or the partition function lies at the heart of all statistics, we will see why in Module 4.

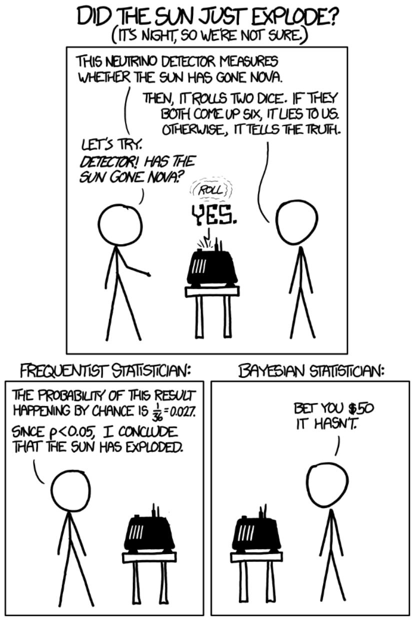
MAP estimation finds the weights that maximize this *a posteriori* probability

$$\begin{aligned} w^*_{\text{MAP}} &= \underset{w}{\operatorname{argmax}} \left\{ \log p(D; w) + \log p(w) \right\} \\ &= -\frac{1}{n} \sum_{i=1}^{n} \log p_w(y^i | x^i; w) + \Omega(w) + \log Z(D) \\ &= -\frac{1}{n} \sum_{i=1}^{n} \log p_w(y^i | x^i; w) + \Omega(w). \end{aligned} \tag{7.11}$$

In the second step, we have denoted the log-prior by Ω

$$\log \text{prior}(w) := \Omega(w).$$

The final step follows because $Z(D)$ is not a function of the weights w and can therefore can be ignored in the optimization.


DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY. DETECTOR! HAS THE SUN GONE NOVA?
(ROLL)
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027. SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Frequentist vs. Bayesian point of view

This section was our first view into Bayesian probabilities, as opposed to frequentist methods where we estimate probabilities by counting how many times a certain event occurs across our experiments. Frequentist probabilities are not designed to handle all situations. For instance we may be interested in estimating the probability of a very unlikely event, say that of the sun going supernova. This event has of course not happened yet and a frequentist notion of probability where we repeat the experiment many times and estimate the probability as the fraction of times the event occurs is not appropriate. The Bayesian point of view provides a natural way to answer these questions and the key idea is to encode our belief that the sun cannot go supernova as a prior probability.

An alternate way to think about this is that the weights w of a model are considered a fixed quantity that we are supposed to estimate in a frequentist setting. The likelihood $p(D; w)$ is used to compare different models w and if one wanted an estimate of how much error we are making in our estimate, we would compute the variance in the Bias-variance tradeoff namely, the variance of our estimate across different draws of the dataset D. In the Bayesian point of view, there is a single dataset D and the uncertainty of our estimate of w^* would be expressed as the variance of the posterior distribution $p(w|D)$ in Bayes law.

Weight decay regularization is MAP estimation with Gaussian prior

Weight decay can be seen as using a Gaussian prior

$$p_{\text{weight-decay}}(w) \propto e^{-\frac{\|w\|_2^2}{(2\alpha^{-1})}}.$$

This is a multi-variate Gaussian distribution with mean zero and a diagonal covariance matrix with α^{-1} on the diagonal. The denominator is a function of α^{-1} and we do not need to worry about it while performing MAP estimation because it does not depend on w.

In other words, we have seen that weight decay in the training objective can be thought of as a MAP estimation using a Gaussian prior instead of ML estimation.

The Gaussian prior captures our a priori estimate of the true weights: the probability of the weights w being large is low (it is distributed as a Gaussian/Normal distribution). The likelihood term fits the weights to the data but instead of relying completely on the data which may result in a large variance (in cases when data is few), we also rely on the prior while fitting the model. This reasoning is captured in Bayes law.

Similarly, a sparsity penalty is MAP estimation with a Laplace prior For scalar random variables, the Laplace distribution is given by

$$p(w) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}.$$

If we have

$$\Omega(w) = \|w\|_1$$

we can see that regularized ML, i.e., MAP estimation corresponds to using a Laplace prior on the weights w.

7.3 Dropout

We will next look at a very peculiar regularization technique that is unique to deep networks. Consider a two-layer network given by

$$\hat{y} = v^\top \text{dropout}\left(\sigma\left(S^\top x\right)\right).$$

Dropout is an operation that is defined as

$$\text{dropout}_{1-p}(h) = h \odot r \tag{7.12}$$

⚠ It is important to remember that a new dropout mask r is chosen for every input in the mini-batch.

where $r \in \{0,1\}^p$ is a binary mask and the notation $\odot$ denotes element multiplication. Each element of this mask r_k is a Bernoulli random variable with probability $1-p$

$$r_k = \begin{cases} 0 & \text{with probability } p \\ 1 & \text{with probability } 1-p. \end{cases}$$

In simple words, dropout takes the input activations h and zeros out a random subset of these; on an average p fraction of the activations are set to zero and the rest are kept to their original values. In pictures, it looks as follows.

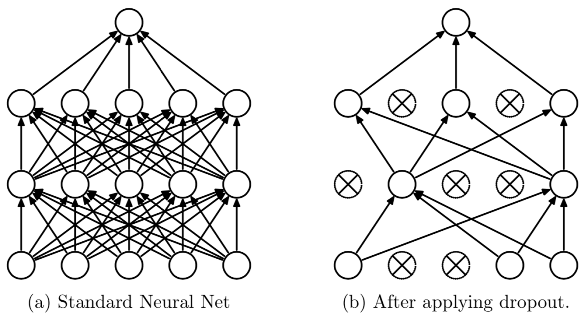


Figure 7.4: Dropout picks a random sparse subnetwork of a large deep network using the mask.

❓ The dropout mask is chosen at random for each image. Let us imagine that we have one dropout layer after every fully-connected layer. For the network shown in the figure with two hidden layers and 5 neurons at each layer, how many distinct sparse networks could be chosen for each input if $p = 0.5$?

The default Dropout probability is $p = 0.5$ in PyTorch, i.e., about half of the activations are set to zero for each input. Although you will see a lot of online code and architectures with this default value, you should experiment with the value of p, different values often given drastically different training and validation errors.

7.3.1 Bagging classifiers

Bagging, which is short for *bootstrap aggregation*, can be explained using a simple experiment. Suppose we wanted to estimate the average height μ of people in the world. We can measure the height of N individuals and obtain *one* estimate of the mean μ. This is of course unsatisfying because we know that our answer is unlikely to be the mean of the entire population. Bootstrapping computes multiple estimates of the mean μ_k over many *subsets* of the data N and reports the answer as

$$\mu := \text{mean}(\mu_k) + \text{stddev}(\mu_k).$$

Each subset of the data is created by sampling the original data with N samples *with replacement*. This is among the most influential ideas in statistics (Efron, 1992) because it is a very simple and general procedure to obtain the uncertainty of the estimate.

Effectively, the standard deviation of our new bootstrapped estimate of the mean is simply the standard deviation in the Bias-Variance trade-off with the big difference that we created multiple datasets D by sub-sampling with replacement of the original dataset.

Bagging is a classical technique in machine learning (Breiman, 1996) that trains multiple predictive models $f(x; w^k)$ for $k \in \{1, \dots, M\}$, one each for bootstrapped versions of the training dataset $\{D^1, \dots, D^M\}$. We aggregate the outputs of all these models together to form a *committee*

$$f(x; w^1, \dots, w^M) = \frac{1}{M} \sum_{k=1}^{M} f(x; w^k).$$

You can see that this procedure reduces the sum of the squared-bias and variance of the model (the first term in (7.4)) in the bias-variance trade-off by a factor of M if the errors with respect to the optimal classifier f^* of all the models $\{w^k\}$ are zero-mean and uncorrelated. In other words, the average error of a model can be reduced by a factor of M by simply averaging M versions of the model.

Bagging is always a good idea to keep in your mind. The winners of most high-profile machine learning competitions, e.g., the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize) or the ImageNet challenge, have been bagged classifiers created by fitting multiple architectures on the same dataset. Even today, random forests are among the most popular algorithms in the industry; these are ensembles of hundreds of models called decision trees on bootstrapped versions of data. A lot of times, if we are combining diverse architectures in to the committee, we do not even need to bootstrap the data. Bagging does not work when the errors of the different models are not uncorrelated; this is however easy to fix by censoring out features in addition to boostrapping like it is done while training a random forests.

7.3.2 Some insight into how dropout works

Consider the following, very heuristic but nevertheless beautiful, argument in the original paper on dropout (Srivastava et al., 2014).

We will remove the nonlinearities and consider only a single layer linear model with dropout directly applied to the input layer $f(x; v) = v^\top \text{dropout}(x)$. Linear regression minimize the objective $\|y - Xw\|_2^2$ and similarly the dropout version of linear regression for our model would minimize

$$\min_w \mathop{\mathbb{E}}_R \left[\|y - (R \odot X)w\|_2^2\right] \tag{7.13}$$

where each row of the matrix R consist of the dropout mask for the i^{th} row x^i of the data matrix X. Think carefully about the expectation over R on the outside, since we choose a random dropout mask each time an input is presented to SGD, the correct way to write dropout is using this expectation over the masks. Each entry of R is a Bernoulli random variable with probability $1 - p$ of being 1. Note that

$$\mathop{\mathbb{E}}_R [R \odot X] = (1-p)X$$

and the $(ij)^{\text{th}}$ element is

$$\left(\mathop{\mathbb{E}}_R \left[(R \odot X)^\top (R \odot X)\right]\right)_{ij} = \begin{cases} (1-p)^2 \left(X^\top X\right)_{ij} & \text{if } i \neq j \\ (1-p) \left(X^\top X\right)_{ii} & \text{else.} \end{cases}$$

We can use these two expressions to compute the objective in (7.13) to be

$$\mathop{\mathbb{E}}_R [\|y - (R \odot X)w\|_2] = \|y - (1-p)Xw\|^2 + \underbrace{p(1-p)w^\top \text{diag}(X^\top X)w}_{\Omega(w)}.$$

In other words, for linear regression, dropout is equivalent to weight-decay where the coefficient α in (7.9) depends on the diagonal of the data covariance and is different for different weights. If a particular data dimension varies a lot, i.e., $(X^\top X)_{ii}$ is large, dropout tries to squeeze its weight to zero. We can also absorb the factor of $1 - p$ into the weights w to get

$$\mathop{\mathbb{E}}_R [\|y - (R \odot X)w\|_2] = \|y - X\tilde{w}\|^2 + \underbrace{\left(\frac{p}{1-p}\right) \tilde{w}^\top \text{diag}(X^\top X)\tilde{w}}_{\Omega(w)} \tag{7.14}$$

where $\tilde{w} = (1-p)w$. This makes the regularization more explicit, if $p \approx 0$, most activations are retained by the mask and regularization is small.

Next, bagging provides a very intuitive understanding of how dropout works in a deep network at test time. We now write out the classifier explicitly as

$$f(x; w, r^k) = \sum_{i=1}^{d} w_i \left(x_i \odot r_i^k \right);$$

note that the mask r^k is not a parameter of the model, we have simply chosen to make it more explicit for the sequel. We now imagine each mask as creating a *bootstrapped* version of the model; different masks r^k give different classifiers even if the weights v and the input x is the same for all.

⚠ Training with dropout is equivalent to introducing weight decay on the weights. Remember however that this argument is only rigorous for linear regression models (the derivation essentially remains the same for matrix factorization). This connection of dropout with weight decay will also be apparent in Module 4 when we look at how to train a Bayesian deep network.

It is important to realize that there is no subsampling of training dataset happening here like classical boosting; we are instead forming multiple models by adding randomness to how the input is propagating through the deep network. For a linear classifier this is equivalent because

$$\sum_{i=1}^{d} w_i \left(x_i \odot r_i^k \right) = \sum_{i=1}^{d} \left(w_i \odot r_i^k \right) x_k =: f(x; w^k);$$

we can either mask out the input or mask the weights and think of the masked weights w^k as a new model.

Remark 7.1. You will often see folks in the literature say that dropout regularizes by preventing co-adaptation of the neurons at each hidden layer. The motivation for this statement is that the weights of the succeeding layer cannot fixate too much upon a particular feature at the input because the feature can be zeroed out by the dropout mask. This prevents too much specialization of neurons in the hidden layer and ensures that the prediction is made using a large number of diverse features, not just a few specific ones. This is not a rigorous argument but it is a reasonable argument in view of the experiments of Hubel and Wiesel (see http://centennial.rucares.org/index.php?page=Neural_Basis_Visual_Perception). The human brain is robust to large parts of it going missing/inhibited.

Bagging is expensive at test time, it involves having to compute the predictions of all the models in the committee. In the case of dropout, in this linear heuristic argument, we can compute the committee prediction to be

$$\begin{aligned} f(x; w) &= \frac{1}{M} \sum_{k=1}^{M} \sum_{i=1}^{d} \left(w_i \odot r_i^k \right) x_k \\ &= \sum_{i=1}^{d} \left(w_i \odot \frac{1}{M} \sum_{k=1}^{M} r_i^k \right) x_k \\ &\approx \sum_{i=1}^{d} \left(w_i \odot (1-p) \right) x_k. \end{aligned} \tag{7.15}$$

This is very fortunate, it indicates that given weights w of a model trained using dropout, we can compute the *committee average* over models created using dropout masks simply by scaling the weights by a factor $1-p$. This should not be surprising, after all the equivalent training objective in (7.14) has $\tilde{w} = (1-p)w$ as the effective weights of the weights. Another important point to note is that there is no masking of activations at test time.

Although the argument in this section works only for linear models, we will bravely extend the intuition to deep networks.

7.3.3 Implementation details of dropout

The recipe for using dropout is simple: (i) the activations at the input of each dropout layer are zeroed out using a Bernoulli random variable of probability $1-p$ (the PyTorch layer takes the probability of zeroing out activations as argument which is p in our derivations; (ii) at test time, the weights of layers immediately preceding dropout are scaled by a factor of $1-p$ to compute the predictions of the "committee".

Inverted Dropout. It is cumbersome to remember the parameter p that was used for training at test time. Deep learning libraries use a clever trick: they simply scale the output activations of dropout layer by $1/(1-p)$ during training. Training or testing the modified model using dropout gives an extra factor of $(1-p)$ like (7.14) and (7.15) respectively and therefore the final model can be used as is without any further scaling of the weights or activations.

The operation `model.train()` in PyTorch sets the model in the training mode. This is a null-operation and does not do anything for fully-connected, convolutional, softmax etc. layers. For the dropout later, it sets a boolean variable in the layer that samples the Bernoulli mask for all the input activations and scales the output activations by $1/(1-p)$. The complementary operation is `model.eval()` in PyTorch which you should use to set the model in evaluation mode. This is again a null-operation for other layers but for the dropout layer, it resets this boolean variable to indicate that no Bernoulli masks should be sampled and no masking should be performed.

7.3.4 Using dropout as a heuristic estimate of uncertainty

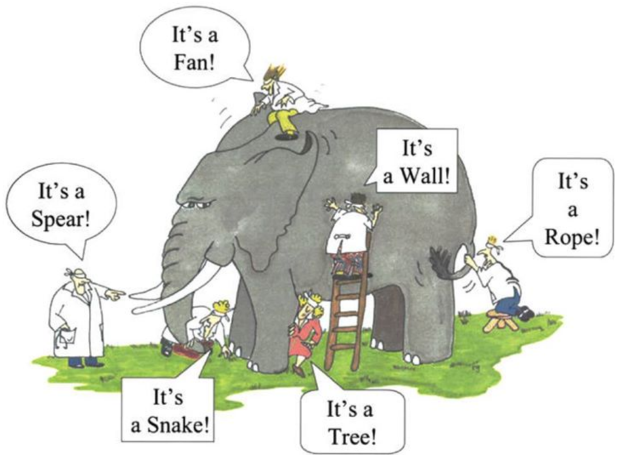
We can extend the motivation from bagging to use dropout as a cheap heuristic to get an estimate of the uncertainty of the prediction at test time. Suppose we use dropout at test time just like we do it at training time, i.e., each time one test input is presented to the deep network, we sample multiple Bernoulli masks $r^1, \ldots, r^M$ and compute multiple predictions for the same test input

$$\left\{ f(x; w, r^1), \ldots, f(x; w, r^M) \right\}.$$

The variance of these predictions can be used as heuristic of the uncertainty of the deep network while making predictions on the test input x. This is an estimate of the so-called *aleatoric or statistical uncertainty*. It captures our understanding that the weights w of a trained deep network are inherently uncertain and different training experiments, in particular, different masks r^k will give rise to different weights. The variance across a few sampled masks thus indicates how uncertain the model is about its predictions. Dropout is a neat and cheap trick for this purpose; it is quite commonly used in this fashion in medical applications where it is important to not only predict the outcome but also characterize the uncertainty of this prediction. We will see more powerful ways to compute aleatoric uncertainty in Module 4.

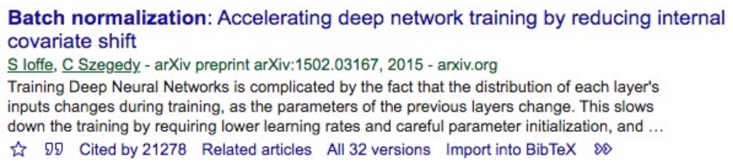
Remark 7.2. Broadly speaking, the connection of dropout with weight decay is contentious. If it were rigorous, we should be able to get the same performance as dropout by using appropriate weight decay (this is a good idea for the course project!). In practice, the validation error using dropout is very good and cannot be achieved by tweaking weight decay. Another aspect is that since we would like to average over lots of dropout masks in the training process, networks with dropout should be trained for many more iterations of SGD than networks without dropout to get the same training error. The benefit is that the test error is much better for dropout. What exactly dropout does is a subject of some mystery and there are other alternative explanations (e.g., Bayesian dropout in Module 4).

Our understanding of dropout is no different than that of these blind scientists trying to identify an elephant.



7.4 Batch-Normalization

Batch-Normalization (BN) is another layer that is very commonly used in deep learning. BN is very popular with more than 20,000 citations in about 5 years.


Batch normalization: Accelerating deep network training by reducing internal covariate shift
S Ioffe, C Szegedy - arXiv preprint arXiv:1502.03167, 2015 - arxiv.org
Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and ...
Cited by 21278 Related articles All 32 versions Import into BibTeX

7.4.1 Covariate shift

Covariate shift is a common problem with real data. The experimental conditions under which training data was gathered are subtly different from the situation in which the final model is deployed. For instance, in cancer diagnosis the training set may have an over-abundance of diseased patients, often of a specific subtype endemic in the location where the data was gathered. The model may be deployed in another part of the world where this subtype of cancer is not that common.

The mis-match between training and test *data* distribution is called covariate shift. Even if the labels depend on some known way $y|x$ on the covariates, i.e., given the genetic features of a person x their likelihood of a cancer y is the same regardless of which part of the world the person is from, the fact that we do not have training data from the entire population of the world forces the classifier to be tested on a data distribution that is different from what it was trained for.

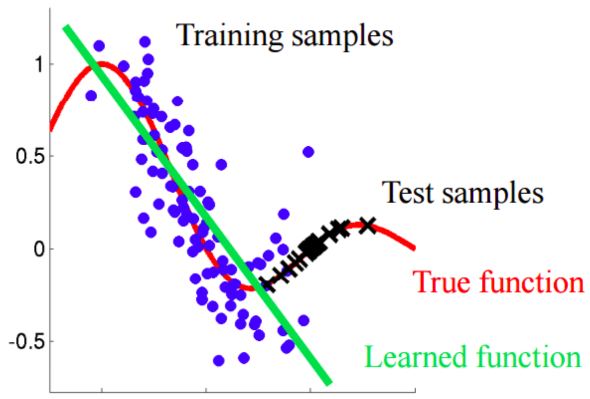


Figure 7.5: Covariate shift correction for a regression problem

Covariate shift is outside our fundamental assumption in Chapter 1 that training and test data come from the same distribution. It is however a problem that is often seen in practice and typical ways to counter it basically look as follows.

1. Train a classifier $\hat{w}$ on the available training data D.
2. Update the trained classifier using data from the test distribution $D' = \left\{(x^i, y^i)\right\}_{i=n+1,\dots,n+m}$ in addition to the original training dataset

$$w^* = \underset{w}{\text{argmin}}\ \frac{1}{n+m} \sum_{i=1}^{n+m} p^i\ \ell^i(w) + \Omega(w - \hat{w}) \tag{7.16}$$

   where p^i is some weighing factor that indicates how similar the datum (x^i, y^i) is to the *test data distribution*. The regularization $\Omega(w - w^*)$ forces the new weights w^* to remain close to the old weights $\hat{w}$.

The above methods go under the umbrella of *doubly robust estimation*. We will not study it in this course. The results look similar to the ones shown in Figure 7.5.

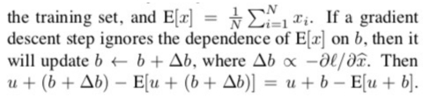
7.4.2 Internal covariate shift

If we are working under the standard machine learning assumption of test data drawn from the same distribution as the training data, then there is no covariate shift.

Recall that we whiten the inputs, say using Principal Component Analysis (PCA), for linear regression in order to decorrelate the input features; you can using a simple argument of how this changes the condition number of the data covariance matrix $X^\top X$ and accelerates the convergence of gradient descent using a calculation similar to the final problem in HW 2.

Deep networks are like any other model in this aspect and whitening of the inputs is also beneficial; the ZCA transform (or Mahalanobis whitening) is a close cousin of PCA and usually works better for image-based data. It is natural to expect that since each layer of a deep network takes the activations of the preceding layer as input, we should whiten the activations before the computation in the layer. The authors of the BN paper came upon an interesting through what is clearly a mistake. Their reasoning was based on a simple example: if we have a mini-batch of inputs $\left\{x^1, \ldots, x^{\text{ƀ}}\right\}$ and our layer simply adds a learnable bias b to this

⚠ This is the mistake in the original BN paper.


the training set, and $\mathrm{E}[x] = \frac{1}{N}\sum_{i=1}^{N} x_i$. If a gradient descent step ignores the dependence of $\mathrm{E}[x]$ on b, then it will update $b \leftarrow b + \Delta b$, where $\Delta b \propto -\partial\ell/\partial\widehat{x}$. Then $u + (b + \Delta b) - \mathrm{E}[u + (b + \Delta b)] = u + b - \mathrm{E}[u + b]$.

$$h = x + b.$$

If this layer whitens its output before passing it on to the next layer, we will have

$$\hat{h} := h - \frac{1}{\text{ƀ}} \sum_{i=1}^{\text{ƀ}} x^i.$$

The output $\hat{h}$ does not depend on the bias b. They argued, incorrectly, that the back-propagation update of the bias $\overline{b}$ is equal to $\overline{\hat{h}}$. This is not true of course because

$$\overline{b} = \overline{\hat{h}}\, \frac{\mathrm{d}\hat{h}}{\mathrm{d}b} = 0$$

in our notation where $\overline{h} = \mathrm{d}\ell/\mathrm{d}h$. Nevertheless the motivation of the batch-normalization operation is sound, we would like to whiten the input activations to each layer of a deep network.

> Batch-Normalization is a technique for whitening the output activations of each layer in a deep network.

Naively, this would involve computing expressions of the form

$$\hat{h} = (\mathrm{Cov}(h))^{-1/2} \left(h - \frac{1}{\text{ƀ}} \sum_{i=1}^{\text{ƀ}} h^i \right).$$

This is not easy to do because the features are high-dimensional vectors, the covariance matrix $\mathrm{Cov}(h)$ is a very large matrix. This makes computing $\hat{h}$ difficult for every mini-batch. Nevertheless, whitening helps and here is how it is done in the batch-normalization module:

$$\hat{h} = \frac{h - \mathbb{E}(\left\{h^1, \ldots, h^{\text{ƀ}}\right\})}{\sqrt{\mathrm{Var}(\{h^1, \ldots, h^{\text{ƀ}}\}) + \epsilon}}. \tag{7.17}$$

The constant ϵ in the denominator prevents $\hat{h}$ from becoming very large in magnitude if the variance is small for a particular mini-batch. It is important to note that both the expectation and the variance are computed for every feature. Let us make this clear: if $h \in \mathbb{R}^{\text{ƀ} \times p}$, i.e., p features for this layer, the $i^{\text{th}} \in \{1, \ldots, \mathscr{b}\}$ input of the mini-batch and the $j^{\text{th}} \in \{1, \ldots, p\}$ of the feature for $\hat{h}$ is given by

$$\hat{h}_{ij} = \frac{\hat{h}_{ij} - \frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} h_{ij}}{\sqrt{\text{Var}(\{h_{1j}, h_{2j}, \ldots, h_{\mathscr{b}j})\}}}.$$

Let us give names to these parameters

$$\begin{aligned} \mathbb{R}^p \ni \mu &= \mathbb{E}(\{h^1, \ldots, h^{\mathscr{b}}\}) \\ \mathbb{R}^p \ni \sigma^2 &= \text{Var}(\{h^1, \ldots, h^{\mathscr{b}}\}). \end{aligned} \tag{7.18}$$

The authors of the original BN paper decided that mere normalization may not be enough, e.g., if you normalize the activations *after a sigmoid activation*, the layer may essentially become linear because the activations are prevented from going too far to the right or too far too the left of the origin. This brings the second key idea in BN, that of affine scaling of the output $\hat{h}$. The BN layer really implements two

$$\hat{h} = a \left(\frac{h - \mathbb{E}(\{h^1, \ldots, h^{\mathscr{b}}\})}{\sqrt{\text{Var}(\{h^1, \ldots, h^{\mathscr{b}}\}) + \epsilon}} \right) + b. \tag{7.19}$$

where $a, b \in \mathbb{R}^p$, i.e., each feature has its own multiplier a and bias b. The final BN operation in short is therefore

$$\hat{h} = a \left(\frac{h - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + b.$$

The affine scaling parameters a, b are the only parameters in BN that are updated using backpropagation. The mean μ and variance σ^2 are unique to every mini-batch and therefore do not have any backpropagation gradient.

Execute the following code in your Jupyter notebook and check how the BN layer is implemented in PyTorch

```
import torch.nn as nn
m = nn.BatchNorm1d(15)
print(m.weight, m.bias)
print(m.running_mean, m.running_var)
```

The weight and bias here are the affine scaling parameters; and running_mean, running_var are μ, σ^2 respectively. You will see that requires_grad is True only for the former.

BN for convolutional layers

The activations of a convolutional layer are a 4-dimensional matrix (or a tensor)

$$h \in \mathbb{R}^{\mathscr{b} \times c \times w \times h}.$$

The distinction between convolutional layers compared to fully-connected layers is that the convolutional filter weights are shared for the whole input channel $w \times h$. We can therefore think of each *channel as a feature* and compute the BN mean and standard deviation over the batch dimension, as well as the width and height. In pseudo-code, this looks as follows.

```
# t is still the incoming tensor of shape [bb, c, w, H]
# but mean and stddev are computed along (0, 2, 3) axes and
have just [c] shape
mean = mean(t, axis=(0, 2, 3))
stddev = stddev(t, axis=(0, 2, 3))
for i in 0..bb-1, x in 0..h-1, y in 0..w-1:
    out[i,:,x,y] = normalize(t[i,:,x,y], mean, stddev)
```

Running updates of the mean and variance in BN

BN computes the statistics over mini-batches. Even if we trained a model using mini-batch updates we would still like to be able to use this model at test time with a single input; it may not always be possible to wait for a few test images to make predictions. The weights of the network are trained to work with whitened features so we definitely need some way to whiten the features of a test input, ignoring the whitening at test time will result in wrong predictions.

The BN layer solves this issue by maintaining a running average of the mean and variance statistics of mini-batches during training. Effectively, the buffers running_mean, running_var (note that these are not parameters/weights, they are not updated using backprop) are updated after *each mini-batch* during training as

$$\begin{aligned} \text{running\_mean}^{t+1} &= \rho\, \text{running\_mean}^{t} + (1-\rho)\, \mu \\ \text{running\_var}^{t+1} &= \rho\, \text{running\_var}^{t} + (1-\rho)\, \sigma^2. \end{aligned}$$

The parameter ρ is called a momentum parameter for the BN layer and makes sure that updates to running_mean/var are slow and one mini-batch cannot affect the stored value too much. Note that whitening is still performed at training time using μ, σ^2; we simply record the running average in the buffers running_mean/var. If model.train() is called, then the mini-batch statistics are used to whiten the features. If model.eval() is called, then the stored buffers running_mean/var are used to whiten the outputs.

How is all this related to internal covariate shift?

You might be surprised that nothing in this section is related to covariate shift that we discussed at the beginning. Let us try to understand heuristically why BN is said to help with internal covariate shift.

Each layer of a deep network treats its input activations as the data and predicts the output activations. As the weights of different layers are updated using backprop during training, the *distribution* of input activations keeps shifting. Effectively, each layer is constant suffering a covariate shift because the layers below it are updated and the weights of the top layers have to adapt to this shifting distribution. This is what is known as *internal covariate shift*. BN normalizes the output activations to approximately have zero mean and unit variance and therefore reduces the internal covariate shift.

⚠ There are many caveats with this heuristic argument. The main one is to observe that the backpropagation gradient of all layers is coupled, so it is not as if the layers are updated independently of each other and cause interval covariate shifts to the other layers; the updates of all the weights in the network are coupled and it is unclear why (or even if) internal covariate shift occurs.

7.4.3 Problems with batch-normalization

There are two big problems with BN.

1. The affine parameters are updated using backpropagation and small changes mini-batch statistics which can result in large changes to the whitened output $(h-\mu)/\sqrt{\sigma^2+\epsilon}$ will result in very large updates to a, b. This makes the affine parameters problematic when you train networks. In general, it is a good idea to first fit a model without the affine BN parameters, you can do so by using affine=False in nn.BatchNorm1d.

2. The mean and variance buffers of the BN layer are updated using runnings statistics of the per-mini-batch statistics. This does not affect training because the statistics of each mini-batch are computed independently, but it does affect evaluation because the buffers are used to whiten the features of the test input. If the test input has slightly different pixel intensity statistics than the training image, then the BN buffers are not ideal for whitening and such images are classified incorrectly.

BN before ReLU or ReLU before BN

Should we apply BN before or after the nonlinearity? The purpose of a BN layer is to keep the activations close to zero in their mean and a standard-deviation of one. Imagine if we are using a ReLU nonlinearity after BN, about half of our features h have negative values which the rectification will set to zero. In this case the distribution of features given to the next layer is not zero-mean, unit-variance so we are not achieving our goal of whitening correctly. Further, it is possible that the bias parameter b in BN is negative in which case the activations could mostly be negative and ReLU will set all of them to zero and result in a large loss in information. On the other hand, if we have BN after ReLU, the input to the BN layer has a lot of zeros and we are now computing mean/variance over a number of sparse features; the mini-batch mean/variance estimated here may not be accurate therefore BN may not perform its job of correctly whitening its outputs. You can read more about similar problems at http://torch.ch/blog/2016/02/04/resnets.html

As you can see, BN is an incredibly intricate operation without necessarily sound theoretical foundation for all the moving parts. But it works, training a deep fully-connected network is very difficult without BN, and even for convolutional layers it often makes training insensitive to the choice of learning rate. You should think about BN very carefully in your implementations; a lot of problems of the kind, “I trained my model, it gives a good training error but very poor validation error”, or “I am fine-tuning from this task, but get very poor validation error on a new task”, or other problems in reinforcement learning, meta-learning, transfer learning etc. can be boiled down to an incorrect/inaccurate understanding of batch-normalization. This is further complicated by the interaction with other operations such as Dropout, e.g., see https://arxiv.org/abs/1801.05134. Studying the effect of BN in meta-learning/transfer-learning is a good idea for a course project.

How does Dropout affect BN?

Since dropout is active during training, the buffered statistics are the running mean/variance of the dropped out activations. Dropout is not used at test time, so the test time statistics, even for the same image can be quite different. A simple way to solve this problem is to run the model in training model once on the validation set (without making weight updates using backpropagation) for the BN buffers to settle to their non-droppped out values and then compute the validation error; this usually results in a marignal improvement in the validation error.

Variants of BN

There are variants of batch-normalization that have cropped out to alleviate some of its difficulties. For instance, layer normalization (https://arxiv.org/abs/1607.06450) normalizes across the features instead of the mini-batch which makes it work better for small mini-batches. Another variant known as group-normalization computes the mean/variance estimate in BN across multiple partitions of the mini-batch which makes the result of group-normalization independent of the batch-size. These variants work in some cases and do not work in some cases and often the specific normalization is largely dependent on the problem domain, e.g., group normalization works better for image segmentation but layer normalization and batch-normalization do not so well there.

Chapter 8

Recurrent Architectures, Attention Mechanism

Reading

1. Goodfellow 10.1-10.3, 10.5-10.7, 10.9-10.12
2. D2L.ai book Chapters 8, 9, 10
3. Paper on long short-term memory (Hochreiter and Schmidhuber, 1997)
4. Paper on the Transformer architecture (Vaswani et al., 2017)

In this chapter we will consider data that is a function of time. Typical examples of such data are videos and sentences in written/spoken language. Some typical problems that we are interested in solving given such data are classifying the activity going on in a video, classifying the object that is being described in a sentence. We can also think of generative models for such temporal data, i.e., forecasting how the video/sentence will look like a few time-steps into the future using the approaches in this chapter.

We will look at three kinds of neural architectures, namely Recurrent Neural Networks (RNNs), and the Long Short-Term Memory (LSTM) and Attention modules, that are typically used to model such data.

8.1 Recursive updates in a Kalman filter, sufficient statistics

Consider a scalar signal in time $h_t \in \mathbb{R}$ that evolves according to some dynamics

$$h_{t+1} = a h_t + \xi_t;$$

with the scalar $a \in \mathbb{R}$ that we have modeled and the noise $\xi_t \in \mathbb{R}$ reflects our understanding that the scalar a in our model of evolution of the signal h_t may not be the same as that of Nature. We model this discrepancy by setting ξ_t to be zero-mean Gaussian noise that is i.i.d across time

$$\xi_t \sim N(0, \sigma_\xi^2).$$

Let us say that our dataset consists of observing the signal for some time $\{x_1, x_2, \dots, x_t\}$. Think of h_t being the location of a car at time t and our dataset being the observation of the trajectory of vehicle up to time t. Assume that we do not observe the true trajectory of the vehicle, but observe some noisy estimate of the state at each time

$$x_t = h_t + \nu_t$$

where $\nu_t \sim N(0, \sigma_\nu^2)$ is the noise in our observation.

In this section, we will estimate the true signal at the next time instant $\hat{h}_{t+1}$. A good estimate is the one that minimizes the MSE loss with the true (unknown) signal

$$\underset{\hat{h}_{t+1}}{\text{argmin}} \underset{\xi_1, \nu_1, \dots, \xi_{t+1}, \nu_{t+1}}{\mathbb{E}} \left[\left(h_{t+1} - \hat{h}_{t+1} \right)^2 \mid \underbrace{x_1, \dots, x_t, x_{t+1}}_{\text{“dataset”}} \right]. \tag{8.1}$$

The expectation is taken over the noise because there could be many trajectories that the system could have taken, each corresponding to a particular realization of the noise.

Our estimate should only depend on the dataset

$$\hat{h}_{t+1} = \text{function}\left(x_1, \dots, x_t, x_{t+1}\right).$$

⚠ In machine learning parlance, this setup is called online learning where data occur sequentially one after other and you train/update the model to incorporate the latest datum; future predictions of this model are made using this updated model.

Since predictions are likely to be required across a long range of time, we want to construct a *recursive* update for $\hat{h}_{t+1}$ that takes in the estimate at the previous time-step $\hat{h}_t$ and updates it using the most recent observation x_{t+1}.

Kalman filter updates sufficient statistics

Like we computed the optimal predictor in the bias-variance tradeoff for regression as the conditional distribution of the labels given the data, it is possible to prove that the best estimate $\hat{h}_{t+1}$ is the conditional mean given past data

$$\hat{h}_{t+1} = \mathbb{E}\left[h_{t+1} \mid x_1, x_2, \dots, x_{t+1}\right].$$

Not surprisingly, to estimate the location of the car at time $t+1$, you need to watch the entire past trajectory of the car.

However, surprisingly, a powerful and deep result in control theory is that for our problem (where the model of the signal is linear with additive Gaussian noise and our observations x_t are a linear function of h_t corrupted with Gaussian noise) we only need to recursively update of the first two moments of our estimate. If we have

$$\hat{h}_{t+1} = N(\mu_{t+1}, \sigma_{t+1}^2)$$

where

$$\begin{aligned} \mu_{t+1} &= \mathbb{E}\left[\hat{h}_{t+1} \mid x_1, \dots, x_{t+1}\right] \\ \sigma_{t+1} &= \text{var}\left(\hat{h}_{t+1} \mid x_1, \dots, x_{t+1}\right). \end{aligned} \tag{8.2}$$

and update the mean and variance recursively using their values at the previous time-step as

$$\begin{aligned}\mu_{t+1} &= \mu_t + k_t \left(x_{t+1} - a\hat{h}_t \right) \\ \sigma_{t+1} &= \sigma_t^2 \left(1 - k_t\right) \\ k_t &= \frac{a^2\sigma_t^2 + \sigma_\nu^2}{a^2\sigma_t^2 + \sigma_\nu^2 + \sigma_\xi^2}.\end{aligned} \tag{8.3}$$

You can derive this part very easily. Show that if the objective in (8.1) was minimal at time t, then the expressions in (8.3) also minimize it at time $t+1$. This algorithm is known as the Kalman filter is one of the most widely used algorithms for estimation of signals based on their observation. The key property to remember for us from the Kalman filter is the following.

The two quantities μ_t, σ_t capture *all* the information from the past trajectory $x_1, \dots, x_t$. Instead of creating our MSE estimate $\hat{h}_t$ using the entire data as shown in (8.1), if we maintain these two quantities and recursively update them using (8.3) we obtain the best MSE estimate.

In other words, μ_t, σ_t are sufficient statistics of the data $x_1, \dots, x_t$ for the problem of estimating the next state h_{t+1}. The notion *sufficient statistic* means that you do not need anything beyond these two to estimate any function of the data $x_1, \dots, x_{t+1}$. A statistic is simply any function of data, therefore a sufficient statistic is a quantity such that if you have it, you can throw away all the data without losing any information. Not all statistics are sufficient, and not all sufficient statistics look like a few moments of data; for more interesting signals the sufficient statistics are non-trivial and difficult to find.

The structure of neural architectures for sequence modeling is intimately related to the above result. Just like a CNN learns the best features that classify the input data, a recurrent model learns the best statistics of the past sequence (sufficient) that predict the future elements of the sequence.

8.2 Recurrent Neural Networks (RNNs)

The data to an RNN is a set of n sequences

$$D = \left\{ (x_1^i, y_1^i), (x_2^i, y_2^i), \dots, (x_T^i, y_T^i) \right\}_{i=1,\dots,n}.$$

Each sequence has length T and each element of the sequence $x_t^i \in \mathbb{R}^d$. There can be labels at every time-step, e.g., these labels can be, say, ground-truth annotations of the activity "playing with a basketball" going on the video at that time, or also forecasting the inputs by one (or more) time-steps $y_t^i := x_{t+1}^i$.

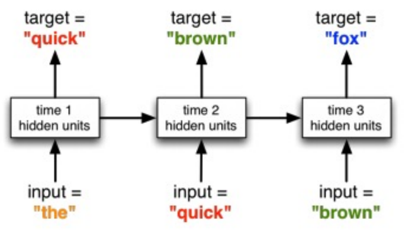


Figure 8.1: A recurrent model predicting the next word in a sentence.

Focus on one particular sequence $\{(x_1, y_1), \dots, (x_T, y_T)\}$ from the dataset. To predict the labels y_t at each time instant, the RNN would like to maintain a statistic, let us denote it by

$$h_t^i = \varphi\left((x_1, y_1), \dots, (x_t,)\right).$$

where φ is some function that we would like to build. Similar to a Kalman filter we *hope to learn* a sufficient statistic, in this case sufficiency means that the quantity h_t can predict the target y_t. Again, we would like to update the statistic recursively.

$$h_{t+1} = \varphi\left(h_t, x_{t+1}\right); \tag{8.4}$$

notice the similarity with the updates in (8.3) where updates to μ_t, σ_t also used the latest observation x_{t+1}. We will also have the RNN use the latest input x_{t+1}. You can think of h_t as a summary of the past sequence or some memory that is updated recursively. This summary/statistic is also called the "hidden state" in the RNN literature.

⚠ Note that just like we cannot claim that the features learned by a CNN are sufficient features, i.e., the only information from the data necessary to predict the targets, we cannot *claim* that h_t is a sufficient statistic of the past sequence. If the RNN/CNN is making predictions accurately, then it is reasonable to expect that we have learned something close to a sufficient statistic.

We do not know what function φ to pick so we are going to learn it using parameters. We will set

$$h_{t+1} = \sigma\left(w_h\, h_t + w_x\, x_{t+1}\right); \tag{8.5}$$

where $w_h \in \mathbb{R}^{p\times p}, w_x \in \mathbb{R}^{p\times d}$ are weights that multiply the previous statistic and the current input to calculate the current statistic. Again $\sigma(\cdot)$ is a nonlinearity that is applied element-wise.

Weights of an RNN are not a function of time. It is important to observe that the weights w_h, w_x do not change as the sequence moves forward. The same function is used to update the statistic at different points of time; notice that this does not mean that the statistic h_t^i remains the same across t. In this sense, an RNN is effectively the same neural model unrolled into the future as it takes in inputs of a sequence.

Output predictions can now be made as usual by learning weights

$$\hat{y}_t = v^\top h_t. \tag{8.6}$$

The loss function of an RNN is a sum of the error in the predictions for all time-steps

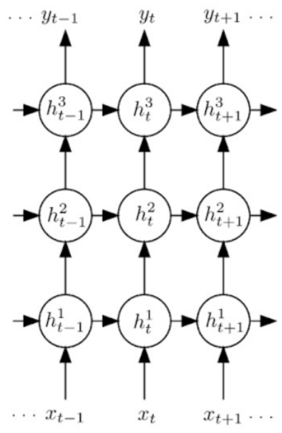
$$\sum_{t=1}^{T} \ell(y_t, \hat{y}_t) \tag{8.7}$$

and we can train the RNN by updating weights w_h, w_x using backpropagation. In some problems, you may only have targets for the final time-step y_T (say predicting whether it is going to rain right now or not based on the weather data of the past few hours); this does not change much conceptually, we will simply have only one term in the summation of the loss above.

How should we initialize the first hidden vector h_0 in an RNN? We have not seen any element of the sequence yet, so the value of h_0 has no meaning per se. Typically, h_0 is initialized either using Gaussian noise; it is sometimes also initialized to zeros.

Multi-layer RNNs

We have created a single-layer RNN in (8.5). We can use the same idea to create a multi-layer RNN the same way that we did for CNNs. We combine different parts of the hidden state/statistic and use these as features. In an RNN, it is traditional to combine the features both from the lower layer and features form the previous time-step of the same layer. As a picture it looks as follows



We can write an expression for this as

$$h_t^{l+1} = \sigma\left(w_{tt}\, h_{t-1}^{l+1} + w_{hh}\, h_t^l\right).$$

Again we have used trainable weights $w_{tt} \in \mathbb{R}^{p\times p}$ and $w_{hh} \in \mathbb{R}^{p\times p}$ to compute the hidden state/statistic/activations of the top layer. For a multi-layer RNN with L layers, the predictions at each time step are given by

$$\hat{y}_t = v^\top h_t^L.$$

The utility of having multiple layers in an RNN is similar to that of a CNN, more layers let us create more complex predictors than the recurrent perceptron-style predictor in (8.6) by learning a richer set of features.

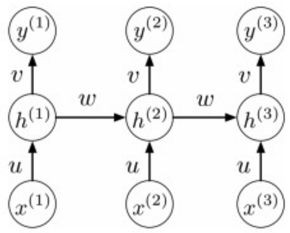
8.2.1 Backpropagation in an RNN

Let us see how to compute the gradient of the loss function with respect to the weights of an RNN in order to train the model using SGD. We will consider a sequence of two time-steps for a single-layer RNN

$$\begin{aligned} h_1 &= \sigma(u x_1) \quad \text{where we set } h_0 = 0 \\ \hat{y}_1 &= v h_1 \\ h_2 &= \sigma(u x_2 + w h_1) \\ \hat{y}_2 &= v h_2 \end{aligned} \tag{8.8}$$

The weights we would like to update are u, v and w. Let us say that the loss function is only computed at the final time-step $t = 2$ as $\ell := \ell(y_2, \hat{y}_2) = \|y_2 - \hat{y}_2\|^2$. Using our notation for backpropagation we have

⚠ Computational graph of a single-layer RNN. Please ignore the notation in this figure and see (8.8).



$$\begin{aligned}
\frac{\mathrm{d}\ell}{\mathrm{d}\ell} &= \overline{\ell} = 1 \\
\overline{\hat{y}_2} &= \overline{\ell}\,\frac{\mathrm{d}\ell}{\mathrm{d}\hat{y}_2} \\
&= -(y_2 - \hat{y}_2). \\
\overline{v} &= \overline{\hat{y}_2}\,\frac{\mathrm{d}\hat{y}_2}{\mathrm{d}v} \\
&= -(y_2 - \hat{y}_2)\,h_2 \\
\overline{h_2} &= \overline{\hat{y}_2}\,v \\
\overline{u} &= \overline{h_2}\,\sigma'(ux_2 + wh_1)\,x_2 \\
&\vdots
\end{aligned}$$

You should write down the update steps completely for an RNN making predictions at each time-step, using the loss function

$$\ell := \|y_1 - \hat{y}_1\|^2 + \|y_2 - \hat{y}_2\|^2$$

and see how the gradient of the loss at each time-step with respect to weights "accumulates" in $\overline{w}, \overline{v}$ and $\overline{u}$. Backpropagation in RNNs is also called backpropagation-through-time (BPTT). There is nothing special going on inside BPTT, it is simply backpropagation applied to a computational graph that is unrolled in time.

8.2.2 Handling long-term temporal dependencies

Implementations of BPTT for RNNs has a number of numerical issues.

Gradient vanishing

Notice that the gradient

$$\begin{aligned}
\overline{u} &= \overline{h_2}\,\sigma'(ux_2 + wh_1)\,x_2 \\
&= -(y_2 - \hat{y}_2)v\,\sigma'(ux_2 + wh_1)\,x_2.
\end{aligned}$$

in our backprop equations depends on the gradient of non-linearity. If we have a sigmoid non-linearity and the input activations to it $ux_2 + wh_1$ have large magnitude, the output h_2 will be saturated. This results in $\overline{u}, \overline{h_2}$ having small magnitudes. Further notice that $\overline{u}$ also depends upon products of the weights v and the inputs x_2. If you unroll this further for a few more time-steps (like we did in HW2) you will see that even future activations h_t are recursive products of past activations with weights. It is easy to observe that if we have a matrix A and a vector x the product

$$\lim_{k \to \infty} A^k x \tag{8.9}$$

goes to zero if the largest eigenvalue of A is less than 1, i.e., $\lambda_{\text{max}} = \|A\|_2 < 1$. The product goes to positive/negative infinity if the largest eigenvalue is greater than 1 if x has a non-zero inner product with the corresponding eigenvector. In other words, if the length of the sequence is long, the recursive computation in an RNN entails that the activations can blow up to infinity in both cases, this can also lead to gradient explosion; they can also become zero which can result in gradient vanishing.

The same is also true for CNNs with many layers: the weights of the lower layers get their backprop gradient after it goes through multiple nonlinearities (ReLUs lead to saturation as well if the input is negative) and can therefore receive a small gradient. While typical CNNs have 10 or so layers, typical RNNs handle sequences of length 50–100 (or more). The chance of having vanishing gradients to the weights is thus much higher in RNNs.

You would think that if the objective is a sum of the loss at each time; this alleviates the problem of gradient vanishing. But there is a deeper point we are trying to make here. The backprop gradient is an indication of how much we should change u, h_2 to make more accurate predictions at some future time-step y_t. If $t \gg 2$, the value of h_2 does not play a strong role in making predictions too far into the future. In other words, the predictions of the RNN become myopic we do not learn statistics that are a function of the entire past trajectory, the statistics are highly dominated by the near past which makes it difficult to capture long-range correlations in the sequence and predict high-level concepts.

Which nonlinearities are good for RNNs?

Think about which nonlinearities are good for training RNNs. Gradient vanishing is a large problem with sigmoids whereas both gradient vanishing and gradient explosion can occur for ReLU nonlinearities. You might be tempted to design a nonlinearity that does not saturation on either side of the origin but such nonlinearities look closer to and closer to an identity mapping and as we have a seen a linear model is much less powerful than a nonlinear model. In other words, gradient explosion/vanishing is a problem in BPTT for RNNs but there is really no effective solution to it.

Gradient clipping

We can avoid gradient explosion from ruining the weights being updated by gradient descent using gradient clipping. There are many ways of implementing this idea. The most prevalent one is to clip the ℓ_2 norm of the gradient to a pre-specified value. The SGD update is modified to be

$$w^{t+1} = w^t - \eta \, \text{clip}_c(\nabla \, \ell^{\omega_t}(w^t))$$

where $\nabla \, \ell^{\omega_t}(w^t)$ is the gradient of the objective on the sample with index $\omega_t \in \{1, \ldots, n\}$ in the dataset computed at weights ω_t and clipping performs the operation

$$\text{clip}_c(v) = \frac{cv}{\|v\|_2 + \epsilon}$$

where c is a pre-specified value and it is the ℓ_2 norm of the clipped gradient. The scalar ϵ in the denominator prevents numerical issues when the gradient magnitude is small.

⚠ The function clip_grad_norm performs gradient clipping. When you observe it closely you will realize that it is really scaling the gradient and should therefore be called gradient scaling.

Sometimes you instead clip the per-weight gradient at values $[-c, c]$, i.e., if the gradient vector is $v \in \mathbb{R}^p$ and v_k is the gradient at the k^{th} element

$$\text{clip}_c(v) = [\min(\max(-c, v_1), c), \ldots, \min(\max(-c, v_p), c)] .$$

Orthogonal initializations

All eigenvalues of an orthogonal matrix have an absolute value of 1. If A is an orthogonal matrix, we have

$$A^\top A = I.$$

This helps when we perform repeated multiplication with the weight matrices in forward-backward propagation because the norm of the intermediate products does not change

$$\|A^k x\|_2 = \|x\|$$

if A is orthogonal. The weight matrices of an RNN are typically initialized as orthogonal matrices; this is easy to do by first initializing the matrix using random Gaussian entries as usual and then setting the actual weights to be the left singular vectors after computing an SVD of the matrix.

❓ If the weights of an RNN are initialized as orthogonal matrices, do they remain so all through training after multiple steps of SGD?

Moving window over the data

We wrote down SGD updates as sampling a random (input,target) pair from the dataset at each iteration. The data for an RNN consists of a number of trajectories/sequences. We can sample one (or a mini-batch) of such sequences and a contiguous chunk of each of those sequences as a mini-batch in an RNN

$$\begin{aligned} D_{\text{mini-batch}} = &\left\{(x_1^i, y_1^i), \ldots, (x_{25}^i, y_{25}^i)\right\} \cup \\ &\left\{(x_5^j, y_5^j), \ldots, (x_{30}^i, y_{30}^i)\right\} \cup \\ &\left\{(x_{13}^k, y_{13}^j), \ldots, (x_{38}^i, y_{38}^i)\right\} \cup \\ &\vdots \end{aligned}$$

The hidden state h_0 of the RNN is initialized to zero/randomly at the beginning for all these trajectories.

We can also play a neat trick while sampling mini-batches in an RNN to give it the ability to handle more long-range correlations. The mini-batch is treated as a moving window over the data and it is rolled forward sequentially, i.e.,

$$\begin{aligned} D_{\text{mini-batch 1}} = &\left\{(x_1^i, y_1^i), \ldots, (x_{25}^i, y_{25}^i)\right\} \cup \\ &\left\{(x_1^j, y_1^j), \ldots, (x_{25}^i, y_{25}^i)\right\} \cup \\ &\left\{(x_1^k, y_1^j), \ldots, (x_{25}^i, y_{25}^i)\right\} \cup \ldots \end{aligned}$$

and the next mini-batch is chosen to be

$$\begin{aligned} D_{\text{mini-batch 2}} = &\left\{(x_{25}^i, y_{25}^i), \ldots, (x_{50}^i, y_{50}^i)\right\} \cup \\ &\left\{(x_{25}^j, y_{25}^j), \ldots, (x_{50}^i, y_{50}^i)\right\} \cup \\ &\left\{(x_{25}^k, y_{25}^j), \ldots, (x_{50}^i, y_{50}^i)\right\} \cup \ldots \end{aligned}$$

In this case, we simply copy the hidden state/statistic h_{25} of the previous mini-batch as the initialization h_0 for the next mini-batch. While creates strong correlations in the consecutive mini-batches and data for SGD is not sampled iid, it is a clever trick to increase the effective rage of temporal correlations modeled in the RNN without essentially any special operations. You can see an implementation of this idea at
https://github.com/pytorch/examples/blob/master/word_language_model/main.py#L131

Roughly speaking, data that consists of sequences of length up to 25 can be trained with RNNs.

8.3 Long Short-Term Memory (LSTM)

Innovations on top of the basic RNN architecture try to improve their ability to handle long-range correlations in the data. We saw that the updates to the hidden state/statistic h_t is the key to doing so. The architectures called LSTMs, and their simpler counterparts called GRUs, are mechanisms that give us more control to update the hidden state.

8.3.1 Gated Recurrent Units (GRUs)

GRUs "gate" the hidden state, i.e., the architecture has a mechanism to control when the hidden state gets updated and when it does not. For instance, if the first symbol in our sequence is very predictive of the future of the sequence we want the RNN to learn to not update the hidden state, and similarly if there are irrelevant words in the middle of the sequence we want the hidden state to not be updated at those time-steps. A GRU also has a mechanism to "reset" the hidden state that reduces the influence of the previous hidden state on the next hidden state.

⚠ The idea that the hidden state is the memory in sequence models is more clear in this context. In some cases we may want to update our memory after observing a particular part of the sequence, in some cases we want to keep the memory unchanged while in some cases we may wish to reinitialize the memory before observing the future data.

Recall that the hidden state for an RNN with a single layer is updated as

$$h_{t+1} = \sigma\left(w_h h_t + w_x x_{t+1}\right).$$

A GRU has two more variables that are called the reset variable and the zero variable respectively, each created from previous x_t, h_t using learnable weights

$$\begin{aligned} r_{t+1} &= \text{sigmoid}(w_{xr}\ x_t + w_{hr}\ h_t) \\ z_{t+1} &= \text{sigmoid}(w_{xz}\ x_t + w_{hz}\ h_t). \end{aligned} \tag{8.10}$$

The entires of r_t, z_t are between $(0, 1)$. The update to the hidden state in an RNN is modified to be

$$h_{t+1} = z_{t+1} h_t + (1 - z_{t+1}) \odot \tanh\left(w_h\left(r_{t+1} \odot h_t\right) + w_x x_{t+1}\right). \tag{8.11}$$

If entires of z_{t+1} are close to 1, the old state is propagated almost unchanged to result in h_{t+1}; information from x_{t+1} is essentially ignored in this case. In z_{t+1} are close to zero, the reset gate is used to decide what the next state h_{t+1} is: if r_{t+1} is close to one, the update is the same as that of a conventional RNN, if r_{t+1} is close to zero, the previous hidden state does not play any role in the update and the update is only dependent on the observation x_{t+1}.

⚠ GRUs are very useful recurrent models because they are more general than RNNs but at the same time much simpler than other models such as LSTMs. In most cases, it is a good idea to first try to fit the data using a GRU before using more complex models.

8.3.2 LSTMs

The design of an LSTM was inspired by logic gates in a computer and is a bit complicated; the original LSTM paper is an assigned reading for this lecture. LSTMs are powerful models in sequence modeling and in spite of being developed all the way back in 1997, they are among the few deep learning models that remained popular through the second AI winter and are still the workhorse of the NLP industry today.

An LSTM has three new variables on top of an RNN, these are called the "input, forget, and output" gates respectively

$$\begin{aligned} i_{t+1} &= \sigma(w_{hi}\, h_t + w_{xi}\, x_{t+1}) \\ f_{t+1} &= \sigma(w_{hf}\, h_t + w_{xf}\, x_{t+1}) \\ o_{t+1} &= \sigma(w_{ho}\, h_t + w_{xo}\, x_{t+1}) \end{aligned} \tag{8.12}$$

where all the above weight matrices are learnable parameters. In the GRU we had the convex combination using the zero gate in (8.11) to prevent forgetting. In an LSTM we use the two gates f_t, i_t for this purpose. The hidden state of an LSTM is propagated as

$$h_{t+1} = o_{t+1} \odot c_{t+1} \tag{8.13}$$

where the variable

$$c_{t+1} = f_{t+1} \odot c_t + i_{t+1} \odot \tanh(w_{hc}\, h_t + w_{xc} x_{t+1}) \tag{8.14}$$

is thought of as a memory cell. Understanding crisply what an LSTM ought to learn is a bit difficult but we can think of an LSTM as parameterizing the operations of GRU; convex combination in (8.11) is replaced by a weighted combination using the input and forget gates in (8.14) while the output gate in (8.13) is identity in a GRU.

Just like we can handle multiple layers in an RNN, we can also have multiple layers in an GRU. Each layer gets its own gates; temporal propagation is performed using the above equations and only the hidden state h_t is propagated up to the deeper layers.

You will notice that a lot of non-linearities in GRUs/LSTMs are sigmoids and hyperbolic tangents. This is because these gates are interpreted as Boolean variables that the model is supposed to learn. There are two lessons to draw from this. First, if you are modeling some computation and would like to learn a Boolean variable, it is a good idea to compute a learnable function of the inputs and use a sigmoid nonlinearity. Second, vanishing gradients are a problem with LSTMs/GRUs as well, the various mechanisms (reset/zero in GRUs and input/forget/output in LSTMs) alleviate this to an extent but do not eliminate vanishing gradients. Roughly speaking, we can use LSTMs to model sequences of up to length 50.

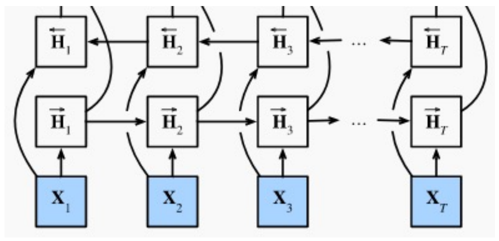
8.4 Bidirectional architectures

Until now, we have imagined that we would like to predict the future words in a sequence or design a predictor that uses a statistic of the sequence to predict the output. Our recurrent models were causal in the temporal direction, i.e., future elements of the sequence did not play a role in the outputs and updates of the model at time t. This is indeed how a lot of computation is performed, e.g., if you want to predict the next location of a vehicle in a video, you should not build a predictor that uses future frames because this model cannot be run at test time without access to the future frames. However, there are also problems in which you have access to some future observation and estimate the present state. For instance, you may fill in the following blanks totally differently depending upon the context of the future words.

I am very ______ .
I am very ______ for school.
I am very ______ , I need a big dinner.

Bidirectional models help us distinguish between the three situations and allow predicting context-specific output. Just like we motivated recurrent models using a Kalman filter and sufficient statistics of the past sequence, we can also derive an analogy with what is called Kalman smoothing (predicting the current state given the past observations *and* the future observations).

Building bidirectional models using RNNs is easy. We have two RNNs running in opposite directions as shown in the following picture.


$\overleftarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_3$
...
$\overleftarrow{\mathbf{H}}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
...
$\overrightarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

We maintain two sets of weights, one for the forward RNN and the other for the backward RNN. This gives two hidden states, one in the forward direction and another in the backward direction

$$\begin{aligned} h_{t+1}^{\text{forward}} &= \sigma(w_h^{\text{forward}}\, h_t^{\text{forward}} + w_x^{\text{forward}}\, x_{t+1}) \\ h_t^{\text{backward}} &= \sigma(w_h^{\text{backward}}\, h_{t+1}^{\text{backward}} + w_x^{\text{backward}}\, x_t). \end{aligned}$$

The concatenation of these two hidden states is now the sufficient statistic of the entire sequence. So the output $\hat{y}_t$ is now a function of both these hidden states

$$\hat{y}_t = {v^{\text{forward}}}^\top h_t^{\text{forward}} + {v^{\text{backward}}}^\top h_t^{\text{backward}}. \tag{8.15}$$

Let us emphasize that these two directions have nothing to do with backpropagation. There is a backpropagation for the backward directions as well, which updates $\overline{h_{t+1}^{\text{backward}}}$ using $\overline{h_t^{\text{backward}}}$. You should do the following exercise: imagining that the loss is only computed on the predictions at time t, i.e., $\ell = \ell(y_t, \hat{y}_t)$ and think of how the backpropagation gradient flows in a bidirectional RNN.

Just like we have bidirectional RNNs, we can also build bidirectional GRUs and LSTMs.

8.5 Attention mechanism

The human perception system is quite limited by its sensors, we do not have eyes at the back of our heads. It is also limited by computation, the human brain consumes only about 12W of power when it works, about 30% of this power is consumed by the visual system.

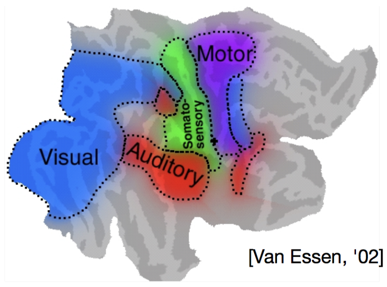


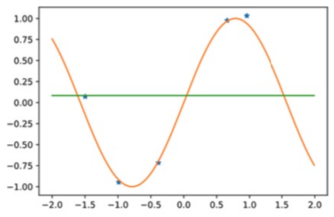
Figure 8.2: This is a picture of the human brain by a famous neuroscientist named David Van Essen. Around the early 90s it became clear that brains consist of different parts, each specialized to processing different kinds of data. The visual system takes up a bulk (30%) of the real estate.

Our perceptual system is very powerful considering the limits of this computation. We discussed reasons for this in Chapter 1, the ability to move gives us the ability to specialize the processing on different parts of the environment instead of passively processing all the incoming data from the sensors. For instance, when you are driving, you look over your shoulder only before you merge on the right, you do not really care to remember where every car in your vicinity is at any given point of time. Similarly, experiments on race car drivers reveal that even at high speeds they do not pay attention to all parts of the environment, a driver typically only cares about two variables, the heading of the car while going into a turn and the distance to the apex of the turn. When you watch TV, you are paying attention to only a small part of the TV screen. You can read more about these experiments at http://ilab.usc.edu/surprise and in the work of many other researchers who study such problems.

The human perceptual system is tuned to pay attention to only parts of the input data that is relevant. Attention in machine learning is an attempt to model this phenomenon. It turns out that since understanding which part of a long sequence is relevant to making a prediction at a particular time instant, attention is well-suited to mitigating the problems with long-range correlations in sequence data. We will not go very deep into the architectural intricacies of attention models (you can read the suggested reading material) but we provide an introduction that makes it easy to understand the papers.

8.5.1 Weighted regression estimate

Consider a regression problem where the true function is drawn in orange and the dataset is shown in blue.



If we wanted to predict the targets the green line given by

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} y^i.$$

is the world's dumbest estimator; it predicts the output irrespective of the input x. We can do better using the so-called Watson-Nadaraya estimator in statistics. We compute the weighted combination

$$\hat{y}(x) = \sum_{i=1}^{n} k(x, x^i)\; y^i$$

where kernel $k(x, x^i)$ computes some similarity between the input x^i in the dataset and the test input x; the kernel weighs the target y^i higher if x is close to x^i.

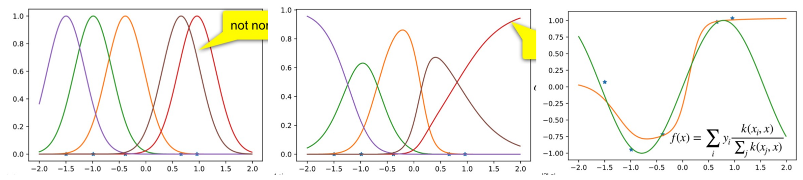


Figure 8.3: The left panel shows the Gaussian kernel $k(\cdot, x^i)$ for different inputs in the dataset. The kernel is not normalized so we cannot match the target values y^i easily using a weighted combination of the kernels. The second panel fixes this by picking a normalized kernel $k(x, x^i) := \frac{k(x,x^i)}{\sum_j k(x,x^j)}$. The estimate of the target $\hat{y}(x)$ using a weighted combination of this normalized kernel is a non-parametric estimator of the targets.

The Watson-Nadaraya estimator in Figure 8.3 is a simple interpolation mechanism and it is also consistent, i.e., as the amount of data $n \to \infty$, the regression error goes to zero. There are no weights in this model. All the intricacy lies in choosing the kernel over the data.

An attention layer can be thought of as learning the weighing function in our regression estimate, and a weighted average instead of an unweighted average.

8.5.2 Attention layer in deep networks

Let us consider a typical kind of attention that is heavily employed in deep learning. It is called the dot-product attention mechanism. This takes in two matrices as input: $k \in \mathbb{R}^{T \times p}$ which is called the "key" and $v \in \mathbb{R}^{T \times p}$ which are called "values". Given a query vector $q \in \mathbb{R}^p$ the attention module outputs

$$\sum_{i=1}^{T} \sigma\left(k_i^\top q\right) v_i \tag{8.16}$$

where k_i denotes the i^{th} row of the matrix and likewise for the values. Observe that the summation is a weighted combination of all the values v_i with weights given by the similarity of the query with each of the keys k_i. Just like the Watson-Nadaraya estimator, we would like these weights to be normalized, so we choose

$$\sigma(k_i^\top q) = \text{softmax}_i(k_i^\top q);$$

the softmax normalization is performed over the time-axis i. In simple words, the expression is a weighted combination of the values where the kernel is computed using a simple dot product and normalization of the kernel is performed using softmax. If a particular query vector q is similar to one of the keys k_i, that value v_i gets upweighted in the summation.

If the query vector is one of the keys k_i, we have the so-called **self-attention operation**.

How can we use this in a deep network? First let us consider a standard convolutional network with features $h^l \in \mathbb{R}^{m\times c}$ at the l^{th} layer; we have reshaped the width and height of the feature map into a single dimension of size m, the number of channels is c. If we set the keys, values and queries to be learnable quantities

$$\begin{aligned}\mathbb{R}^{m\times c} &\ni k = \sigma(w_k^\top\, h^l)\\ \mathbb{R}^{m\times c} &\ni q = \sigma(w_q^\top\, h^l)\\ \mathbb{R}^{m\times c} &\ni v = \sigma(w_v^\top\, h^l)\end{aligned} \tag{8.17}$$

where $\sigma(\cdot)$ is some nonlinearity, say ReLU, then the output of the attention block would be given by a weighted summation over the features for each pixel given by

$$h_j^{l+1} = \sum_{i=1}^{m} \text{softmax}\left(k_i^\top q_j\right) v_i. \tag{8.18}$$

⚠ Draw a picture of the computation in an attention module

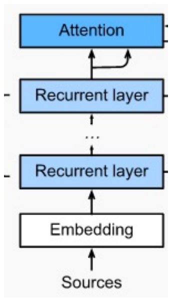
This is a just a more complex version of the correlation operator. It creates output features $h_j^{l+1}; j \in \{1,\dots,m\}$ that captures the similarities between the queries and the keys.

8.5.3 Attention as one of the layers of a recurrent networks

The attention layer is much more popular for sequence modeling because it offers a very powerful way to mitigate the problem with vanishing/exploding gradients for long sequences. For a sequence of length T, the attention layer computes the same operation as in (8.18). Observe that this expression, rewritten here with the number of features $m := T$ corresponding to the time dimension and the feature size $c := p$

$$h_j^{l+1} = \sum_{i=1}^{T} \text{softmax}\left(k_i^\top q_j\right)\ v_i$$

has hidden state h_j^{l+1} that depends on the hidden states of the lower layer $h_i^l, i \in \{1,\dots,T\}$. Effectively, the attention layer acts as a temporal shortcut that makes the hidden states of an RNN dependent on both past and future hidden states for the sequence. In a picture, this looks as follows.



The recurrent layers compute features in a causal fashion but the attention layer connects all the time-steps together. If you think of how backpropagation gradient flows down from the output layer via the attention, you will realize that the gradient of the loss computed at step t, say $\ell(y_t, \hat{y}_t)$ flows back to the hidden states h_2 using two paths; the first is the standard BPTT path of the recurrent layers while the second one is a more direct path of the cross-correlation operation in the attention layer. This is a huge benefit because it essentially eliminates problems with gradient vanishing and allows recurrent model very long sequences. Modifications of this attention module can easily handle sequences of a few hundred words.

To conclude, attention is a powerful operation and has become very popular in the past 1-2 years. It has been used predominantly for NLP models but also works surprisingly well as a replacement for convolutional layers for image-based data. One can think of the attention module as a fully-connected layer that performs very strong weight sharing.

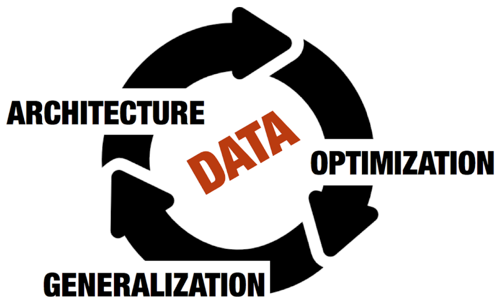
You can read Chapter 10.3 in the D2L.ai book and the original paper on a popular attention-based architecture called the Transformer to know more about this operation.

Chapter 9

Background on Optimization, Gradient Descent

We have covered the cliff-notes of the practice of deep learning in the previous eight chapters. It is by no means a complete overview. The practice of deep learning is an enticing, mysterious, and sometimes frustrating. The more time you spend playing with code the more you will learn about deep learning. New ideas are routinely discovered using very simple experiments that each of you is capable of running in your Colab now.

As we discussed, there three main concepts in machine learning. First, the class of functions $f(x; w)$ that you use to make predictions, this is called the hypothesis class or the architecture. Second, the algorithm you use to find the best model in this class of functions that fits your data; this uses tools from optimization theory. Third is the generalization performance of your classifier. Machine Learning is about picking a good hypothesis class, finding the best model within this class and making sure that the model generalizes.



The above process is relatively well-understood for simpler models such as SVMs but the story is quite murky for deep networks. Often in practice, it is never clear which architecture you should pick for your problem (many of you have asked this question in the office hours for instance). Training a deep network involves a number of bells and whistles (some of which like Batch-Normalization and Dropout that we have seen) and if at the end of this exercise we get a high validation error, it is unclear how one should change the parts of the process to improve performance. Disentangling this viscious cycle is what "understanding deep learning" is all about.

Goal Module 2 will develop an understanding of optimization and generalization for more generic machine learning models first. It will end with an insight into understanding their interplay for deep networks. Module 2 has a different flavor, it is more theoretical. Our goal is to grasp the general concepts behind these theoretical results and understand the training process of deep networks better. This will also help us train deep networks much better in practice.

9.1 Convexity

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

$$\ell(\lambda w + (1-\lambda)w') \leq \lambda \ell(w) + (1-\lambda)\ell(w'). \tag{9.1}$$

A function $\ell(w)$ is concave if $-\ell(w)$ is convex. Some examples of convex functions are

- affine functions $Aw + b$, norms $\|w\|_p = \left(\sum_{i=1}^p |w_i|^p\right)^{1/p}$, or $\|w\|_\infty = \max_k |w_k|$.
- exponential e^w for $w \in \mathbb{R}$
- powers w^α for $w > 0$ and $\alpha \geq 1$ or $\alpha \leq 0$
- powers of absolute values $|w|^p$ for $w \in \mathbb{R}$ and $p \geq 1$

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

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

First-order condition for convexity If ℓ is differentiable, the definition of convexity in (9.1) is equivalent to the following first-order condition. A differentiable function ℓ with convex domain is convex iff

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle . \tag{9.2}$$

for all w, w' in the domain. Note that the first-order condition is equivalent to the definition of convexity in (9.1) for differentiable functions. The proof is long but easy; you can see https://www.princeton.edu/ aaa/Public/Teaching/ORF523/S16/ORF523_S16_Lec7_gh.pdf for the proof. For strictly convex functions the inequality is strict

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

Monotonicity of the gradient for convex functions The first-order condition for convexity gives a useful, and equivalent, characterization of the gradient. Write (9.2) for w, w' in two opposite directions

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

and add them to get

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \geq 0. \tag{9.3}$$

It is also true that monotonicity of the gradient implies convexity (try to prove it).

Second-order condition for convexity If ℓ is twice-differentiable with a convex domain, it is convex iff

$$\nabla^2 \ell(w) \succeq 0 \tag{9.4}$$

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

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

For strictly convex functions, the inequality in (9.4) is strict, i.e., the Hessian is positive definite. As an example, the least squares objective $\ell(w) = \|y - Xw\|_2^2$ is convex because

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

which is positive definite for any non-singular X.

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

$$\ell(w) - \frac{m}{2}\|w\|_2^2 \text{ is convex.} \tag{9.5}$$

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

$$\begin{aligned} &\ell(\lambda w + (1-\lambda)w') - \frac{m}{2}\|\lambda w + (1-\lambda)w'\|^2 \\ &\leq \lambda \left(\ell(w) - \frac{m}{2}\|w\|^2\right) + (1-\lambda)\left(\ell(w') - \frac{m}{2}\|w'\|^2\right). \end{aligned} \tag{9.6}$$

But

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

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

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

In other words, we have

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

Observe that strongly convexity in (9.6) is a stronger version of Jensen's inequality. Strongly convex functions are easier to optimize for algorithms. It will also always be much easier to prove a result in optimization on strongly convex functions. It is easy to see using the second-order condition for convexity that an m-strongly convex function has

$$\nabla^2 \ell(w) \succeq m I_{p \times p}.$$

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

$$\ell(w') \geq \ell(w) + \langle \nabla \ell(w), w' - w \rangle + \frac{m}{2} \|w' - w\|^2 \tag{9.7}$$

for any w, w' in the domain.

9.2 Introduction to Gradient Descent

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

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

The objective ℓ will always be a function of the entire dataset but we will keep the dependence implicit. Note that the number of samples n is usually quite large in deep learning, so the summation above has a large number of terms on the right-hand side.

Gradient descent is a simple algorithm to minimize $\ell(w)$. Before we study its properties, it will help to refresh the following few facts.

9.2.1 Conditions for optimality

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

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

$$\ell(v) - \ell(w) \leq \lambda \left(\ell(w') - \ell(w)\right)$$

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

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

$$\ell(w) = \ell(w') \leq \ell(w'') \quad \forall w''. \tag{9.8}$$

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

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

which contradicts (9.8). The least-squares objective is strictly convex, so the solution is unique global minimizer of the objective.

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

$$\nabla \ell(w) = 0. \tag{9.9}$$

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

9.2.2 Different types of convergence

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

$$w^* = \underset{w}{\text{argmin}}\, \ell(w)$$

be the global minimizer of this function.

We would like to develop an iterative scheme that takes in the initialization of the weights w^0 and updates them to obtain a sequence

$$w^0, w^1, \ldots, w^t, \ldots$$

Along this sequence we are interested in understanding the

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

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

$$w^{t+1} = w^t + \eta\, d^t \tag{9.10}$$

where d^t is called the descent direction and the scaler parameter $\eta > 0$ is called the step-size and determines how far we travel using this descent direction. Any direction such that

$$\left\langle \nabla \ell(w^t), d^t \right\rangle < 0$$

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

$$w^{t+1} = w^t - \eta \, \nabla \, \ell(w^t) \tag{9.11}$$

given an initial value w^0. The step-size (also called the learning rate) is chosen by the user. The step-size need not always be fixed, for instance you chose it to be a function of the number of weight updates t in the homework. A good step-size is one that does not overshoot the minimum w^*. For instance, after having chosen a particular descent direction d^t we can compute the best step-size to use at time t by solving

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

$$\eta^t = \underset{\eta \geq 0}{\text{argmin}} \, \ell(w^t + \eta \, d^t).$$

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

$$w^{t+1} = w^t - \left(\nabla^2 \, \ell(w^t)\right)^{-1} \nabla \, \ell(w^t). \tag{9.12}$$

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

❓ Can you think of an algorithm for minimizing a function that does not use the gradient of the function to compute the descent direction?

9.3 Convergence rate for gradient descent

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

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

9.3.1 Some assumptions

Before we begin, we will make a few simplifying assumptions on the function $\ell(w)$. These are quite typical in optimization and ensure that we are not dealing with pathological functions that make minimizing them arbitrarily hard.

1. **Lipschitz continuity/bounded gradients** We will assume that ℓ is Lipschitz continuous

$$|\ell(w) - \ell(w')| \leq B \|w - w'\|_2. \tag{9.13}$$

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

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

for differentiable functions.

2. **Smoothness** We will always consider functions such that their gradients are L-Lipschitz, i.e.,

$$\|\nabla \ell(w) - \nabla \ell(w')\|_2 \leq L\|w - w'\|_2. \tag{9.14}$$

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

$$\nabla^2 \ell(w) \preceq L\, I_{p\times p}. \tag{9.15}$$

From the Cauchy-Schwarz inequality which states that

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

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

$$\langle \nabla \ell(w) - \nabla \ell(w^*), w - w^*\rangle \leq L\|w - w^*\|^2. \tag{9.16}$$

A related concept is called **co-coercivity** of the gradient. The gradient being L-Lipschitz is equivalent to co-coercivity of the gradient with parameter $1/L$

$$\frac{1}{L}\|\nabla \ell(w) - \nabla \ell(w)\|^2 \leq \langle \nabla \ell(w) - \nabla \ell(w'), w - w'\rangle\,. \tag{9.17}$$

We can see that co-coercivity implies Lipschitz continuity of the gradients $\nabla \ell(w)$ using (9.16) and (9.17). The reverse is also true, Lipschitz-continuity of the gradient implies the Descent Lemma Lemma 9.1 which is seen by applying the Descent Lemma twice for the two functions $g(u) = \ell(u) - \langle \nabla \ell(w'), u\rangle$ and $h(u) = \ell(u) - \langle \nabla \ell(w), u\rangle$.

9.3.2 GD for convex functions

We begin with the so-called Descent Lemma.

Lemma 9.1 (Descent Lemma). For an L-smooth function, we have

$$\ell(w') \leq \ell(w) + \langle \nabla \ell(w), w' - w\rangle + \frac{L}{2}\|w' - w\|^2. \tag{9.18}$$

for any two w, w' in the domain.

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

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

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

$$\ell(w') = \ell(w) + \int_0^1 \langle \nabla \ell(v), w' - w\rangle \;\mathrm{d}\lambda \tag{9.19}$$

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

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

Observe that

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

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

We can use the Descent Lemma twice on two points to w, w' to get (9.16). Another direct consequence of the Descent Lemma is the following corollary that relates the value $\ell(w)$ at any point w in the domain to that of the global minimum.

Corollary 9.2. For L-smooth convex function ℓ, if w^* is the global minimizer, then

$$\frac{1}{2L}\|\nabla \ell(w)\|^2 \le \ell(w) - \ell(w^*) \le \frac{L}{2}\|w - w^*\|^2. \tag{9.20}$$

Proof. Since $\nabla \ell(w^*) = 0$, the right-hand side follows directly from the Descent Lemma. To get the left-hand size, let us optimize the upper bound in the Descent Lemma using $w' = w + \lambda v$ with $\|v\| = 1$ as follows

$$\begin{aligned}\ell(w^*) = \inf_{w'} \ell(w') \le \inf_{w'} &\left\{\ell(w) + \langle \nabla \ell(w), w' - w\rangle + \frac{L}{2}\|w' - w\|^2\right\} \\ &= \inf_{\|v\|=1} \inf_{\lambda} \left\{\ell(w) + \lambda \langle \nabla \ell(w), v\rangle + \frac{L}{2}\lambda^2\right\} \\ &= \inf_{\|v\|=1} \left\{\ell(w) - \frac{1}{2L}\left(\langle \nabla \ell(w), v\rangle\right)^2\right\} \\ &= \ell(w) - \frac{1}{2L}\|\nabla \ell(w)\|^2.\end{aligned}$$

□

In other words, the gap between the function values $\ell(w) - \ell(w^*)$ is upper-bounded by the gap to the minimizer $\frac{L}{2}\|w - w^*\|^2$ and lower-bounded by the norm of the gradient $\frac{1}{2L}\|\nabla \ell(w)\|^2$.

Lemma 9.3 (Monotonic progress for gradient descent). For gradient descent $w^{t+1} = w^t - \eta \nabla \ell(w^t)$, if we pick the step-size

$$\eta \le \frac{1}{L} \tag{9.21}$$

we have

$$\ell(w^{t+1}) \le \ell(w^t) - \frac{\eta}{2} \|\nabla \ell(w^t)\|^2. \tag{9.22}$$

Further,

$$\ell(w^{t+1}) - \ell(w^*) \leq \frac{1}{2\eta}\left(\|w^t - w^*\|^2 - \|w^{t+1} - w^*\|^2\right) \tag{9.23}$$

which implies

$$\|w^{t+1} - w^*\|^2 \leq \|w^t - w^*\|^2. \tag{9.24}$$

Proof. Substitute $\eta \leq 1/L$ in the Descent Lemma and simplify to get (9.22). The second result is obtained by

$$\begin{aligned} 0 \leq \ell(w^{t+1}) - \ell(w^*) &\leq \ell(w^t) - \ell(w^*) - \frac{\eta}{2}\|\nabla\,\ell(w^t)\|^2 \\ &\leq \left\langle \nabla\,\ell(w^t), w^t - w^* \right\rangle - \frac{\eta}{2}\|\nabla\,\ell(w^t)\|^2 \\ &= \frac{1}{2\eta}\left(\|w^t - w^*\|^2 - \|w^t - w^* - \eta\,\nabla\,\ell(w^t)\|^2\right) \\ &= \frac{1}{2\eta}\left(\|w^t - w^*\|^2 - \|w^{t+1} - w^*\|^2\right). \end{aligned}$$

Observe that since the left-hand side is positive, the claim in (9.24) is true. □

We have therefore shown that if the step-size is not too large (the smoothness parameter of the function determines how large the step-size can be) then gradient descent always improves the value of the function with each iteration (9.22). It also improves the distance of the weights to the global minimum at each iteration (9.24).

Lemma 9.4 (Convergence rate for gradient descent, convex function). For gradient descent $w^{t+1} = w^t - \eta\,\nabla\,\ell(w^t)$ with step-size $\eta < 1/L$, we have

$$\ell(w^{t+1}) - \ell(w^*) \leq \frac{1}{2\,t\,\eta}\,\|w^0 - w^*\|^2. \tag{9.25}$$

Proof. We sum up the expression in (9.23) for all times t to get

$$\begin{aligned} \sum_{s=1}^{t} \ell(w^s) - \ell(w^*) &\leq \frac{1}{2\eta}\sum_{s=1}^{t}\left(\|w^{s-1} - w^*\|^2 - \|w^s - w^*\|^2\right) \\ &= \frac{1}{2\eta}\left(\|w^0 - w^*\|^2 - \|w^t - w^*\|^2\right) \\ &\leq \frac{1}{2\eta}\|w^0 - w^*\|^2. \end{aligned}$$

We know from (9.22) that $\ell(w^t)$ is non-increasing, so we can write

$$\ell(w^t) - \ell(w^*) \leq \frac{1}{t}\sum_{s=1}^{t}\left(\ell(w^s) - \ell(w^*)\right) \leq \frac{1}{2\,t\,\eta}\,\|w^0 - w^*\|^2.$$

□

If we want to find a weights with

$$\ell(w^t) - \ell(w^*) \leq \epsilon$$

for a convex function, we need to run gradient descent for at least

$$\mathcal{O}(1/\epsilon)$$

iterations. This is an important result to remember.

9.3.3 Gradient descent for strongly convex functions

Things are much better if the function we are minimizing is strongly convex. First we have the following lemma for strongly-convex functions which involves a rewriting co-coercivity condition for strongly convex functions.

Lemma 9.5 (Co-coercivity for strongly convex function). If $\ell(w)$ is m-strongly convex, and L-smooth, then the function $g(w) = \ell(w) - \frac{m}{2}\|w\|^2$ is convex and $L - m$-smooth. The co-coercivity condition for $\nabla g(w)$ can therefore be re-written as

$$\langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle \;\; \geq \frac{mL}{m+L}\|w - w'\|^2 + \frac{1}{m+L}\|\nabla \ell(w) - \nabla \ell(w')\|^2. \tag{9.26}$$

Proof. The convexity of $g(w)$ is immediate to see from the definition of strong convexity of $\ell(w)$. Use the monotonicity of the gradient of $g(w)$ to get

$$\begin{aligned} 0 &\leq \langle \nabla g(w) - \nabla g(w'), w - w' \rangle \\ &= \langle \nabla \ell(w) - \nabla \ell(w'), w - w' \rangle - m\|w - w'\|^2 \\ &\leq (L - m)\|w - w'\|^2. \end{aligned}$$

We can now rewrite the co-coercivity condition for $\nabla g(w)$ with the smoothness parameter $L - m$ and simplify to get (9.26). □

Lemma 9.6 (Convergence rate of gradient descent for strongly convex functions). For strongly convex functions we have pick a step-size

$$0 < \eta < \frac{2}{m+L}$$

to get

$$\|w^{t+1} - w^*\|^2 \leq \left(1 - \eta\frac{2mL}{m+L}\right)\|w^t - w^*\|^2. \tag{9.27}$$

which gives

$$\|w^t - w^*\|^2 \leq c^t \; \|w^0 - w^*\|^2 \tag{9.28}$$

where $c = \left(1 - \eta\frac{2mL}{m+L}\right)$.

Proof. We expand the left hand-side in (9.27) to get

$$\begin{aligned} \|w^{t+1} - w^*\|^2 &= \|w^t - \eta \nabla \ell(w^t) - w^*\|^2 \\ &= \|w^t - w^*\|^2 - 2\eta \left\langle \nabla \ell(w^t), w^t - w^* \right\rangle + \eta^2\|\nabla \ell(w^t)\|^2 \\ &\leq \left(1 - \eta\frac{2mL}{m+L}\right)\|w^t - w^*\|^2 + \eta\left(\eta - \frac{2}{m+L}\right)\|\nabla \ell(w^t)\|^2 \\ &\leq \left(1 - \eta\frac{2mL}{m+L}\right)\|w^t - w^*\|^2. \end{aligned}$$

We have substituted the co-coercivity condition from (9.26) for the inner-product with $w' := w^t$ and $w := w^*$ to get the first inequality. This implies that the distance to the global minimum $\|w^t - w^*\|$ decreases multiplicatively; compare this with (9.24) where the progress is additive. The additional assumption of strong convexity therefore means that we are making very quick progress towards the global minimum. We can use this inequality repeatedly for all iterations t to get

$$\|w^t - w^*\|^2 \leq c^t \ \|w^0 - w^*\|^2$$

where $c = \left(1 - \eta \frac{2mL}{m+L}\right)$. □

Strong convexity enables much faster progress towards the global minimum. If we want $\|w^t - w^*\| \leq \epsilon$ we need

$$\mathcal{O}(\log(1/\epsilon))$$

iterations of gradient descent. This is *much* less than that for a convex function. This is called *linear* convergence because we need a constant number of iterations to reduce the gap to the optimal in half. The naming convention is a bit unusual here but you will see that if we plot $\log \|w^t - w^*\|$ (or $\log\left(\ell(w^t) - \ell(w^*)\right)$) on the Y-axis and number of iterations t on the X-axis, we get a straight line for gradient descent on strongly-convex functions; you can see this from (9.28).

We say that the convergence rate of gradient descent for non-strongly convex functions is *sub-linear*. The longer we run GD for convex functions, the slower its progress.

Further, if we pick the largest step-size $\eta = 2/(m + L)$ we get

$$c = \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 < 1. \tag{9.29}$$

where $\kappa = L/m$ is the condition number of the Hessian (it is the ratio of the largest eigenvalue and the smallest eigenvalue). Larger the condition number κ, closer to 1 the multiplicative constant c and *slower* the convergence rate of gradient descent.

⚠ Plot the convergence rate of gradient descent for convex and strongly-convex functions.

⚠ In the optimization literature, an algorithm with

$$\lim_{t \to \infty} \frac{\ell(w^{t+1}) - \ell(w^*)}{\ell(w^t) - \ell(w^*)} = \rho$$

is said to be sub-linear if $\rho \in (0, 1)$, linear if $\rho = 1$ and super-linear if $\rho = 0$.

A few more points to note

1. The step-size if limited by $m + L$ but the convergence rate depends on $\kappa = L/m$. Smaller the value of c, faster the convergence.
2. Larger the L, smaller the ideal step-size η
3. You can get the upper bound

$$\ell(w^t) - \ell(w^*) \leq \frac{L}{2} \|w^t - w^*\|^2 \leq \frac{c^t L}{2} \|w^0 - w^*\|^2 \tag{9.30}$$

using (9.20).

You will also see the convergence rate written in many papers as

$$\|w^t - w^*\| \leq e^{-4t/\kappa} \ \|w^0 - w^*\|. \tag{9.31}$$

You can get this inequality by using the fact that $1 + x \leq e^x$ in (9.29). We can use this to pull out the dependence on κ in the convergence rate; for strongly convex functions, gradient descent requires

$$\mathcal{O}(\kappa \log(1/\epsilon))$$

iterations to reach within an ϵ-neighborhood of the global minimum $\ell(w^*)$. This suggests that smaller the condition number κ fewer the iterations required.

We can intuitively understand why convergence of gradient descent is slower for a large condition number. A large condition number means that some directions of the objective ℓ are highly curved while some others are very flat. It is difficult to pick one single scalar step-size in such situations that makes quick progress along the flat directions but also does not overshoot the highly curved directions. You might imagine that clever schemes to change the step-size depending upon the local geometry of the function $\ell(w^t)$ could help solve this issue and indeed it does, but such adaptive schemes are expensive to implement computationally. We will see some algorithms that pick different step-sizes for different weights in Chapter 11.

⚠ Draw a picture of this phenomenon for a quadratic objective $\ell(w) = \langle w, Aw \rangle$ for matrices $A \succ 0$ with different condition numbers κ.

9.4 Limits on convergence rate of first-order methods

It is a powerful and deep result that we cannot do better than a linear convergence rate for optimization methods that only use the gradient of the function $\ell(w)$. More precisely, for any first-order method, i.e., any method where the iterate at step t given by w^t is chosen to be

$$w^t \in w^0 + \text{span}\left\{\nabla \ell(w^0), \ldots, \nabla \ell(w^t)\right\},$$

we have the following theorem by Yurii Nesterov.

Theorem 9.7 (Nesterov's lower bound). If $w \in \mathbb{R}^p$, for any $t \leq (p-1)/2$ and every initialization of weights w^0 there exist functions $\ell(w)$ that are convex, differentiable, L-smooth with finite optimal value $\ell(w^*)$ such that any first-order method has

$$\ell(w^t) - \ell(w^*) \geq \frac{3}{32} \frac{L\|w^0 - w^*\|^2}{(t+1)^2}.$$

Let us read the statement of the theorem carefully. It states that *fixed* a time t and initial condition w^0, we can *find* a convex function $\ell(w)$ such that it takes gradient descent at least $\mathcal{O}(1/\epsilon^2)$ to reach an ϵ-neighborhood of the optimal value $\ell(w^*)$. The implication of this theorem is as follows. The convergence rate $\mathcal{O}(1/\epsilon)$ we obtained for convex functions is not the best rate we can get. Nesterov's lower bound suggests that there should be gradient-based algorithms that only require $\mathcal{O}(1/\sqrt{\epsilon})$ iterations. Such methods will be the topic of the next Chapter.

Chapter 10

Accelerated Gradient Descent

Reading

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

In the previous chapter we saw two results that characterize how many iterations gradient descent requires to reach within an ϵ-neighborhood of the global optimum for convex functions. If the function $\ell(w)$ is convex, GD with a step-size at most $1/L$ requires $\mathcal{O}(1/\epsilon)$ iterations. If the function $\ell(w)$ is strongly convex, then the step-size can be as large as $2/(m+L)$ and GD requires $\mathcal{O}(\kappa \log(1/\epsilon))$ iterations where

$$\kappa = \frac{L}{m}$$

is the condition number of the Hessian $\nabla^2 \ell(w)$. A large value of κ means that there are some directions where the function is highly curved and others where it is relatively flat. The main point to remember is that we often do not know a good value for m, L. Since the step-size of GD depends upon the curvature of the function; if the step-size is too large then GD overshoots on the highly curved directions and if the step-size is too small then GD makes slow progress along a direction.

We will study two algorithms in this chapter which accelerate the progress of gradient descent.

10.1 Polyak's Heavy Ball method

The most natural place to begin is to imagine gradient descent as a kinematic equation. Let w^t be the iterate of GD at time t, let us associate to it an auxiliary variable called the "velocity" v^t

$$v^t := \frac{w^{t+1} - w^t}{\eta}. \tag{10.1}$$

Gradient descent can then be written as

$$v^t = -\nabla \ell(w^t), \tag{10.2}$$

which allows us to think of the term $-\nabla \ell(w^t)$ is the force that acts on a particle to update its position from w^t to w^{t+1}. This particle has no inertia, so the force applied directly affects its position. If the magnitude of this gradient is small in a certain direction, the velocity is also small in that direction.

We now give our particle some inertia. Instead of the force directly affecting the position we will write down Newton's second law of motion ($F = ma$) for a particle with unit mass $m = 1$ as

$$\begin{aligned} -\nabla \ell(w^t) &=: \frac{v^{t+1} - v^t}{\eta} \\ &= \frac{1}{\eta}\left(w^{t+1} - 2w^t + w^{t-1}\right) \\ \Rightarrow w^{t+1} &= w^t - \eta \nabla \ell(w^t) + \left(w^t - w^{t-1}\right). \end{aligned} \tag{10.3}$$

Notice the third term on the right-hand side above, it is the gap between the current weights w^t and the previous weights w^{t-1}, if we have

$$\left\langle w^t - w^{t-1}, \nabla \ell(w^t) \right\rangle < 0,$$

i.e., the change from the previous time-step is along the descent direction, then the weights w^{t+1} get an extra boost. If instead, the change from the previous direction is not along the gradient descent direction, then the third term reduces the magnitude of the gradient. The third term is effectively the inertia of gradient updates. This method is therefore called Polyak's Heavy Ball method.

We give ourselves some more control over how inertia enters the update equation using a hyper-parameter ρ

$$w^{t+1} = w^t - \eta \nabla \ell(w^t) + \rho\left(w^t - w^{t-1}\right). \tag{10.4}$$

If $\rho = 0$, we do not use any inertia and Polyak's method boils down to gradient descent. Typically, we choose $\rho \in (0, 1)$. This inertia is called *momentum* in the optimization literature and ρ is called the momentum coefficient.

Polyak's method is simple yet very powerful. In the previous chapter, we showed a lower-bound of Nesterov which indicates that first-order optimization algorithm (that only depends on the gradient of the objective) cannot be faster than $\mathcal{O}(1/\sqrt{\epsilon})$. It turns out that Polyak's method converges at this rate, i.e., if we want

$$\|w^t - w^*\| \leq \epsilon$$

we need to run Polyak's Heavy Ball method for $\mathcal{O}(1/\sqrt{\epsilon})$ iterations for convex functions. If the function is strongly convex, the number of iterations comes down to

$$\mathcal{O}(\sqrt{\kappa}\log(1/\epsilon)).$$

Both of these are improvements upon their convergence rates for gradient descent. These improvements are also quite a lot, we need quadratically fewer iterations than gradient descent in Polyak's method and the only incremental cost of doing so is that we have to maintain a copy of the weights w^{t+1} while implementing the updates in (10.4).

An alternative way to write Polyak's updates We can rewrite the updates in (10.4) using a dummy variable u^t as

$$\begin{aligned} u^t &= (1+\rho)w^t - \rho\, w^{t-1} \\ w^{t+1} &= u^t - \eta \nabla \ell(w^t). \end{aligned} \tag{10.5}$$

This is how these updates are implemented in PyTorch. This is convenient: effectively, the code needs to maintain only the difference $u^t = (1+\rho)w^t - \rho w^{t-1}$ in a buffer u^t and subtract the gradient $\nabla \ell(w^t)$ from this update to result in the new updates. GD can be implemented with a simple change by setting $u^t := w^t$. *The dummy variable is initialized to* $u^0 = w^0$.

A yet another way to write Polyak's updates We can also rewrite the updates in (10.5) as

$$\begin{aligned} u^{t+1} &= \rho\, u^t - \nabla \ell(w^t) \\ w^{t+1} &= w^t + \eta\, u^{t+1}. \end{aligned} \tag{10.6}$$

This set of updates brings out idea of momentum more clearly. The variable u^t in this case is exactly the velocity v^t that we have seen above except that it is updated slightly different than our expression $(F = ma)$ in the first equation. The first term

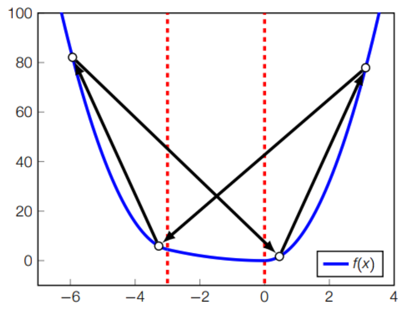
$$u^{t+1} = \rho\, u^t - \nabla \ell(w^t)$$

reduces the velocity u^t by a factor ρ before adding the gradient to it.

⚠ Draw Polyak's updates for a two-dimensional function.

10.1.1 Polyak's method can fail to converge

The caveat with relying on the inertia of the particle to make progress is that near the global minimum, when the iterates overshoot the global minimum, the inertia is often very different from the gradient. Polyak's method can become unstable and can result in oscillations under such conditions, e.g.,



However it is a very simple method to accelerate gradient descent and works great in practice.

10.2 Nesterov's method

Nesterov's method is an advanced version of Polyak's method and removes the oscillations towards the end. Let us understand these oscillations better. We saw that incorporating a notion of inertia in Polyak's method gave us accelerated convergence; this is intuitive, if the velocity is along the descent direction the particle descends faster. The same inertia hurts towards the end because the velocity can be very different than the gradient when the particle overshoots the minimum.

A simple way to fix this is to incorporate damping (friction) into Newton's law of motion; you can read about the simple harmonic oscillator at https://en.wikipedia.org/wiki/Harmonic_oscillator. We are going to write

$$ma = F - cv.$$

where m is the mass, c is the coefficient of damping, a and v are acceleration and velocity respectively and F is the force as usual. The effective force decreases with the velocity. Doing so does not slow down the weight updates much when the gradient magnitude is large at the beginning of training. But when the gradient magnitude is small (when the particle is in the neighborhood of the global minimum), this friction prevents excessive overshooting.

With that background, let us see how Nesterov's updates for gradient descent look.

We will write a similar set up of updates as that of (10.6). Nesterov's updates correspond to

$$\begin{aligned} u^{t+1} &= \rho\, u^t - \nabla\, \ell(w^t + \eta\, \rho\, u^t) \\ w^{t+1} &= w^t + \eta\, u^{t+1}. \end{aligned} \tag{10.7}$$

The only difference between (10.7) and (10.6) is the term in blue; effectively the gradient is computed as if the weights w^t moved using the velocity u^t; and then this new velocity u^{t+1} is used to obtain the new updates w^{t+1}. Nesterov's method solves the problem of instability in Polyak's method by essentially computing the gradient (the blue term) as given by the current velocity. You can see how this *slows down* the updates if the velocity is well-aligned with the gradient; we are reducing the benefit of inertia/momentum. However, doing so prevents overshooting as the iterates reach the neighborhood of the global minimum.

An alternative way to write Nesterov's updates We can rewrite the updates in (10.7) to look like those in (10.5), to get

$$\begin{aligned} u^t &= (1+\rho)w^t - \rho\, w^{t-1} \\ w^{t+1} &= u^t - \eta \nabla\, \ell(u^t). \end{aligned} \tag{10.8}$$

Again the blue term is the only difference between Polyak's method and Nesterov's method. The term u^t is conceptually a forecasted version of the weights w^t because notice that

$$u^t = w^t + \rho(w^t - w^{t-1}).$$

The new weights w^{t+1} are now obtained by thinking of u^t as the actual weights. We initialize $w^{t+1} = w^t$ to w^0 for $t = 0$.

10.2.1 Yet another way to write Nesterov's updates

We now tie back Nesterov's updates to our introductory narrative on friction. We will set the damping coefficient (ρ) in (10.8) to a special value

$$\rho = \frac{t-1}{t+2};$$

effectively as $t \to \infty$ the friction becomes larger and larger. This simplifies our updates to

$$\begin{aligned} u^t &= w^t + \frac{t-1}{t+2}\left(w^t - w^{t-1}\right) \\ w^{t+1} &= u^t - \eta\, \nabla\, \ell(u^t). \end{aligned}$$

which upon collapsing together give

$$w^{t+1} - w^t = \frac{t-1}{t+2}\left(w^t - w^{t-1}\right) - \eta\, \nabla\, \ell(u^t). \tag{10.9}$$

We now choose a different way of interpreting these updates. We will imagine that the sequence

$$\{w^0, w^1, \ldots, w^t, w^{t+1}, \ldots\}$$

is the discretization of a smooth curve

$$\{W(\tau) : \tau \in [0, \infty)\}.$$

How is this curve $W(\tau)$ related to the original sequence? We are going to study the updates under the setting

$$\tau := \sqrt{\eta}\, t. \tag{10.10}$$

Essentially the time of the discrete-time updates (10.9) increments in intervals of 1, but the time of the curve $W(\tau)$ is real-number. Each increment in discrete-time corresponds to $\sqrt{\eta}$ increment of the time τ for the curve $W(\tau)$. This gives

$$\begin{aligned} W(\tau) &= w^t \\ W(\tau + \sqrt{\eta}) &= w^{t+1}. \end{aligned}$$

We now do a Taylor expansion for the continuous curve $X(\tau)$ to get

$$\begin{aligned} w^{t+1} - w^t &= W(\tau + \sqrt{\eta}) - W(\tau) \\ &= \dot{W}(\tau)\sqrt{\eta} + \frac{1}{2}\ddot{W}(\tau)\eta + o(\sqrt{\eta}). \end{aligned} \tag{10.11}$$

Here

$$\dot{W}(\tau) = \frac{\mathrm{d}}{\mathrm{d}\tau}W(\tau), \quad \ddot{W}(\tau) = \frac{\mathrm{d}^2}{\mathrm{d}\tau^2}W(\tau)$$

are the first and second derivative of the curve with respect to time and $o(\sqrt{\eta})$ denotes higher-order terms. Similarly

$$\begin{aligned} w^t - w^{t-1} &= W(\tau) - W(\tau - \sqrt{\eta}) \\ &= \dot{W}(\tau)\sqrt{\eta} - \frac{1}{2}\ddot{W}(\tau)\eta + o(\sqrt{\eta}). \end{aligned}$$

Note that due to our special scaling of time we have

$$\frac{t-1}{t+2} = 1 - \frac{3}{t+2} \approx 1 - \frac{3}{t} = 1 - \frac{3\sqrt{\eta}}{\tau}.$$

We now do a Taylor expansion of the loss term $\nabla \ell(u^t)$ to get

$$\begin{aligned} \nabla \ell(u^t) &= \nabla \ell\left(w^t + \frac{t-1}{t+2}\left(w^t - w^{t-1}\right)\right) \\ &= \nabla \ell(w^t) + \text{higher order terms} \\ &= \nabla \ell(W(\tau)) + o(\sqrt{\eta}). \end{aligned} \tag{10.12}$$

Substitute (10.11) and (10.12) in (10.9) to get

$$\begin{aligned} \dot{W}(\tau) + \frac{1}{2}\ddot{W}(\tau)\sqrt{\eta} + o(\sqrt{\eta}) = \left(1 - \frac{3\sqrt{\eta}}{\tau}\right)\left(\dot{W}(\tau) - \frac{1}{2}\ddot{W}(\tau)\sqrt{\eta} + o(\sqrt{\eta})\right) \\ - \sqrt{\eta}\ \nabla \ell(W(\tau)) + o(\sqrt{\eta}). \end{aligned}$$

This equation is true for all values of η, so we can compare the coefficients of $\sqrt{\eta}$ on both sides to get

$$\ddot{W} + \frac{3}{\tau}\dot{W} + \nabla \ell(W) = 0. \tag{10.13}$$

This equation looks very similar to Newton's law with friction $ma + cv = F$. Again, the term $\nabla \ell(W)$ is acting as the force, the second derivative $\ddot{W}$ is the acceleration and the friction term $\frac{3}{t}\dot{W}$ increases with velocity. We have shown that for a particularly chosen value of the momentum coefficient, Nesterov's updates result in an ordinary differential equation that looks much like that simple harmonic oscillator that most of you have seen before in high-school. This approach gives an alternative, and very simple, way of understanding Nesterov's updates which is nice because the updates in (10.7) and (10.8) were quite non-intuitive and created by Nesterov through a sheer *tour de force*.

Remark 10.1. Derive a similar ordinary differential equation for Polyak's updates using the same setting of friction $(t-1)/(t+2)$ as that in (10.9). You will notice that if viewed in continuous-time Polyak's updates are exactly the same as Nesterov's updates. This suggests that that the continuous-time construct is a more abstract point-of-view and eliminates the subtle differences between the updates between the two algorithms.

Such continuous-time constructs are however very useful to understand what these updates actually do, e.g., we know that Nesterov's (and Polyak's) updates correspond to having friction in Newton's law which is not apparent by looking at the equations in (10.8). It is also very easy to obtain the convergence rate of the continuous-time version; it is an ordinary differential equation and we can use a lot of tools from dynamical systems, in particular Lyapunov functions. It will amuse you to know that obtaining the convergence rate for Nesterov's updates using the continuous-time version (10.13) takes about half a page.

10.2.2 How to pick the momentum parameter?

Nesterov's updates converge at a rate that is similar to that of Polyak's updates. For convex functions, we need

$$\mathcal{O}(1/\sqrt{\epsilon})$$

iterations to get within the ϵ-neighborhood of the global minimum if we set

$$\rho = (t-1)/(t+2)$$

in (10.6). If we are minimizing a strongly-convex function we can pick the momentum coefficient to depend on m, L: we can set

$$\rho = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} \tag{10.14}$$

and $\eta < 2/(m+L)$. Doing so entails that we need only

$$\mathcal{O}(\sqrt{\kappa}\log(1/\epsilon))$$

weight updates to reach within an ϵ-neighborhood of the global minimum. The expression in (10.14) gives some insight in how momentum accelerates things. If $\kappa \approx 1$, i.e., the Hessian of the objective is well-conditioned without a big diversity in the curvature in different directions, we do not really need friction $\rho \approx 0$ to avoid overshooting close to the minimum; progress in all directions is balanced. On the other hand, if $\kappa \gg 1$, the objective is badly conditioned and the friction coefficient $\rho \approx 1$ should be large to avoid overshooting near the global minimum.

How to pick ρ in practice? If we know what m, L are for a given problem (you will see an example of this in HW 4), it is straightforward to pick the momentum coefficient and get accelerated convergence of gradient descent. If we do not know m, L, we pick some constant value of ρ. For instance, $\rho = 0.9$ is popularly used in most deep learning libraries. Typically, the momentum coefficient is not increased with the number of weight updates using $(t-1)/(t+2)$. You will some heuristics for modifying the momentum coefficient in this week's recitation.

Chapter 11

Stochastic Gradient Descent

Reading

1. "Stochastic gradient descent tricks" by Bottou (2012). Great paper with lots of little tricks of how to use SGD in practice.
2. Up to Section 4.2 of "Optimization methods for large-scale machine learning" by Bottou et al. (2018). This is advanced material, you do not need to be able to follow it completely.
3. http://fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html
4. Stochastic Weight Averaging (SWA) by Izmailov et al. (2018).

Stochastic Gradient Descent (SGD) has its roots in stochastic optimization. A stochastic optimization problem looks like

$$w^* = \underset{w}{\text{argmin}} \, \underset{\xi}{\mathbb{E}}[\ell(w, \xi)] \tag{11.1}$$

where ξ is a random variable. This is a very old and rich area, there was lots of action in it already in the 1950s, e.g., (Kushner and Yin, 2003; Robbins and Monro, 1951). It is also a highly relevant problem: for instance, when a plane goes from Los Angeles to Philadelphia, the route that the plane takes depends on the local weather conditions along its path and airlines will optimize this route using a stochastic optimization problem of the above form. The variable w will be the trajectory of the plane and ξ are the weather conditions which we do not know exactly but may perhaps have estimated a distribution for them. Such problems are very common in other fields like operations research, e.g., optimizing the time at which an Amazon package arrives with various disturbances such as delays in shipping, missing inventory in the warehouse etc.

In machine learning, we are interested in solving a slightly different problem called the finite-sum problem. Given a finite dataset $D = \left\{(x^i, y^i)\right\}_{i=1,\dots,n}$

we minimize

$$\ell(w) := \frac{1}{n}\sum_{i=1}^{n} \ell^i(w) \tag{11.2}$$

where we will use the shorthand

$$\ell^i(w) := \ell(w; x^i, y^i)$$

to denote the loss on the datum (x^i, y^i) with weights w. Essentially, the random variable ξ in (11.1) represents the samples in the training dataset; with important differences being that neither do we know anything about the distribution of the input data, nor do we have an infinite number of samples.

It is difficult to do gradient descent if the number of samples n is large because the gradient is a summation of a large number of terms

$$\nabla \ell(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla \ell^i(w).$$

If the mini-batch size is 1, i.e., at each iteration we sample one of the training samples denoted by

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

we update the weights using

$$w^{t+1} = w^t - \eta \nabla \ell^{\omega_t}(w^t). \tag{11.3}$$

For a larger mini-batch of size $\mathscr{b}$ let us denote the samples in the mini-batch by

$$\left\{(x^{\omega_t^1}, y^{\omega_t^1}), \ldots, (x^{\omega_t^{\mathscr{b}}}, y^{\omega_t^{\mathscr{b}}})\right\}$$

where each $\omega_t^k \in \{1, \ldots, n\}$ is the index chosen uniformly randomly from the training dataset. We will choose these indices with replacement (analyzing SGD for sampling without replacement is quite hard). The gradient on this sampled mini-batch is denoted by

$$\nabla \ell_{\mathscr{b}}(w) := \frac{1}{\mathscr{b}}\sum_{i=1}^{\mathscr{b}} \ell^{\omega_t^i}(w) \tag{11.4}$$

and we update the weights as usual using

$$w^{t+1} = w^t - \eta \nabla \ell_{\mathscr{b}}(w^t).$$

If $\mathscr{b} = 1$, we will denote the gradient by $\nabla \ell_\omega$ to keep the notation clear.

What is an epoch in PyTorch? We will not think of epochs when we develop the theory for SGD. An epoch is a construct introduced in deep learning libraries for bookkeeping purposes. It also ensures that if Algorithm A obtains so and so training/validation error after 100 epochs, it can be compared directly with Algorithm B which obtains the same training/validation error after, say, 120 epochs, e.g., one can say Algorithm A is faster than Algorithm B at training a network. Instead of sampling a mini-batch of data uniformly randomly with replacement, PyTorch shuffles the entire training set at the beginning of each epoch and samples the mini-batch *with replacement* during each epoch. This is reasonable but there will be some discrepancies in the performance of SGD as predicted by theory and obtained by PyTorch on deep networks, especially if the mini-batch size is large.

Although we will not discuss this, SGD using mini-batches sampled with replacement is faster than with mini-batches sampled without replacement (Recht and Ré, 2012).

11.1 SGD for least-squares regression

Let us understand SGD for one dimensional least-squares, our data and targets are $x^i, y^i \in \mathbb{R}$ and the objective is

⚠ Draw the objective here for different values of w^i and understand how SGD works for this problem.

$$\ell(w) = \frac{1}{2n} \sum_{i=1}^{n} (x^i w - y^i)^2 \tag{11.5}$$

for the weights $w \in \mathbb{R}$. Notice that the objective is a sum of n different quadratics, each quadratic is minimized by *different* weights

$$w^*(i) := \frac{y^i}{x^i};$$

in other words, each sample in the training dataset would like the weight to be y^i/x^i to minimize its residual and the least-squares objective which sums up their individual residuals forces them to made trade-offs. Focus on two quantities

$$w_{\text{min}} = \min_i \{w^*(i)\}, \quad w_{\text{max}} = \max_i \{w^*(i)\}.$$

Notice that the interval $(-\infty, w_{\text{max}})$ is the region where the descent direction on any sample in the dataset moves the weights w^t to the right. The interval (w_{max}, ∞) is the region where the descent direction on any sample moves the weights to the left. If weights are initialized in the latter region, $w^0 \gg \max_i w^*(i)$, successive iterations of SGD will quickly bring the weights to

$$w^t \in (w_{\text{min}}, w_{\text{max}}) \tag{11.6}$$

which we will call the "zone of confusion". Similarly, if weights are initialized $w^o \ll w_{\text{min}}$, they will move right until iterates reach the zone of confusion.

After $w^t \in (w_{\text{min}}, w_{\text{max}})$, there is no real convergence of the weights, if the learning rate η is fixed, since the samples ω_t are sampled uniformly randomly, depending upon which sample was chosen to compute the gradient the weights move to the right or the left and therefore keep shuttling back and forth in this region.

Notice that the objective in (11.5) is convex because it is the sum of convex functions so there is a unique global minimum namely

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

If one were to execute gradient descent on this same problem $w^{t+1} = w^t - \eta \nabla \ell(w^t)$, we will converge to this value. But since SGD samples a different sample at each iteration, SGD never converges, it remains in this large zone $(w_{\text{min}}, w_{\text{max}})$.

11.2 Convergence of SGD

If the learning rate is large, SGD makes quick progress outside the zone of confusion but bounces around a lot inside the zone of confusion. If the learning rate is too small, SGD is slow outside the zone of confusion but does not bounce around too much inside the zone. You can explore how the learning rate changes the dynamics of SGD at
http://fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html.

In this section, we will study under what conditions SGD converges to the global minimum and how the learning rate of SGD should be reduced to make it converge quickly. We will first analyze SGD with mini-batch size of 1.

Strongly convex functions The proofs for convex functions are tedious, so we will only consider strongly convex functions in this section. As usual the strong convexity parameter is m and smoothness parameter is L. One key thing to notice that these that constants L, m refer to the full objective, i.e.,

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

and

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

Here $\ell(w)$ is the *full objective*

$$\ell(w) = \frac{1}{n}\sum_{i=1}^{n} \ell^i(w).$$

What is the appropriate notion of convergence? The key difference between updates of SGD and those of GD is that SGD updates also depend on the random variable ω_t. The iterate ω_t is a *random variable* and therefore instead of simply bounding the gap $\ell(w^t) - \ell(w^*)$ we will have to obtain an upper bound for

$$\mathop{\mathbb{E}}_{w^t}\left[\ell(w^t)\right] - \ell(w^*).$$

Similar to the case of SGD, let us construct a descent lemma for one iteration of SGD update.

Lemma 11.1 (Descent Lemma for SGD).

$$\mathop{\mathbb{E}}_{\omega_t}\left[\ell(w^{t+1}) - \ell(w^t) \mid w^t\right] \leq -\eta \left\langle \nabla \ell(w^t), \mathop{\mathbb{E}}_{\omega_t}\left[\nabla \ell^{\omega_t}(w^t)\right] \right\rangle + \frac{L\eta^2}{2} \mathop{\mathbb{E}}_{\omega_t}\left[\|\nabla \ell^{\omega_t}(w^t)\|^2\right]. \tag{11.7}$$

Proof. First, compare this with the descent lemma for gradient descent (if we substitute $w^{t+1} - w^t = -\eta \nabla \ell(w^t)$ from Chapter 9)

$$\ell(w^{t+1}) - \ell(w^t) \leq -\eta \left\langle \nabla \ell(w^t), \nabla \ell(w^t) \right\rangle + \frac{L\eta^2}{2} \|\nabla \ell(w^t)\|^2$$

The only difference now is that in the case of SGD we have

$$w^{t+1} - w^t = -\eta \nabla \ell^{\omega_t}(w^t).$$

The most important different however is that the descent term, namely the left-hand side in (11.7) is conditioned on the random variable w^t. The proof of this lemma is easy, we simply substitute the expression for the weight updates of SGD and take an expectation over the index of datum sampled by SGD ω_t on both sides of the inequality. □

The implication of the above lemma is that SGD updates need more refined conditions under which we can claim monotonic progress towards the global minimum. Effectively, we need to make sure that the right-hand side is negative, *always* irrespective of what the value of the random variable w^t is. We would like to upper bound the right-hand side by a deterministic quantity ideally.

11.2.1 Typical assumptions in the analysis of SGD

1. **Stochastic gradients are unbiased.** Assume that the stochastic gradient is unbiased
$$\nabla \ell(w) = \mathop{\mathbb{E}}_{\omega}\left[\nabla \ell^{\omega}(w)\right] \tag{11.8}$$
for all w in the domain. This is akin to assuming that the way we sample images in the mini-batch is such that the average is always pointing towards the true gradient with a similar magnitude. This is a natural condition and will only change if the sampling distribution is not uniform. This assumption allows to control the first term in the descent lemma.

2. **Second moment of gradient norm does not grow too quickly.** We will assume that there exist scalars σ_0 and σ such that
$$\mathop{\mathbb{E}}_{\omega_t}\left[\|\nabla \ell^{\omega}(w)\|^2\right] \leq \sigma_0 + \sigma \|\nabla \ell(w)\|^2. \tag{11.9}$$
This assumption allows to control the second term in the descent lemma for SGD. It assumes that the stochastic estimate of the gradient in SGD $\nabla \ell^{\omega_t}(w)$ is not too different than the full gradient $\ell(w^t)$. In

the neighborhood of a critical point (locations where the full gradient $\nabla \ell(w) = 0$), the stochastic gradient is allowed to grow in a similar fashion as the true gradient except with a scaling factor $\sigma > 0$ and a constant σ_0.

Let us see how the descent lemma changes with these additional assumptions.

Lemma 11.2 (Descent Lemma for SGD with additional assumptions). If SGD gradients are unbiased and the second moment of the stochastic gradients can be bounded, we have

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\omega_t}\left[\ell(w^{t+1}) - \ell(w^t) \mid w^t\right] \\
&\leq -\eta \left\langle \nabla \ell(w^t), \mathop{\mathbb{E}}_{\omega_t}\left[\nabla \ell^{\omega_t}(w^t)\right] \right\rangle + \frac{L\eta^2}{2} \mathop{\mathbb{E}}_{\omega_t}\left[\|\nabla \ell^{\omega_t}(w^t)\|^2\right] \\
&\leq -\eta \|\nabla \ell(w^t)\|^2 + \frac{L\eta^2}{2} \mathop{\mathbb{E}}_{\omega_t}\left[\|\nabla \ell^{\omega_t}(w^t)\|^2\right] \\
&= -\eta \left(1 - \frac{\eta L \sigma}{2}\right) \|\nabla \ell(w^t)\|^2 + \frac{\eta^2 L \sigma_0}{2}.
\end{aligned} \tag{11.10}$$

The proof is given in (11.10) itself. Compare this to the corresponding result we have derived for gradient descent in Chapter 9

$$\ell(w^{t+1}) - \ell(w^t) \leq -\frac{\eta}{2}\|\nabla \ell(w^t)\|^2.$$

In addition to the negative term $-\frac{\eta}{2}\|\nabla \ell(w^t)\|^2$, we have two additional positive terms

$$\frac{\eta^2 L \sigma}{2}\|\nabla \ell(w^t)\|^2 + \frac{\eta^2 L \sigma_0}{2};$$

this indicates that depending upon the magnitude of these terms we may not get monotonic improvement of the objective for SGD. There is no such concern for gradient descent, we get monotonic progress at all parts of the domain.

> We need to pick the learning rate η in such a way that balances the the right-hand side of (11.10) and makes it negative.

11.2.2 Convergence rate of SGD for strongly-convex functions

Theorem 11.3 (Optimality gap for SGD). If we pick a step-size

$$\eta \leq \frac{1}{L\sigma}$$

for m-strongly convex and L-smooth function $\ell(w)$ then the expected optimality gap satisfies

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\omega_1, \omega_2, \ldots, \omega_t}\left[\ell(w^{t+1})\right] - \ell(w^*) \\
&\leq \frac{\eta L \sigma_0}{2m} + (1 - \eta m)^t \left(\ell(w^0) - \ell(w^*) - \frac{\eta L \sigma_0}{2m}\right).
\end{aligned} \tag{11.11}$$

We will not cover the proof of this theorem, it is a direct application of the descent lemma. See Bottou et al. (2018, Theorem 4.6) for an elaborate proof.

This theorem beautifully demonstrates the interplay between the step-size and and the variance of SGD gradients. If there is no stochasticity, i.e., $\sigma_0 = 0$ and $\sigma = 1$, we get the same result as that of gradient descent, namely, the function value $\ell(w^{t+1})$ converges at a *linear rate* $(1 - \eta m)^t$. Some points to notice

1. The random variable w^{t+1} depends upon all the indices $\omega_1, \omega_2, \ldots, w_t$ that were sampled during updates of SGD and therefore the expectation in (11.11) should be over all these random variables.
2. When the stochastic gradient is noisy, we have a non-zero σ_0 we can no longer get to the global minimum, there is a first term which does not decay with time.
3. If we pick a small η, we get closer to the global minimum but go there quite slowly. On the other hand, we can pick a large η and get to a neighborhood of the global minimum quickly but we will then have a large error leftover at the end.

How can we make SGD converge and drive down the first term in (11.11) to zero? A simple trick is to reduce the learning rate η with time. We do not want to decay the learning rate too quickly however because the second term in (11.11) is small, i.e., optimality gap is reduced quickly by its multiplicative nature, for a large value of the learning rate. A good schedule to pick is

$$\sum_{t=1}^{\infty} \eta_t = \infty, \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \tag{11.12}$$

Heuristic for training neural networks The two terms in the convergence rate of SGD explain the widely used heuristic of "divide the learning rate by some constant" if the training error seems plateaued. We are reducing the size of the ball in which SGD iterates bounce around by doing so.

Theorem 11.4 (Convergence rate of SGD for decaying step-size). For a schedule

$$\eta_t = \frac{\beta}{t + t_0} \text{ where } \beta > \frac{1}{m} \text{ and } t_0 \text{ is such that } \eta_1 < \frac{1}{L\sigma}$$

then the expected optimality gap satisfies

$$\mathop{\mathbb{E}}_{\omega_1, \omega_2, \ldots, \omega_t} \left[\ell(w^{t+1})\right] - \ell(w^*) = \mathcal{O}\left(\frac{1}{t + t_0}\right). \tag{11.13}$$

We will not do the proof. If you are interested, see Bottou et al. (2018, Theorem 4.7). Compare this to the convergence rate of $\mathcal{O}(\kappa \log(1/\epsilon)$ for gradient descent for strongly-convex functions. Notice that we converge only at a sub-linear rate for SGD even for strongly convex loss functions. SGD is a much slower algorithm than GD.

Convergence rate for mini-batch SGD The mini-batch gradient $\nabla \ell_{\mathcal{b}}(w)$ is still an unbiased estimate of the full-gradient

$$\mathop{\mathbb{E}}_{\mathcal{b}}\left[\nabla \ell_{\mathcal{b}}(w)\right] = \nabla \ell(w)$$

but the second assumption in SGD improves a bit. Since the mini-batch gradient is averaged over $\mathcal{b}$ samples we have

$$\mathop{\mathbb{E}}_{\mathcal{b}}\left[\|\nabla \ell_{\mathcal{b}}(w)\|^2\right] \leq \frac{\sigma_0}{\mathcal{b}} + \frac{\sigma}{\mathcal{b}}\|\nabla \ell(w)\|^2$$

if σ_0, σ were the constants in (11.9). This changes the convergence rate in Theorem 11.3 to

$$\begin{aligned} &\mathop{\mathbb{E}}_{\omega_1,\omega_2,\ldots,\omega_t}\left[\ell(w^{t+1})\right] - \ell(w^*) \\ &\leq \frac{\eta L \sigma_0}{2m\mathcal{b}} + (1-\eta m)^t \left(\ell(w^0) - \ell(w^*) - \frac{\eta L \sigma_0}{2m\mathcal{b}}\right). \end{aligned} \tag{11.14}$$

Note that the maximum learning rate in Theorem 11.3 is inversely proportional to σ so we can also pick a larger learning rate $\eta < \frac{\mathcal{b}}{L\sigma}$. If we do so, the first and last terms above are not affected by the batch-size but multiplicative term $(1-\eta m)^t$ is. Since

$$(1-\eta m)^t \leq e^{-tm\eta}$$

we see that increasing the learning rate by a factor of $\mathcal{b}$ will reduce the number of iterations required to reach the zone of confusion by a factor of $\mathcal{b}$. Of course, this comes with the caveat that each iteration also requires $\mathcal{O}(\mathcal{b})$ more computation to compute the gradient compared to single-sample SGD.

11.2.3 When should one use SGD in place of GD?

Theorem 11.4 indicates that SGD is a very slow algorithm, GD is much faster than SGD to minimize strongly convex functions. This gap also exists if we do not have strong convexity: we did not prove this but SGD requires $\mathcal{O}(1/\epsilon^2)$ to reach an ϵ-neighborhood of the global optimum for convex functions whereas GD requires a much smaller $\mathcal{O}(1/\epsilon)$. One might wonder why we should use SGD at all.

It is critical to remember that the objective in machine learning is a sum of many terms

$$\ell(w) = \frac{1}{n}\sum_{i=1}^{n} \ell^i(w)$$

One iteration of SGD requires us to compute only $\nabla \ell^{\omega_t}(w)$ whereas one update of GD requires us to compute the full gradient $\nabla \ell(w)$. One weight update of GD is $\mathcal{O}(n)$ more expensive than one weight update using SGD. Let us do a back-of-the-envelope computation for convex functions. If we want to reach an ϵ-neighborhood of the global minimum of a convex function, we need $\mathcal{O}(1/\epsilon)$ iterations of GD, which requires

$$\mathcal{O}\left(\frac{n}{\epsilon}\right)$$

operations. SGD needs $\mathcal{O}(1/\epsilon^2)$ iterations and therefore requires

$$\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$$

operations to reach the ϵ-neighborhood. This indicates that if our chosen ϵ-ball is

$$\epsilon \lessapprox \frac{1}{n}$$

GD requires fewer overall operations. But if ϵ-ball is larger than this, we should use SGD because it is computationally cheaper.

SGD is particularly suited to machine learning compared to GD for the following reason. Let $\epsilon^i = \ell^i(w^t) - \ell^i(w^*)$ be the residual on the i^{th} datum in the training dataset. Observe that our ϵ-neighborhood is

$$\epsilon = \ell(w^t) - \ell(w^*) = \frac{1}{n} \sum_{i=1}^{n} \epsilon^i.$$

If ϵ^i is constant and does not depend on the number of training samples n (i.e., say we are happy with the average error over the training dataset being 2% even and do not seek a smaller one even if we collect more data) then we should use SGD to train our model because it is cheaper. This is not always the case for other problems, e.g., if you are doing computational tomography (capturing multiple images from a CT-scan machine and trying to reconstruct the heart/lung region in the thoraric cavity), we may seek a more and more accurate answer, i.e., small ϵ if we have more data.

11.3 Accelerating SGD using momentum

The convergence rate of SGD is quite bad, it is sub-linear. Roughly speaking, the successive iterates of SGD are computed using different mini-batches; the gradient on each such mini-batch is a noisy approximation of the full-gradient on the training dataset (that of GD). This makes the SGD iterates noisy and one may improve the convergence rate of SGD by simply averaging the weights. This leads to a simple technique to accelerate SGD which we discuss next.

Polyak-Ruppert averaging Consider the updates

$$\begin{aligned} w^{t+1} &= w^t - \eta_t \nabla \ell_{\omega_t}(w^t) \\ u^t &= \frac{w^0 + w^1 + \cdots + w^t}{t}. \end{aligned} \tag{11.15}$$

In a series of papers, Polyak (1990); Polyak and Juditsky (1992); Ruppert (1988) showed that the quantity

$$\mathop{\mathbb{E}}_{\omega_1,\ldots,\omega_{t-1}} \left[\ell(u^t)\right] - \ell(w^*)$$

converges faster than the quantity

$$\mathop{\mathbb{E}}_{\omega_1,\ldots,\omega_{t-1}} \left[\ell(w^t)\right] - \ell(w^*);$$

both of these still converge at rate $\mathcal{O}(1/\epsilon^2)$ but the former has a smaller constant. This is quite surprising and useful: essentially we are still performing mini-batch updates for the weights w^t but instead of thinking of w^t as the answer, we think of u^t as the output of SGD. This averaging of iterates does not change the SGD algorithm. Computing this output requires us to remember all the past iterations $w^0, \dots, w^t$ but we can easily approximate that step by exponential averaging of the *weights*

$$u^t = \rho\, u^{t-1} + (1-\rho)\, w^t;$$

exponential averaging is likely to achieve the same purpose with a much smaller memory requirement.

Further, this idea of using averaged iterates to speed up stochastic optimization algorithms is quite general and also works for algorithms other than SGD. The paper on Stochastic Weight Averaging by Izmailov et al. (2018) performs weight averaging (with quite different motivations) and works very well in practice.

11.3.1 Momentum methods do not accelerate SGD

We saw that momentum is very useful to accelerate the convergence of gradient descent. The power of momentum lies in making faster progress using the inertia of the particle: if the velocity and the current gradient are aligned with each other (as is the case at the beginning of training when the iterates are far from the global optimum) momentum speeds up things. Towards the end of training when gradients are typically mis-aligned with the velocity, we need friction (as in Nesterov's updates) to reduce this effect.

Observe that in SGD, the gradient is *always* incorrect; it is after all only a noisy stochastic approximation of the full gradient on the dataset. Since the velocity $w^t - w^{t-1}$ was computed using the previous stochastic gradient, there is no reason to believe that this velocity is accurate and will speed up SGD. Here is a very important point (Kidambi et al., 2018; Liu and Belkin, 2018) that you should remember.

Momentum methods (Polyak's or Nesterov's) do not significantly accelerate SGD.

To be more precise, we saw that for Nesterov's updates in GD for strongly-convex functions we have a result of the form

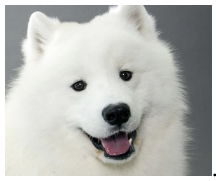
$$\|w^t - w^*\| \leq e^{-t/\sqrt{\kappa}}\, \|w^0 - w^*\|$$

while the constant without momentum is larger, it is $e^{-t/\kappa}$. This term is directly related to the second term in Theorem 11.4. The above authors come up with counterexamples to show that Nesterov's updates with SGD only improve this multiplicative term to something like $e^{-ct/\kappa}$ for some c; in other words using Nesterov's updates with SGD only lead to a constant factor improvements in the convergence rate.

Accelerating stochastic optimization algorithms is done via the use of control variates (Le Roux et al., 2012). Broadly speaking these methods work by using the previous gradients in SGD $\left\{\nabla\, \ell^{\omega_1}(w^1), \dots, \nabla\, \ell^{\omega_t}(w^t)\right\}$ to compute some surrogate for the current full gradient $\nabla\, \ell(w^t)$ and compute the descent direction using both this surrogate full gradient and the standard SGD gradient.

Why do we use Nesterov's method to train deep networks? It is worthwhile to think why we use Nesterov's momentum to train deep networks: (i) we know that momentum does not help speed up training, and (ii) momentum is simply a faster way to minimize the same objective ℓ so it does not have any regularization properties that help generalization either. We do not have a definitive answer to this question yet but here is what we know.

Datasets that we use in deep learning represent quite narrow distributions (natural images of animals, household objects etc.). For instance, the two images below are essentially the same in spite of belonging to different classes.


.

Most weights of a deep network will have a similar gradient for these images as input, the weights for which the gradient will differ are likely to be the weights at the top few layers of the network. This entails that even if the stochastic gradients are computed on different mini-batches, they are essentially quite similar to each other, and thereby to the full-gradient. More precisely, the covariance of mini-batch gradients

$$\text{Cov}\left(\nabla \ell_{\mathcal{B}}(w),\ \nabla \ell_{\mathcal{B}'}(w)\right) = \mathop{\mathbb{E}}_{\mathcal{B},\mathcal{B}'}\left[\left(\nabla \ell_{\mathcal{B}}(w) - \nabla \ell(w)\right)\left(\nabla \ell_{\mathcal{B}'}(w) - \nabla \ell(w)\right)^\top\right]$$

is a matrix with very few non-zero eigenvalues; only about 0.5% of the eigenvalues are non-zero (Chaudhari and Soatto, 2017) even for large networks. This means that the SGD gradient while training deep networks is essentially the full gradient and we should expect momentum to accelerate convergence in practice.

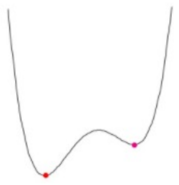
11.4 Understanding SGD as a Markov Chain

The preceding development tells us how SGD works and how many iterations of SGD we need to get within an ϵ-neighborhood of the global minimum for convex functions. Things are not this easy to understand for non-convex functions; essentially if we have two minima u^*, v^*

$$\nabla \ell(u^*) = \nabla \ell(v^*) = 0$$

depending upon where GD/SGD are initialized they can converge to different places. In this section, we will look at an alternative way of understanding how SGD works for non-convex functions. The development here will be much more abstract that the preceding section because we want to capture the overall properties of SGD.

⚠ A non-convex function with two local minima. The one on the left is the global minimum but gradient descent may not always reach here.



11.4.1 Gradient flow

Let us first talk about gradient descent. Just like we constructed a model for Nesterov's updates using a differential equation, we will first construct a model for gradient descent using a differential equation. The updates are given by

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

If we again imagine a continuously differentiable curve $W(\tau)$ as a model for these discrete-time updates and time

$$\mathrm{d}\tau := \eta$$

we can write a differential equation of the form

$$\frac{\mathrm{d}W}{\mathrm{d}\tau} = \dot{W}(\tau) = -\nabla \ell(W(\tau)); \quad W(0) = w^0. \tag{11.16}$$

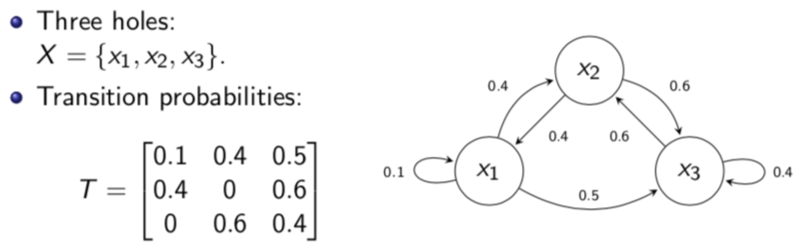
This is called gradient flow. If we wanted to execute gradient flow on a computer, we can do so using Euler discretization

$$\dot{W}(\tau) \approx \frac{W(\tau + \Delta\tau) - W(\tau)}{\Delta\tau} = -\nabla \ell(W(\tau)).$$

for any value of the time-step $\Delta\tau$. If the time-step $\Delta\tau = \eta$ we get exactly gradient descent. More precisely, gradient flow is the limit of gradient descent as the learning rate $\eta \to 0$. It is important to always remember that gradient flow is a model for GD, not GD itself. Our goal in the remainder of the section is to develop a similar model for SGD.

11.4.2 Markov chains

Consider the Whack-The-Mole game: a mole has burrowed a network of three holes w^1, w^2, w^3 into the ground. It keeps going in and out of the holes and we are interested in finding which hole it will show up next so that we can give it a nice whack.



- Three holes: $X = \{x_1, x_2, x_3\}$.
- Transition probabilities:

$$T = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.4 & 0 & 0.6 \\ 0 & 0.6 & 0.4 \end{bmatrix}$$

This is an example of a Markov chain. There is a transition matrix P which determines the probability P_{ij} of the mole resurfacing on a given hole w^j given that it resurfaced at hole w^i the last time. The matrix P^t is the t-step transition matrix

$$P^t_{ij} = \mathbb{P}(w^t = w^j \mid w^{(0)} = w^i).$$

If there exist times t, t' such the both the probabilities

$$\mathbb{P}(w^t = w^j \mid w^{(0)} = w^i) \quad \mathbb{P}(w^{(t')} = w^i \mid w^{(0)} = w^j)$$

are non-zero the two states w^i and w^j are said to "communicate"

$$w^i \leftrightarrow w^j$$

The set of states in the Markov chain that *all* communicate with each other are an equivalence class. This means that the Markov chain can visit any state from any other state in this equivalence class with non-zero probability, we just might have to wait for a really long time if $P^t_{ij} \approx 0$ for two states w^i, w^j. If all the states in the Markov chain belong to the same equivalence class, it is called irreducible. A related concept is that of "positive recurrence", i.e., if the Markov chain was at a state w at some time, it comes back to the same state after some finite time. Since the process is Markov it forgets that is just came back to the same state and therefore positive recurrence also means that if we consider an infinitely long trajectory of a Markov chain, the chain visits the same state infinitely many times along this trajectory. You can see the animations at https://setosa.io/ev/markov-chains to build more intuition.

Invariant distribution of a Markov chain The probability of being in a state w^i at time $t+1$ can be written as

$$\mathbb{P}(w^{t+1} = w^i) = \sum_{j=1}^N \mathbb{P}(w^{t+1} = w^i \mid w^t = w^j)\ \mathbb{P}(w^t = w^j).$$

This equation governs how the probabilities $\mathbb{P}(w^t = w^i)$ change with time t. Let's do the calculations for the Whack-The-Mole example. Say the mole was at hole w^1 at the beginning. So the probability distribution of its presence

$$\pi^{(t)} = \begin{bmatrix} \mathbb{P}(w^t = w^1) \\ \mathbb{P}(w^t = w^2) \\ \mathbb{P}(w^t = w^3) \end{bmatrix}$$

is such that

$$\pi^1 = [1, 0, 0]^\top.$$

We can now write the above formula as

$$\pi^{(t+1)} = P^\top \pi^{(t)}$$

and compute the distribution $\pi^{(t)}$ for all times

$$\begin{aligned} \pi^2 &= P^\top \pi^1 = [0.1, 0.4, 0.5]^\top; \\ \pi^3 &= P^\top \pi^2 = [0.17, 0.34, 0.49]^\top; \\ \pi^4 &= P^\top \pi^3 = [0.153, 0.362, 0.485]^\top; \\ &\vdots \\ \pi^\infty &= \lim_{t \to \infty} P^t\, \pi^1 \\ &= [0.158, 0.355, 0.487]^\top. \end{aligned}$$

If such a distribution π^∞ exists, the Markov chain is said to have "equilibriated" or reached an invariant distribution. The numbers $\mathbb{P}(w^{t+1} = w^i)$ stop changing with time. We can compute this invariant distribution by writing

$$\pi^\infty = P^\top \pi^\infty.$$

Does such a limiting invariant distribution π^∞ always exist? It turns out that if a Markov chain has a finite number of states then the invariant distribution π^∞ always exists; this is easy to show yourself. If the Markov chain is irreducible and aperiodic, then the invariant distribution is also unique. We can also compute the π^∞ given a transition matrix P: the invariant distribution is the (right-)eigenvector of the matrix $P^\top$ corresponding to the eigenvalue 1.

Periodicity of a Markov chain A state of a Markov chain is periodic with period k if the probability of coming back to the same state is zero for times that *are not* integral multiples of k and the probability of coming back to the same state is non-zero for all times that *are* integral multiplies of k. To take a simple example, every number of a clock is a periodic state; the Markov chain comes back to that state at regular intervals. If we cannot find such a time k for a given state, then the state is aperiodic. It is easy to see that if there exists an aperiodic state in one communicating class, then all the other states in that class also have to be aperiodic. It is useful to remember that if a particular state has a non-zero probability of self-transition, then the state is aperiodic.

Example 11.5. Consider a Markov chain on two states where the transition matrix is given by

$$P = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix}.$$

This is an irreducible Markov chain because you can hop between any two states with non-zero probability within one step. It is also recurrent: this is intuitive because say the Markov chain was in state 1, it is easy for it to come back to this state after a few hops. After the chain comes back to state 1, the Markov property means the chain forgets all the past steps and will again come back to state 1. The expected number of times the Markov chain comes back to state 1 is infinite. Each of the two states has a non-zero probability of self-transition, so both of them are aperiodic.

We are therefore guaranteed that a unique invariant distribution exists for this Markov chain. In this case it is

$$\begin{aligned} \pi^1 &= 0.5\pi^1 + 0.4\pi^2 \\ \pi^2 &= 0.5\pi^1 + 0.6\pi^2. \end{aligned}$$

Note that the constraint for π being a probability distribution, i.e., $\pi^1 + \pi^2 = 1$ is automatically satisfied by the two equations. We can solve for π^1, π^2 to get

$$\pi^1 = 4/9 \quad \pi^2 = 5/9.$$

Time spent at a particular state by the Markov chain We can observe a long trajectory of a Markov chain and compute the number of times the chain is in a particular state w^i. This is directly proportional to $\pi^\infty(w^i)$. In other words, if the invariant distribution gives small probability to a particular state, if we stop the Markov chain at an arbitrary time during its trajectory, we are very unlikely to find the Markov chain at this state.

11.4.3 A Markov chain model of SGD

The updates of SGD with mini-batch size $ƀ$ are given by

$$w^{t+1} - w^t = -\eta \nabla \ell_{ƀ}(w^t).$$

Notice that conditional on the iterate w^t, the next iterate w^{t+1} is independent of w^{t-1}, all these three quantities are random variables because they depend on the input data $\omega_0, \ldots, \omega_t$ sampled by SGD in the previous time-steps. You should never make the mistake of saying that gradient descent is a Markov chain; there is no randomness in the iterates of GD.

Transition probability of SGD What is the transition probability

$$\mathbb{P}(w^{t+1} \mid w^t)$$

for SGD? If we take the conditional expectation on both sides

$$\mathop{\mathbb{E}}_{\mathscr{b}}\left[w^{t+1} - w^t \mid w^t\right] = -\eta \mathop{\mathbb{E}}_{\mathscr{b}}\left[\nabla \ell(w^t)\right] = -\eta \nabla \ell(w^t);$$

in other words, on-average the change in weights at w^t is proportional to the full gradient $\nabla \ell(w^t)$. Notice that the change in weights exactly the same for GD; this should not be surprising after all, if the gradient of SGD is unbiased then SGD is GD "on-average".

Variance of SGD weight updates The variance is computed as follows

$$\begin{aligned}\mathop{\text{Var}}_{\mathscr{b}}\left(w^{t+1} - w^t \mid w^t\right) &= \eta^2 \mathop{\text{Var}}_{\mathscr{b}}\left(\nabla \ell_{\mathscr{b}}(w^t) \mid w^t\right) \\ &= \eta^2 \mathop{\mathbb{E}}_{\mathscr{b}}\left[\left(\nabla \ell_{\mathscr{b}}(w^t) - \nabla \ell(w^t)\right)\left(\nabla \ell_{\mathscr{b}}(w^t) - \nabla \ell(w^t)\right)^\top\right]\end{aligned}$$

Notice that the variance of the weight updates in SGD is proportional to the square of the learning rate. We have seen this before, larger the learning rate more noisy the weight update as compared to the update using the full-gradient $\eta \nabla \ell(w^t)$. The variance is a large matrix $\in \mathbb{R}^{p \times p}$; this matrix depends on the current weight w^t.

If we are sampling the data inside a mini-batch with replacement the stochastic gradients are independent for different samples ω^1 and ω^2 in the mini-batch

$$\nabla \ell^{\omega^1}(w) \perp\!\!\!\perp \nabla \ell^{\omega^2}(w).$$

In other words

$$\mathop{\mathbb{E}}_{\omega^1,\omega^2}\left[\left(\nabla \ell^{\omega_1}(w^t) - \nabla \ell(w^t)\right)\left(\nabla \ell^{\omega_2}(w^t) - \nabla \ell(w^t)\right)^\top\right] = 0.$$

You can use this to show that

$$\begin{aligned}\mathop{\text{Var}}_{\mathscr{b}}\left(w^{t+1} - w^t \mid w^t\right) &= \eta^2 \mathop{\text{Var}}_{\omega^1,\ldots,\omega^{\mathscr{b}}}\left(\frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} \nabla \ell^{\omega^i}(w^t)\right) \\ &= \frac{\eta^2}{\mathscr{b}^2} \sum_{i=1}^{\mathscr{b}} \mathop{\text{Var}}_{\omega^i}\left(\nabla \ell^{\omega^i}(w^t)\right) \qquad (11.17) \\ &= \frac{\eta^2}{\mathscr{b}} \mathop{\text{Var}}_{\omega}\left(\nabla \ell^{\omega}(w^t)\right).\end{aligned}$$

The last step follows because we are sampling inputs ω^i uniformly randomly and therefore gradients $\nabla \ell^{\omega^i}(w^t)$ are not just independent but also identically distributed. In other words, a mini-batch size of $\mathscr{b}$ reduces the variance by a factor of $\mathscr{b}$.

SGD is like GD with Gaussian noise We now *model* the transition probability $\mathbb{P}(w^{t+1} \mid w^t)$ as a Gaussian distribution. Let us denote by W^t, W^{t+1} etc. the updates of this model. We now have

$$W^{t+1} = W^t + \xi^t$$

where ξ^t is Gaussian noise

$$\xi^t \sim N\left(-\eta \nabla \ell(w^t),\ \frac{\eta^2}{b} \underset{\omega}{\mathrm{Var}} \left(\nabla \ell^\omega(w^t)\right)\right).$$

In other words, on-average SGD updates weights like gradient descent, by a term $-\eta \nabla \ell(w^t)$ but SGD's updates also have a variance.

Such equations are called stochastic difference equations and they are quite difficult to understand compared to non-stochastic difference equations (what we see in gradient descent). So we will make a drastic simplification in our model. We will say that the variance of the mini-batch gradients is identity. Our model for SGD is

$$W^{t+1} = W^t - \eta \nabla \ell(W^t) + \frac{\eta}{\sqrt{b}} \xi^t \tag{11.18}$$

where we have zero-mean unit-variance Gaussian noise $\xi^t \sim N(0, I_{p\times p})$.

Remark 11.6. The above model for SGD is a Markov chain except that the states in the Markov chain is infinite; the number of states in the Whack-The-Mole example were finite. It is easy to see that the above model is not exactly SGD: (i) we assumed the the transition probability was a Gaussian which need not be the case while training a deep network, (ii) we further assumed that the Gaussian noise does not depend on w^t and has identity covariance. You can implement the above model on a computer, first you compute the *full gradient* $\nabla \ell(w^t)$ and then sample Gaussian noise ξ^t to update the weights to W^{t+1}. This is obviously not equivalent to SGD which updates weights using the stochastic gradient $\nabla \ell_b(w^t)$.

11.4.4 The Gibbs distribution

In a Markov chain we were interested in the invariant distribution because that gives us a way to understand where the chain spends most of its time. We can compute the invariant distribution for our model of SGD. It is a very powerful result (which we will not do) and leads to the so-called Gibbs distribution. The probability density of the invariant distribution is given by

$$\rho^\infty(w) = \frac{1}{Z(\beta)} e^{-\beta \ell(w)}. \tag{11.19}$$

The quantity

$$\beta = \frac{2b}{\eta} \tag{11.20}$$

and $Z(\beta)$ is a normalizing constant for probability density

$$Z(\beta) = \int_{\mathbb{R}^p} e^{-\beta \ell(w)} \, \mathrm{d}w.$$

Let us list a few properties of the Gibbs distribution that are apparent simply by looking at the above expression.

1. The invariant distribution is reached asymptotically and is the limiting distribution of the weights. For instance the sum of the weights along an infinitely long trajectory converges to the mean of the Gibbs distribution

$$\lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^T W^t = \int_w w\, \rho^\infty(w) \, \mathrm{d}w. \tag{11.21}$$

Similarly, the second moment of the weights along a long trajectory of SGD converges to the second moment of the Gibbs distribution; and same for the variance.

$$\lim_{T\to\infty} \frac{1}{T} \sum_{t'=1}^{T} \sum_{t=1}^{T} \left(W^{t'}\right) \left(W^{t}\right)^\top = \int_{w,w'} w\, {w'}^\top\, \rho^\infty(w)\, \rho^\infty(w')\, \mathrm{d}w \mathrm{d}w'. \tag{11.22}$$

2. The probability that the iterates of SGD are found at a location w is proportional to $e^{-\beta \ell(w)}$. If the training loss $\ell(w)$ is high, this probability is low and if the training loss is low, the probability is high. The Gibbs distribution therefore shows that if we let SGD run until it equilibriates we have a high chance of finding the iterates that have a small training loss. This observation is powerful because it does not require us to assume that $\ell(w)$ is convex. However this statement does require the assumption that the steps-size η of SGD does not go to zero; after all SGD iterates *stop* if $\eta = 0$.

3. The quantity $1/\beta$ is quite common in physics where it is called the "temperature". This temperature $\beta^{-1} = \frac{\eta}{2\text{ƀ}}$ fundamentally governs how the Gibbs distribution looks. Higher the temperature, more the noise in the iterates and vice-versa. If the learning rate η is large or the batch-size ƀ is small, it is easy for *our model of SGD* to jump over hills. This is the reason why the Gibbs distribution will be spread around the entire domain at high temperature. On the other hand, if temperature is very small, the Gibbs distribution puts a large probability mass in places where the training loss is small and the probability of finding iterates at other places in the domain diminishes. In particular, if $\beta \to \infty$, the Gibbs distribution only puts non-zero probability on the global minima of the loss function $\ell(w)$.

4. Written in another way, if we want the Gibbs distribution to remain the same we should ensure that
$$\beta^{-1} = \frac{\eta}{2\text{ƀ}} \text{ is a constant.}$$
If you increased the batch-size by two times, you should also double the learning rate if you desire that the solutions of SGD are qualitatively similar.

5. We have achieved something remarkable by looking at the Gibbs distribution. We have discovered an algorithm to find the global minimum of a non-convex loss function.
    - Start from some initial condition w^0;
    - Take lots of steps of SGD with learning rate η until SGD reaches its invariant distribution, i.e., until it equilibriates;
    - Reduce the step-size η and repeat the previous step

   This is only a formal algorithm but in theory it will converge to the global minimum of a non-convex function $\ell(w)$ if the number of steps is very large. The catch of course is that at each step we have to wait until SGD equilibriates. For many problems, it may take an inordinately long amount of time for SGD to equilibriate.

It is very important to remember that when we train a deep network we are executing one run of SGD. The invariant distribution is an abstract concept that does not really exist on your computer. We have constructed this model to help us understand how updates of SGD behave.

11.4.5 Convergence of a Markov chain to its invariant distribution

For gradient descent and SGD, we had quantities like $\|w^t - w^*\|$ or $\ell(w^t) - \ell(w^*)$ that let us measure the progress towards the global minimum. For a non-convex problem, there may not exist a unique global minimum, or there may be multiple local minima in the domain where the gradient vanishes. We discussed in the preceding section how the invariant distribution of SGD is achieved even if the loss $\ell(w)$ is non-convex. In this section, we will see a simple tool to measure progress towards this distribution.

Let us define a quantity called the Kullback-Leibler (KL) divergence between two probability distributions. For two probability distributions $p(w)$ and $q(w)$ supported on a discrete set $w \in W$, the KL-divergence is given by

$$\text{KL}(p \mid\mid q) = \sum_{w \in W} p(w) \ \log \frac{p(w)}{q(w)}. \tag{11.23}$$

This formula is well-defined only if for all w where $q(w) = 0$, we also have $p(w) = 0$. The KL-divergence is a measure of the distance between two distances, it is zero if and only if $p(w) = q(w)$ for all $w \in W$. It is always positive (you can show this easily using Jensen's inequality). However, the KL-divergence is not a metric because it is not symmetric

$$\text{KL}(p \mid\mid q) \neq \text{KL}(q \mid\mid p) = \sum_{w \in W} q(w) \ \log \frac{q(w)}{p(w)}.$$

For probability densities, the KL-divergence

$$\text{KL}(p \mid\mid q) = \int_w p(w) \ \log \frac{p(w)}{q(w)} \ \text{d}w \tag{11.24}$$

is defined analogously and has the same properties.

We will now show a very powerful result: the KL-divergence of the state distribution of a Markov chain decreases monotonically as the Markov chain converges to its invariant distribution. Although, this result is true for SGD as well, we will only prove it for a Markov chain with finite states. Let the initial distribution of the Markov chain be π^0, its transition matrix be P and its invariant distribution be π^∞. We will assume that the Markov chain is such that the invariant distribution exists (it is irreducible and recurrent).

Let us also assume that a reverse transition matrix

$$P_{ij}^{\text{rev}} = \mathbb{P}(w^t = w^i | w^{t+1} = w^j).$$

exists; such Markov chains are called reversible. For any states w, w' this transition matrix satisfies the definition of conditional probability

$$\mathbb{P}(w^{t+1} = w' | w^t = w)\, \mathbb{P}(w^t = w) \quad = \quad \mathbb{P}(w^t = w | w^{t+1} = w')\, \mathbb{P}(w^{t+1} = w').$$

In our notation, this becomes

$$P^{\text{rev}}_{ww'} = \frac{P_{w'w}\pi(w')}{\pi(w)} = \frac{P_{w'w}\pi(w')}{\sum_{w'} P_{w'w}\pi(w')}.$$

Lemma 11.7. For a reversible Markov chain with an invariant distribution π^∞, KL$(\pi^\infty \mid\mid \pi^t)$ decreases monotonically:

$$\text{KL}(\pi^\infty \mid\mid \pi^{t+1}) \leq \text{KL}(\pi^\infty \mid\mid \pi^t). \tag{11.25}$$

Proof. The proof is a simple calculation.

$$\begin{aligned}
\text{KL}(\pi^\infty \mid\mid \pi^{t+1}) &= \sum_w \pi^\infty(w)\ \log \frac{\pi^\infty(w)}{\pi^{t+1}(w)} \\
&= \sum_w \pi^\infty(w)\ \log \frac{\pi^\infty(w)}{\sum_{w'} P_{w'w}\ \pi^t(w')} \\
&= -\sum_w \pi^\infty(w)\ \log \frac{\sum_{w'} P_{w'w}\ \pi^t(w')}{\pi^\infty(x)} \\
&= -\sum_w \pi^\infty(w)\ \log \left(\sum_{w'} P^{\text{rev}}_{ww'}\ \frac{\pi^t(w')}{\pi^\infty(w')} \right) && \text{(substitute definition of } P^{\text{rev}} \text{ for distribution } \pi^\infty) \\
&\leq -\sum_w \pi^\infty(w) \sum_{w'} P^{\text{rev}}_{ww'}\ \log \frac{\pi^t(w')}{\pi^\infty(w')} && \text{(Jensen's inequality)} \\
&= \sum_{w'} \sum_x P^{\text{rev}}_{ww'}\ \pi^\infty(w)\ \log \frac{\pi^\infty(w')}{\pi^t(w')} && \text{(flip the negative sign, exchange sum)} \\
&= \sum_{w'} \pi^\infty(w')\ \log \frac{\pi^\infty(w')}{\pi^t(w')} \\
&= \text{KL}(\pi^\infty \mid\mid \pi^t).
\end{aligned}$$

The distance to the invariant distribution π^∞ decreases at each step of the Markov chain. A similar statement is true for the reverse KL divergence:

$$\text{KL}(\pi^{t+1} \mid\mid \pi^\infty) \leq \text{KL}(\pi^t \mid\mid \pi^\infty).$$

□

The above result is also true for SGD which, as we discussed, can be modeled as a Markov chain with infinite states. It gives us some very important intuition. Just like gradient descent makes monotonic progress towards the global minimum w^*, a Markov chain (or SGD) makes monotonic progress towards its invariant distribution. The big difference between them is that while we required that the loss function $\ell(w)$ is convex for gradient descent to guarantee this monotonic progress, the loss need not be convex for the case of the Markov chain model of SGD.

This result *does not* mean that SGD makes monotonic progress towards the global minimum $w^* = \text{argmin}_w \ell(w)$. We choose to look at SGD not as one particle undergoing (stochastic) gradient descent updates but rather as a Markov chain. The probability distribution of states of this Markov chain is then a legitimate object (the distribution π^t is the distribution of weights W^t obtained after many independent run of SGD from different initializations). Although π^t is *not* meaningful across *one* run of SGD, we can use it to get an abstract understanding of how SGD also makes monotonic progress as it converges if we imagine many *independent* runs of SGD occurring simultaneously.

Chapter 12

Shape of the energy landscape of neural networks

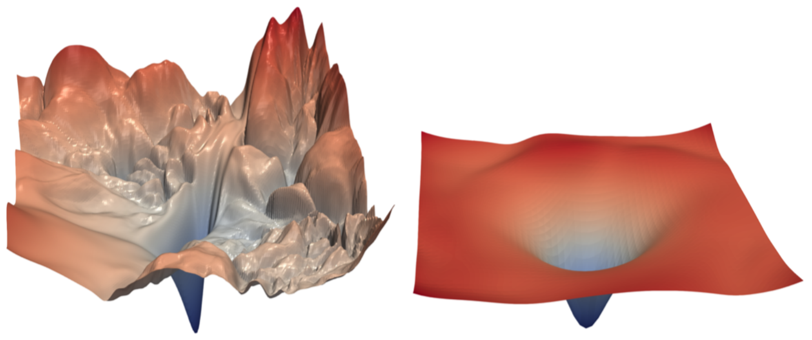
Reading

1. Goodfellow Chapter 13
2. “Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima” by Baldi and Hornik (1989)
3. “Entropy-SGD: Biasing gradient descent into wide valleys” by Chaudhari et al. (2016)

In this chapter, we will try to understand the shape of the objective for training neural networks. We would like to characterize the difficulty of training neural networks. We know that the objective is not convex and training a network is difficult because of it. But how non-convex is the objective? The questions we want to answer here are of the following form.

1. How many global minima exist?
2. How many local minima and saddle points exist?
3. What is the loss at the local minima or saddle points? If we train with gradient descent or stochastic gradient descent, what loss can we expect to obtain even if we don’t reach the global minimum?
4. What is the local geometry of the loss function?
5. What is the global topology of the loss function?

This will help understand how SGD seems to train deep networks so efficiently and why we often get very good generalization error after training. As a pre-cursor to how the picture of the energy landscape of a neural network looks like, here’s one picture from Li et al. (2018):



(a) without skip connections (b) with skip connections

Figure 12.1: A picture of the training loss. The picture on the left was created by sampling two directions randomly out of the millions of weights for a residual network without skip-connections and computing the training loss by discretization of this two-dimensional space. The picture on the right is a similar picture for the resnet with skip-connections intact. In this picture, we see that while the training loss is very complex on the left-hand side with lots of local minima and saddle points, the loss is much more benign on the right-hand side.

12.1 Introduction

Let us introduce a few quantities that will help characterize the energy landscape. We will consider the case when the function $\ell(w)$ is twice-differentiable.

Global minima are all points in the set

$$\{w : \ell(w) \leq \ell(w') \text{ for all } w'\}.$$

Draw the Gibbs distribution of SGD if $\ell(w)$ has multiple global minima.

Note that there may exist many different locations all with the same loss $\ell(w)$, they would all be global minima in this case. Local minima are all points in the set

$$\left\{w : \nabla \ell(w) = 0, \nabla^2 \ell(w) \succeq 0\right\}.$$

Draw the Gibbs distribution of SGD if $\ell(w)$ has multiple global minima and multiple local minima.

i.e., all points w where the Hessian $\nabla^2 \ell(w)$ is positive semi-definite. Note that the two conditions (i) first-order stationarity $\nabla \ell(w) = 0$ and (ii) positive semi-definiteness of the Hessian $\nabla^2 \ell(w) \succeq 0$ also have to be satisfied for all global minima. Critical points are all locations which satisfy only first order stationarity

$$\{w : \nabla \ell(w) = 0\}.$$

Saddle points are critical points but which are neither local minima not local maxima

$$\left\{w : \nabla \ell(w) = 0, \nabla^2 \ell(w) \text{ is neither positive nor negative semi-definite}\right\}.$$

Non-convex functions, in general, can have all these different kinds of locations in the energy landscape and this makes minimizing the objective difficult. Our goal in this chapter is to learn theoretical and empirical results that help paint a mental picture of what the energy landscape looks like.

12.2 Deep Linear Networks

Let us consider the simplest case of linear neural networks first. We will have a two-layer neural network which takes in inputs x^i and aims to predict targets y^i. For simplicity, we will consider the case when both

$$x^i, y^i \in \mathbb{R}^d.$$

and use the regression loss

$$\ell(A, B) = \frac{1}{2n} \sum_{i=1}^{n} \|y^i - AB\, x^i\|_2^2 \tag{12.1}$$

We use the standard trick of appending a 1 to the input x^i so that we don't have to carry around biases in our equations.

The matrices A, B are the weights of the neural network with

$$A \in \mathbb{R}^{d \times p}, B \in \mathbb{R}^{p \times d}.$$

We will consider the case when $p \leq d$. We are interested in finding A and B and will develop some results from Baldi & Hornik's paper.

Least squares solution A simple calculation reveals that for a single-layer network the solution of the problem

$$L^* = \underset{L}{\text{argmin}}\, \frac{1}{2n} \sum_{i=1}^{n} \|y^i - Lx^i\|_2^2$$

is

$$L^* = \Sigma_{yx}\, \Sigma_{xx}^{-1} \tag{12.2}$$

where

$$\Sigma_{yx} = \sum_i y^i {x^i}^\top$$
$$\Sigma_{xx} = \sum_i x^i {x^i}^\top.$$

The matrices Σ_{yx} and Σ_{xx} are the data covaraiance matrices.

Projection of a vector onto a matrix It will be useful to define a projection matrix. Say we have a vector v that we want to project on the span of the columns of a full-rank matrix

$$M = \begin{bmatrix} m_1 & m_2 & \dots & m_n \end{bmatrix}.$$

If this projection is $\hat{v} \in \text{span}\,\{m_1, \dots, m_n\}$, we know that it has to satisfy

$$(v - \hat{v}) \perp m_k \text{ for all } k \leq n \quad \Rightarrow \quad m_k^\top (v - \hat{v}) = 0.$$

The vector $\hat{v}$ is also obtained by a combination of the columns of M, so there exists a vector c which allows us to write

$$\hat{v} = Mc.$$

These together imply
$$c = (M^\top M)^{-1} M^\top \hat{v}$$
and finally
$$\hat{v} = \underbrace{M(M^\top M)^{-1} M^\top}_{\text{projection matrix}} v$$
$$=: P_M\, v.$$
where the matrix P_M is called the projection matrix corresponding to the matrix M.

⚠ Note that $P_M^2 = P_M$, i.e., if we project the vector twice onto the column space of M, the second projection does nothing. Also, any projection matrix P is symmetric. To see this, consider two vectors v, w and the dot products
$$\langle Pv, w\rangle\,, \text{ and } \langle v, Pw\rangle\,.$$
In both cases, one of the vectors lies completely in the column space of M and therefore the dot product ignores any component that is orthogonal to the column space of M. This means
$$\langle Pv, w\rangle = \langle v, Pw\rangle = \langle Pv, Pw\rangle\,.$$
We can now rewrite the first equality to obtain
$$(Pv)^\top w = v^\top (Pw)$$
$$\Rightarrow v^\top P^\top w = v^\top P w$$
and since this is true for any two vectors v, w, we have that $P = P^\top$.

Back to deep linear networks We know from the homework problem that there is no unique solution to the problem
$$A^*, B^* = \operatorname*{argmin}_{A,B} \frac{1}{2n} \sum_{i=1}^{n} \|y^i - AB\, x^i\|_2^2.$$
If A^*, B^* are solutions, so are $A^* P, P^{-1} B^*$ for any invertible matrix P. We also showed in the homework that the objective is not convex. But if we fix either A or B and only optimize over the other, the loss is convex. Notice that the rank of AB is at most p.

Fact 12.1 (Critical points of B if A is fixed). For any A, the function $\ell(A, B)$ is convex in B and has a minimum at
$$\left(A^\top A\right) \widehat{B}(A) \Sigma_{xx} = A^\top \Sigma_{yx}.$$
If Σ_{xx} is invertible and A is full-rank, then we can write
$$\widehat{B}(A) = (A^\top A)^{-1} A^\top \Sigma_{yx} \Sigma_{xx}^{-1}. \tag{12.3}$$
Note that these are all locations when the gradient
$$\frac{\partial \ell}{\partial B} = 0.$$
Fact 12.2 (Critical points of A if B is fixed). We have an analogous version of the previous fact for A: if B is fixed, the loss is convex in A, for full-rank Σ_{xx} and B, then for $\frac{\partial \ell}{\partial A} = 0$, we should have
$$AB\Sigma_{xx}B^\top = \Sigma_{yx}\, B^\top. \tag{12.4}$$
or
$$\widehat{A}(B) = \Sigma_{yx}\, B^\top \left(B\Sigma_{xx}B^\top\right)^{-1}. \tag{12.5}$$
Fact 12.3 (Critical points of (A, B)). We now solve the equations (12.3) and (12.5) to get a critical point, i.e., the gradient of the objective in both A and B is zero. First
$$W = AB = P_A\, \Sigma_{yx} \Sigma_{xx}^{-1}. \tag{12.6}$$
from (12.3). Next, multiply on both sides of (12.4) by $A^\top$ and substitute the above value of W to get that the matrix A should satisfy
$$P_A \Sigma = \Sigma P_A = P_A \Sigma P_A.$$
where
$$\Sigma = \Sigma_{yx} \Sigma_{xx}^{-1} \Sigma_{xy}.$$

Fact 12.4 (If W is a critical point, then it can be written as a projection of the least squares solution $\Sigma_{yx}\Sigma_{xx}^{-1}$ on the subspace spanned by some p eigenvectors of Σ). This is an important fact. Let us say we have a full-rank Σ with distinct eigenvalues $\lambda_1 > \ldots > \lambda_d$. Let u_{i_k} be the eigenvector associated with the i_k^{th} eigenvalue of Σ. So given any set of p eigenvalues

$$\mathcal{I} = \{i_1, \ldots, i_p\} \text{ with } \quad 1 \leq i_k \leq d \text{ for all } k.$$

we can define a matrix of rank p

$$U_{\mathcal{I}} = \begin{bmatrix} u_{i_1} & u_{i_2} & \ldots & u_{i_p} \end{bmatrix}.$$

Then one can show that the matrices A and B are critical points if and only if there is a set $\mathcal{I}$ and an invertible matrix $C \in \mathbb{R}^{p \times p}$ such that

$$\begin{aligned} A &= U_{\mathcal{I}}\, C \\ B &= C^{-1} U_{\mathcal{I}}^{\top}\, \Sigma_{yx}\Sigma_{xx}^{-1}. \end{aligned} \tag{12.7}$$

You can find the proof in the Appendix of Baldi & Hornik's paper. Because $U_{\mathcal{I}}$ is a matrix of orthonormal vectors we also have

$$P_{U_{\mathcal{I}}} = U_{\mathcal{I}}\, U_{\mathcal{I}}^{\top}$$

and therefore

$$W = P_{U_{\mathcal{I}}}\, \Sigma_{yx}\Sigma_{xx}^{-1}$$

which is the same form for W as (12.6) in Fact 3 and L^* in (12.2). In other words, the solution $W = AB$ in a two-layer linear network is given by our original least squares regression matrix followed by an orthogonal projection onto the subspace spanned by p eigenvectors of Σ.

Fact 12.5 (If W is the global minimum for a two-layer network, then it is a projection of the solution for a single-layer network onto the subspace spanned by the top p eigenvectors of Σ). You can further show that the objective

$$\ell(A, B) = \text{trace}(\Sigma_{yy}) - \sum_{i_k \in \mathcal{I}} \lambda_{i_k}. \tag{12.8}$$

at a critical point (A, B). The first term is a constant with respect to the parameters of the network A, B. The second term is a sum of the eigenvalues of the matrix Σ at indices that we picked in our set $U_{\mathcal{I}}$. What is the index set that minimizes this loss? It is simply the largest p eigenvalues of Σ. This is also a unique value for the loss because we have assumed that all the eigenvalues are distinct. This also solidifies the connection of this model with Principal Component Analysis (PCA), the matrix W is projecting on the sub-space spanned by the top p eigenvectors in the auto-associative case.

❓ Based on the previous two facts, what can you say about the solution W if $p \geq d$ and Σ is invertible? Since the two-layer network simply projects on the p eigenvalues of Σ, if $p \geq d$ and Σ is invertible, the solution already lies in the column-space of Σ and therefore $W = L^*$.

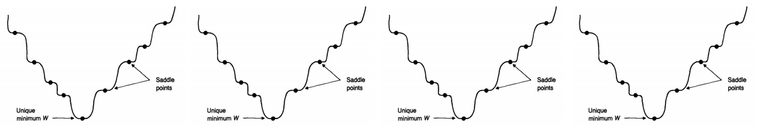
Fact 12.6 (There are exponentially many saddle points for a two-layer network). There are a total of $\binom{d}{p}$ possible index sets $\mathcal{I}$. One of them as we saw above corresponds to a global minimum. It can be shown that all the others are saddle points. Note that there are exponentially many saddle points. This is an important fact to remember: there are exponentially many saddle points in a hierarchical architecture. Smaller the number of neurons in the hidden layer p (also the upper bound for the rank of the weight matrices), fewer are the number of saddle points but this also creates a dimensionality bottleneck in the feature space. If p is too small as compared to d we lose large amounts of information necessary to classify the image and the network may not work well.

Fact 12.7 (No local minima in a deep linear network, all minima are global minima). The proof of the previous fact (see the Appendix of Baldi and Hornik (1989)) shows that any index set $\mathcal{I} \neq \{1, \ldots, p\}$ cannot be a local minimum. There are no local minima for a deep linear network, only global minima and saddle points. This is often said as "linear networks have no bad local minima".

Fact 12.8 (The global minimum is not unique). This is perhaps the most important point of this chapter. The loss at the global minimum is unique, not the global minimum itself. Any full-rank square matrix $C \in \mathbb{R}^{p \times p}$ of our choice gives a pair of solutions (A, B). How many such solutions are there? There are lots and lots of such solutions, in fact, given any solution with a particular C if we can perturb the C without losing rank (quite easy to do by, say, changing the eigenvalues slightly) we get another solution of a linear network.

Fact 12.9 (All the previous results are true for multi-layer linear networks). The same results are true for deep linear networks (Kawaguchi, 2016). These results also hold if $\dim(y_i) = 1$, i.e., for the regression case.

We used a simple two-layer linear network to obtain an essentially complete understanding of how the loss function looks like. A schematic looks as follows.



There are lots of locations where the global minimum of the function is achieved. There are lots of saddle points in the energy landscape. The Gibbs distribution for this energy landscape has a lot of modes, one each at the global minima.

How does weight-decay

$$\Omega(A, B) = \lambda \left(\|A\|_F^2 + \|B\|_F^2 \right)$$

change the energy landscape of deep linear networks? It changes the number of global minima, only the ones that have the smallest ℓ_2 norm remain in the energy landscape. It also reduces the number of saddle points because the Hessian at saddle points has an extra additive term that involves λ.

12.3 Extending the picture to deep networks

Let us think carefully about the non-uniqueness of the solution for a two-layer network. We know that all critical points are of the form

$$\begin{aligned} A &= U_{\mathcal{I}} C, \\ B &= C^{-1} U_{\mathcal{I}}^\top \Sigma_{yx} \Sigma_{xx}^{-1}. \end{aligned}$$

The gradient at these critical points is zero. Given a particular C, we can perturb it slightly and obtain a new critical point (a new saddle point, or a new global minimum) and this keeps the objective unchanged. Effectively, we have a connected set of global minima and saddle points for a deep linear networks.

If one were to try to visualize this energy landscape and extend the picture heuristically to deep networks with nonlinearities, we can think of the global minimum as looking like the basin of the Colorado river.

The important point to remember from this picture is that all the points at the basin of the river are solutions that obtain a good training loss. Although gradient-based algorithms (GD/SGD etc.) do not allow us to travel along the river (the gradient is zero along it), if the river basin snakes around in the entire domain, no matter where the network is initialized, we always have a global minimum close to the initialization. Essentially, the objective of deep networks is not convex, but current results indicate that it is quite benign. And this is perhaps the reason why it is so easy to train them.

Chapter 13

Generalization performance of machine learning models

This chapter gives a preview of generalization performance of deep networks. We will take a more abstract view of learning algorithms here and focus only on binary classification. We will first introduce a "learning model", 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.

13.1 The PAC-Learning model

Our goal in machine learning is to use the training data in order to construct a model that generalizes well, i.e., has good performance outside of the training data. Formally, we search over a hypothesis space $\mathcal{F}$, e.g., a specific neural net architecture, using the available data to find a good hypothesis $f \in \mathcal{F}$. As we motivated in Chapter 2, without further assumptions, we cannot guarantee that this hypothesis works well on test data. We therefore assume two things in this chapter:

1. Nature provides independent and identically distributed samples $x \in \mathcal{X}$ from some (unknown to the learner) distribution P.

2. Nature labels these samples with $c(x)$ which is again unknown to the learner.

Both training and test data are samples from Nature's distribution P. We will also assume that even if the true labeler $c(x)$ is unknown to us, we know that it belongs to a chosen hypothesis class $c(x) \in \mathcal{C}$ and is deterministic, i.e., Bayes error is zero. Changing this assumption does not change the crux of this theory.

Consider a learning algorithm, denoted by L. Given a dataset $D = \left\{(x^i, c(x^i)\right\}_{i=1}^n$ and a hypothesis class $\mathcal{C}$, the population risk (for classification) of the hypothesis output by this learning algorithm is

$$R(h) = \mathop{\mathbb{E}}_{x \sim P} \left[\mathbf{1}_{\{f(x) \neq c(x)\}} \right]$$

Let us assume that the learning algorithm is deterministic for now, i.e., given a training dataset D it returns a unique answer f. Let us assume that the hypothesis class that the learner searches over, named $\mathcal{F}$ is the same as the hypothesis class $\mathcal{C}$. What do we want from this algorithm?

We expect that it works well for all hypotheses Nature could use to label data $c \in \mathcal{C}$ and all datasets D drawn from P. The PAC-Learning model postulates the following desiderata upon the learning algorithm.

1. We are okay with an answer f with error

   $$R(f) \in [0, 1/2)$$

   because we only have access to finitely many training data. This is the "approximate correct" part of the PAC-Learning. However the error should decrease as n increases.

2. The dataset D is a random variable. This implies that the hypothesis outputted by the learning algorithm $f(D)$ is also a random variable. The above statement therefore should hold with some large probability over draws of the dataset D. In other words, there can be a small probability that a non-representative dataset D is drawn and we do not expect the learner to output a good hypothesis with $R(f) < 1/2$. However the probability of such failure, let us call it $\delta \in [0, 1/2)$, should also become smaller if more data is provided. This is the "probably" part of PAC-Learning.

We now have a definition of what it means to be a good learning algorithm.

Definition 13.1 (PAC-learnable hypothesis class). A hypothesis class $\mathcal{C}$ is PAC-learnable if there exists an algorithm L such that for every $c \in \mathcal{C}$, for every $\epsilon, \delta \in [0, 1/2)$, if L is given access to $n(\epsilon, \delta)$ i.i.d. training data from P and their corresponding labels c then it outputs a hypothesis $h_D \in \mathcal{C}$ such that

$$\mathbb{P}_D \left(R(f) < \epsilon \right) \geq 1 - \delta.$$

We want the learner to be statistically efficient, i.e., as our desiderata ϵ, δ get smaller, we should expect $n(\epsilon, \delta)$ to not grow too quickly. For instance, we would like $n(\epsilon, \delta)$ to be a polynomial function of $1/\epsilon$ and $1/\delta$. The minimum number of samples $n(\epsilon, \delta)$ required to learn a hypothesis class $\mathcal{C}$ is called the sample complexity of $\mathcal{C}$. One is also typically interested in the computational complexity of finding f, e.g., to avoid a brute-force algorithm L that searches over the entire hypothesis class $\mathcal{F} = \mathcal{C}$; we will not worry about it here.

It is important to notice that PAC-learning assumes nothing about *how* L is going to use the data, e.g., whether it runs SGD or what surrogate loss it uses, or even whether it performs Empirical Risk Minimization. In this sense, the above learning model is very abstract and we should expect only qualitative answers from this theory.

Example 13.2 (Learning Monotone Boolean Formulae). Let $x = [x_1, \ldots, x_d]$ be the datum and $c(x)$ be a conjunction, e.g.,

$$c(x) = x_1 \wedge x_3 \wedge x_4.$$

To take a few examples, $c(10011) = 0$ and $c(11110) = 1$. Such formulae are called monotone because no literals show up as negated in the formula.

We can have the hypothesis class $\mathcal{F}$ to be the set of all possible conjunctions of d variables $x_1, \dots, x_d$. Each literal x_i can be in the conjunction or not, so the total number of hypotheses in $\mathcal{F}$ is 2^d. [1] Observe that since this is exponential in d, an algorithm L that brute-force searches over $\mathcal{F}$ will have a large computational complexity. Also observe that since the true hypothesis $c \in \mathcal{F}$, there exists an answer f that the algorithm L can output that achieves zero training error, i.e.,

$$\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{f(x^i) \neq c(x^i)\}} = 0.$$

But for a fixed amount of data n, there is some probability that the minimizing hypothesis f has zero training error but large population risk. As the number of data n is large, we expect this event to be less and less probable.

Consider an algorithm L that does the following. It starts with the hypothesis

$$f^0(x) = x_1 \wedge x_2 \wedge \cdots \wedge x_d$$

with all literals and for every datum with a label 1, it deletes all literals x_i that are not in this datum to update the hypothesis f; this makes sense because if the deleted literals were zero in some input, f and c would predict different outputs. Remember that since $c(x) \in \mathcal{F}$, we cannot have a datum with input 1111...1 and output 0.

What kind of errors does this algorithm make? If some literal x_i was deleted, it is because it had the value $x_i = 0$ on a positively labeled sample. So, it should be deleted, otherwise the hypothesis will output 0. So, we only output a wrong hypothesis if more literals present than those in $c(x)$. So, the output $f(x)$ can only make an error on data labeled 1 by $c(x)$, never on the ones labeled zero. Our algorithm therefore only has false negatives.

We now see why requesting more samples diminishes the probability of this event happening. Let $p_i = \mathbb{P}_{x \sim P}\left[c(x) = 1, x_i = 0 \text{ in } x\right]$. Therefore

$$R(h) \leq \sum_{x_i \in f} p_i$$

If some p_i is small, then it does not contribute much to the error. If some p_i is large then we make sure to see enough samples so that we remove that x_i from f. After all, it only takes one appearance of this event to delete this x_i, and the event has probability p_i which is large. Rigorously, if all x_i in f have $p_i < \epsilon/d$ then $R(h) < \epsilon$. On the other hand, if some x_i has $p_i > \epsilon/d$ then the probability of having this x_i in f is the probability that the event of p_i never happens in the draw of n samples. But this new probability is smaller than $1 - \epsilon/d$. And the event will never happen in n i.i.d. draws with probability at most $(1 - \epsilon/d)^n \leq e^{-n\epsilon/d}$. Using the union bound, since there are at most d literals in f, the probability that there is at least one such "bad event" is at most $de^{-n\epsilon/d}$.

[1] Actually the total number of conjunctions is $2^d + 1$ because for the null-conjuction (without any literals) we can have the constant $c(x) = 0$ or $c(x) = 1$ for all x. We should therefore explicitly make sure $c(111\dots11) = 0$ is not in the true labeling function. But we ignore this corner case, and silently assume that only the hypothesis $c(x) = 1 \forall x$ is in our class $\mathcal{C}$.

If this bad event never happens the population risk is less than ϵ. Of course, such a bad event happening would be devastating. For some distributions it could lead the error up to 1. However, in our PAC-learning setting we can accept this as long as it happens rarely with probability at most δ. Since

$$de^{-n\epsilon/d} < \delta \iff n \geq \frac{d}{\epsilon} \log \frac{d}{\delta}$$

we are guaranteed to meet the PAC criteria: of error less than ϵ with probability at least $1-\delta$.

Note that both the sample complexity and computational complexity are polynomial. We have thus shown that the class of Monotone Boolean Formulae is (ϵ, δ)-PAC learnable.

13.2 Concentration of Measure

Two very important results from probability theory that we will use are the Union Bound and the Chernoff Bound.

13.2.1 Union Bound (or Boole's Inequality)

For any countable set of events, $\{A_1, \cdots, A_n, \cdots\}$,

$$\mathbb{P}\left(\bigcup_i A_i\right) \leq \sum_i \mathbb{P}\left[A_i\right].$$

This is a rather loose, but useful, upper bound and is (mostly) embedded in the assumptions of what we call a "probability measure" in probability theory (σ-subadditivity). This essentially means that it can be used without any extra assumptions in practice.

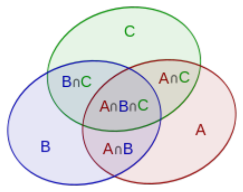
⚠ If we want a better approximation of the probability of the union of multiple events and we know more about the problem at hand we can use what are called Bonferroni inequalities.

By the inclusion-exclusion principle for finite set of events $\{A_1, \cdots, A_n\}$,

$$\mathbb{P}\left(\bigcup_{i=1}^n A_i\right) = \sum_{i=1}^n \mathbb{P}\left(A_i\right) - \sum_{1\leq i<j\leq n} \mathbb{P}\left(A_i, A_j\right) + ... + (-1)^{n-1}\, \mathbb{P}\left(A_1, A_2, \cdots, A_n\right)$$

We can get better approximations of the union, if we use the first $k \leq n$ terms above. If we stop at odd k, we get an upper bound. If we stop at even k we get a lower bound. The error of the approximation is decreasing with k.

❓ Where did we use the union bound in the proof for the PAC-learnability of the class of monotone Boolean functions?



❓ Try to prove that $\mathbb{P}\left(\bigcap_{i=1}^n A_i\right) \geq 1 - \sum_{i=1}^n \mathbb{P}\left(A_i^c\right)$

13.2.2 Chernoff Bound

Let $A_1, \cdots, A_n$ be a sequence of i.i.d. random variables. We focus on the case of Bernoulli random variables where $\mathbb{P}\left(A_i = 1\right) = p$. We would like to estimate p from samples. One way to do this is to compute the empirical average

$$\hat{p} = \frac{1}{n}\sum_{i=1}^{n} A_i$$

and estimate how close it is to the true p. We know that as $n \to \infty$

Weak Law For all $\epsilon > 0$ we have

$$\mathbb{P}\left(|\hat{p} - p| > \epsilon\right) \to 0.$$

This is also known as convergence in probability.

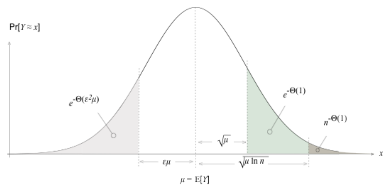
Strong Law We have almost sure convergence, i.e.,

$$\mathbb{P}\left(\lim_{n\to\infty} \hat{p} = p\right) = 1.$$

Central Limit Theorem As $n \to \infty$, the quantity $\sqrt{n}(\hat{p} - p)$ is distributed as a Normal distribution with mean zero and variance $p(1-p)$. Notice that as opposed to the law of large numbers, the central limit theorem also gives us a rate of convergence, i.e., how many samples n are necessary if want the difference to be close to a Normal distribution. If we set $\sigma^2 = p(1-p)$ we can rewrite the Central Limit Theorem as

$$\mathbb{P}\left(|\hat{p} - p| > \epsilon\right) \le 2e^{-n\epsilon^2/(2\sigma^2)}.$$

⚠ This picture makes it easy to remember concentration inequalities for an n-dimensional Gaussian random variable Y.



Chernoff Bound Since $\sigma^2 = p(1-p) < 1/4$ we have from CLT that

$$\mathbb{P}\left(\left|\frac{1}{n}\sum_i A_i - p\right| > \epsilon\right) \le 2e^{-2n\epsilon^2}.$$

An easy way to remember the Chernoff bound is that if we want the average of n random variables to be ϵ-close to their expected value with probability at least $1 - \delta$, then we need

$$n = \Omega\left(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\right)$$

samples.

❓ Do you see any patterns in the Chernoff bound with sample complexity in PAC-learning?

Concentration of measure is an exciting area of probability theory and similar results can be obtained for other distributions, other functions than averaging of random variables $A_1, \ldots, A_n$ etc. Popular inequalities are Markov's Inequality, Chebyshev's Inequality and Chernoff Bounds (and Hoeffding's Inequality as an important special case). They are written in terms of increasing tightness, but also of increasing assumptions of what we need to know in order compute them. You can read a very good introduction to this topic at https://terrytao.wordpress.com/2010/01/03/254a-notes-1-concentration-of-measure/.

In general, Markov's inequality only needs the expectation, Chebyshev's Inequality needs the variance too, while Chernoff bounds usually need the whole moment generating function. They are all applications of Markov's Inequality on higher order statistics. The value of Chernoff bounds is increasingly more important when we talk about distributions of few sufficient statistics, like the Bernoulli distribution (or any exponential distribution).

13.3 Uniform convergence

We now lift the assumption that Nature's labeling function $c \in \mathcal{C}$. After all, even if there exists such a true deterministic c we can never be sure that it is inside $\mathcal{F}$, say the class of neural networks of a specific architecture that we are using. This model is called the Agnostic PAC-Learning model.

We will stay within the confinements of Empirical Risk Minimization where we are provided with some samples where we output the hypothesis with the smallest training error

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{f(x^i) \neq y^i\}} \quad \text{minimizing this gives} \quad f_{\text{ERM}} \in \mathcal{F}.$$

The population risk is

$$R(f) = \mathop{\mathbb{E}}_{(x,y)\sim P} \left[\mathbf{1}_{\{f_{\text{ERM}}(x) \neq y\}} \right] \quad \text{minimizing this gives} \quad f^* \in \mathcal{F}.$$

Observe that f^* is *not* the Bayes optimal predictor that we saw in the bias-variance tradeoff. This is because the former is restricted to the hypothesis class $\mathcal{F}$ while the latter has no such restriction to lie in $\mathcal{F}$, it is simply the optimal hypothesis that minimizes the population risk.

Our goal while computing a *generalization bound* is to ask the following question: if we obtain an ERM hypothesis f_{ERM} with a good training error, then does this also mean that the population risk of the best hypothesis in the class f^* is small?

The above question is central, answering it in the affirmative ensures that we are using a correct hypothesis class (say neural architecture) and that the error on the training dataset is a good indicator of the performance on the entire distribution. This involves the following two steps.

1. First, we need to make sure that the difference

$$\left| \hat{R}(f_{\text{ERM}}) - R(f_{\text{ERM}}) \right| \to 0, \quad n \to \infty.$$

   This is easy, it is akin to the concentration of measure we saw in the previous section.

2. Second, we need to ensure that

$$\hat{R}(f_{\text{ERM}}) \approx \quad R(f^*)$$

   with high probability for every training dataset of n samples using which f_{ERM} is computed. If this is true, it tells us something about the ERM procedure itself, i.e., it tells us whether minimizing the empirical risk

$\hat{R}(f)$ is a good thing if we want to build a classifier that works well on the population.

This is difficult to do, after all f_{ERM} and f^* are totally different hypothesis. Vapnik set up a powerful construction to do this. He showed that a *sufficient* condition to achieve the above is that for all hypotheses in $\mathcal{F}$, the empirical risk and population risk are similar. This is known as uniform convergence.

Let us now develop the two points above. Since data are drawn iid, we can use the Chernoff bound to get that

$$\forall f \in \mathcal{F}, \mathbb{P}\left(\left|\hat{R}(f) - R(f)\right| > \epsilon\right) \leq 2e^{-2n\epsilon^2}.$$

If the hypothesis class is finite $\mathcal{F}$, we can use the union bound to show that for *any* hypothesis, the training error and population risk are close.

$$\begin{aligned}
&\mathbb{P}\left(\exists f \in \mathcal{F} : \left|\hat{R}(f) - R(f)\right| > \epsilon\right) \\
&\leq \sum_{f \in \mathcal{F}} \mathbb{P}\left(\left|\hat{R}(f) - R(f)\right| > \epsilon\right) \\
&\leq |\mathcal{F}|\, 2e^{-2n\epsilon^2}.
\end{aligned}$$

If we want this above probability of a bad event to be less than δ we therefore need

$$n \geq \frac{1}{2\epsilon^2} \log \frac{2\,|\mathcal{F}|}{\delta} \tag{13.1}$$

training data. Notice how this bound changed from the Monotone Boolean function example: we need $\mathcal{O}(1/\epsilon)$ times more samples to get the uniform convergence result.

Suppose we had a classifier f with 2% gap ($\epsilon = 0.02$) between the training error $\hat{R}(f)$ and the validation error (which is a proxy for the population risk $R(f)$), if we want to reduce this gap by half to 1% ($\epsilon = 0.01$) , we need 4 times as many training data. We could also reduce this gap by fitting a classifier with small $|\mathcal{F}|$ but in this case, both the training and validation error will increase even if their gap decreases.

Next, we need a relation between the population risk of f_{ERM} and the best possible predictor $f^*\mathcal{F}$ in our hypothesis class. Observe that

$$\begin{aligned}
R(f_{\text{ERM}}) &\leq \hat{R}(f_{\text{ERM}}) + \epsilon && \text{(Chernoff bound on } f_{\text{ERM}}\text{)} \\
&\leq \hat{R}(f^*) + \epsilon && (f_{\text{ERM}} \text{ has the smallest training error)} \\
&\leq R(f^*) + 2\epsilon && \text{(Chernoff bound on } f^*\text{).}
\end{aligned}$$

The two Chernoff bound inequalities hold with probability at least $1 - \delta$ so the final inequality

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\epsilon$$

holds with probability at least $1 - 2\delta$. Substitute this in (13.1) to get

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{2n} \log \frac{4\,|\mathcal{F}|}{\delta}} \tag{13.2}$$

with probability $1 - \delta$. A result of this kind is called a Vapnik-Chernovenkis (VC) bound or a PAC bound.

Let us consider our monotone Boolean formulae example again. Since $|\mathcal{F}| = 2^d$, if the input dimension is $d = 1000$ and we set $\delta = 10^{-3}$, the VC-bound predicts the following. In this case, we should imagine running ERM to pick the best hypothesis f_{ERM}, not the elimination algorithm we discussed in the section on monotone Boolean formulae.

1. With $n = 1000$ data, we have $R(f_{\text{ERM}}) \leq R(f^*) + 1.42$. This is vacuous/non-informative since the population risk is an expectation of indicator variables and should therefore be less than 1.

2. With $n = 10^5$, we have $R(f_{\text{ERM}}) \leq R(f^*) + 0.45$. This is informative, it means that the population risk of the classifier obtained by ERM is within 44% of the population risk of the best classifier f^* in that class. Of course it is only meaningful if f^* generalizes well, i.e., if $R(f^*)$ is small. This will happen if the hypothesis class $\mathcal{F}$ is large enough.

3. With $n = 10^6$, we have $R(f_{\text{ERM}}) \leq R(f^*) + 0.04$.

13.4 Vapnik-Chernovenkis (VC) dimension

In the above section, the concept/hypothesis class was assumed to be finite $|\mathcal{C}| < \infty$. The union bound of course breaks if this is not the case. Notice that once we pick a neural architecture (hypothesis class), the number of possible models (hypotheses), each with different weight vectors, is infinite. Observe that in the monotone Boolean formulae example, the algorithm L was using the training data to eliminate hypothesis from $\mathcal{C}$, this is not going to work $\mathcal{C}$ is not finite. It is therefore a natural question whether we can still learn a hypothesis class with a finite number of training data.

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 $f \in \mathcal{C}$.

Shattering a set of inputs We say that the set of inputs $D = \{x^1, \cdots, x^n\}$ is *shattered* by the concept class $\mathcal{C}$, if we can achieve every possible labeling out of the 2^n labellings using some concept $c \in \mathcal{C}$. The size of the largest set D that can be shattered by $\mathcal{C}$ is called the VC-dimension of the class $\mathcal{C}$. 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 $\mathcal{C}$ that can realize this labeling, then

$$\text{VC}(\mathcal{C}) \geq n.$$

On the other hand, if for every possible configuration of $n+1$ inputs, we can always find a labeling such that no hypothesis in $\mathcal{C}$ can realize this labeling, then

$$n \leq \text{VC}(\mathcal{C}).$$

If we find some n for which both of the above statements are true, then

$$\text{VC}(\mathcal{C}) = n.$$

3590 Some examples.

3591 • d-dim Linear Threshold Functions: VC-dim $= d + 1$.

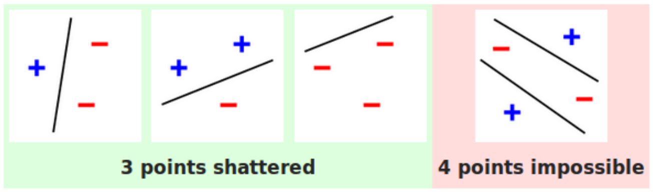


Figure 13.1: d=2: See that for the lower bound, we found some configuration of the 3 points, such that a linear threshold function always separates the points consistently with the labels; for any possible labeling. 3 such labellings are shown, convince yourselves that it can be done for all 8 cases. 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.

3592 • 2 dimensional axis aligned rectangles: VC-dim = 4 (exercise)

3593 • Monotone Boolean Formulae: VC-dim $= d$ (exercise).

3594 • If the hypothesis class is finite, then

$$\text{VC}(\mathcal{F}) \leq \log |\mathcal{F}| .$$

3595 • If $x \in \mathbb{R}$ and our concept class includes classifiers of the form

$$\text{sign}(\sin(wx))$$

3596 where w is a learned parameter, then

$$\text{VC} = \infty.$$

3597 • For a neural network with p weights and sign activation function

$$\text{VC} = \mathcal{O}(p \log p).$$

It is a deep result that if the VC-dimension of concept class is finite $V = \text{VC}(\mathcal{F}) < \infty$, then this class has the uniform convergence property (for any $f \in \mathcal{F}$, the empirical and population error are close). Therefore, we can learn this concept class agnostically (without worrying about whether Nature's labeling function c is in our hypothesis class $\mathcal{F}$ or not) in the PAC framework with

$$n = \Omega\left(\frac{V}{\epsilon^2} \log \frac{V}{\epsilon} + \frac{1}{\epsilon^2} \log \frac{1}{\delta}\right)$$

training data. If a hypothesis class as infinite VC-dimension, then it is not PAC-learnable and it also does not have the uniform convergence property.

The above result written in another form looks as follows. For a (finite or infinite) hypothesis class $\mathcal{F}$ with finite VC-dimension $V = \text{VC}(\mathcal{F})$

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{n}\left(2V - \log\delta\right)} \tag{13.3}$$

with probability at least $1-\delta$. This is an important expression to remember: the number of samples n required to learn a concept class scales linearly with the VC-dimension V. A more refined version of this bound looks like

$$R(f_{\text{ERM}}) \leq R(f^*) + 2\sqrt{\frac{1}{n}\left(V\left(\log\frac{2n}{V}+1\right) + \log\frac{4}{\delta}\right)}. \tag{13.4}$$

Bounds on the VC-dimension of deep neural networks For general classifiers, it is typically difficult to compute the VC dimension. One instead finds upper and lower bounds for the VC dimension to be used in inequalities of the form (13.4). Bounds on the VC-dimension of deep network architectures are available (Bartlett et al., 2019). With p weights and L layers, an essentially tight VC-dimension looks like

$$\Omega\left(p\, L \log\frac{p}{L}\right) = \text{VC}(\mathcal{F}) = \mathcal{O}(p\, L\log p)$$

for deep networks with ReLU nonlinearities.

This bound is not entirely useful in the VC-theory however. For instance, the ALL-CNN network you used in your homework with $p \approx 10^6$ and $L = 10$ has VC $\approx 10^8$. If we use the coarse VC-bound in (13.3) with $n = 50,000$ samples, we have

$$R(f_{\text{ERM}}) \leq R(f^*) + 40$$

which is a vacuous generalization bound. However, remember that this is simply an *upper bound* on the generalization error of ERM. It is clear from empirical results in the literature (including your homework problems) that deep networks indeed generalize well to new data outside the training set and that means $R(f_{\text{ERM}})$ is small.

The gap in applying VC-theory to deep networks therefore likely stems from the need for uniform convergence: we may not need that the empirical and population risk are close for *all* hypotheses in the class. If we only have uniform convergence within a small subset $F \subset \mathcal{F}$ and if $\text{VC}(F) \ll \text{VC}(\mathcal{F})$ and if the training algorithms like SGD always find ERM minimizers $f_{\text{ERM}} \in F$, then VC-theory/PAC-Learning do predict that deep networks will generalize well. Understanding this is the subject of a large body of ongoing research.

Chapter 14

Variational Inference

Reading

1. Sections 1-2 of "Variational Inference: A Review for Statisticians" by Blei et al. (2017).
2. Sections 1-5 of "Auto-Encoding Variational Bayes" by Kingma and Welling (2013)
3. Chapter 2 of Durk Kingma's thesis: https://pure.uva.nl/ws/files/17891313/Thesis.pdf.
4. Bishop Chapter 11.5-11.6
5. Bishop Chapter 10-10.3
6. Lots of great intuition at http://ruishu.io/2018/03/14/vae/

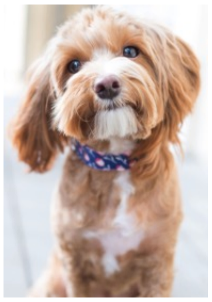
We have been primarily concerned with models for classification and regression as yet in this course. The task there is to match the target (a class identity or a real-valued outcome). We now change tracks to consider generative modeling, these are models that are trained to synthesize new data. Effectively, the task here is not *match* a target datum, but given a training dataset of images/text, create a model that outputs similar images/text at test time. We will first take a look at variational methods and generative modeling using these methods in this chapter and do implicit generative models such as Generative Adversarial Networks in the next chapter.

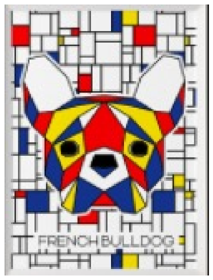
14.1 The model

Imagine how you would draw the image of a dog x on paper. First, you would decide in your mind, its breed, its age, the color of its fur etc. Let us call these quantities "latent factors". Latent factors can also include things that are not specific to the dog, e.g., the background of your painting (grass, house, beach etc.), the weather on that day (cloudy, sunny etc.), the viewpoint (zoomed in/far away). We will denote all such quantities by

$$z := \text{latent factors}.$$

Having decided upon all these factors, you realize your painting x. The painting x is not unique given latent factors z, e.g., two people can start off with the same latent factors and draw two totally different pictures.





We therefore model the generative process as a obtaining samples from a probability distribution

$$p(x|z).$$

Given a latent factor z and an image x, the quantity $p(x|z)$ denotes the likelihood of the sample. Given the painting image x, we do not know what the latent factors are. For instance, it is not easy to say whether the following image is that of a cat or a dog.

In other words, the latent factors of data x are not known to us if we do not take part in the generative process. Nature is in charge of generating the data and our goal here is to guess the parameters of this generative model to be able to synthesize new samples that look as if Nature generated them.

There can be lots of latent factors z. So let us control this complexity and assume that we know a prior over the latent factors

$$\text{prior } p(z)$$

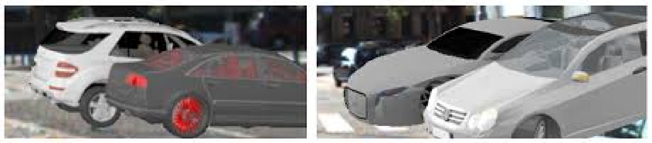
that models our belief of how likely a factor "dog with color blue" is in Nature.

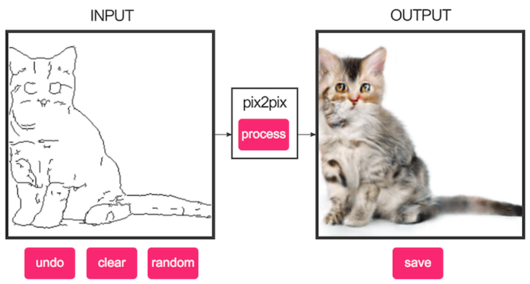
Let us imagine Nature's generative model as running in two steps

1. First, sample a latent factor z from some distribution, and then
2. sample a datum $x \sim p(x|z)$.

The central point to appreciate is that we know neither Nature's distribution for sampling latents z nor its generative model $p(x|z)$. We will need to fit both these quantities using a training dataset of images/text.

The purpose of doing so can be many-fold, e.g., we may want to generate new data to amplify the size of our training set, given a part of the input image (say due to occlusions, or image corruption) we may want to complete the rest of it.





Most such applications require the knowledge of the latent factors that generated the data. Therefore, formally, we are interested in computing the posterior distribution

$$\text{posterior } p(z|x)$$

given the prior distribution $p(z)$ and samples in a training dataset $D = \{x^i\}_{i=1}^n$. Notice that we do not need labels for this problem, effectively labels $y^i = x^i$ itself because our generative model should of course be very good at generating samples from the training data.

14.2 Some technical basics

14.2.1 Variational calculus

We will first take a brief look at what is called variational calculus.

A function is something takes in a variable as input and returns the value of the function as the output, e.g., $\mathbb{R} \ni f(x) = 5\,x^2$ for $x \in \mathbb{R}$. Similarly, a *functional* is an object that takes in a *function* as an input and returns a real number as the output. An example of this is entropy

$$\mathbb{R} \ni H[p] = -\int p(x) \log p(x)\, \mathrm{d}x$$

which takes in a probability density p as the input and returns a real number. Entropy is therefore a *functional*. Just like standard calculus where we take derivatives/minimize over functions, we can also take derivatives of the functional.

The functional derivative $\frac{\delta H[p]}{\delta p}(x)$ is defined in a funny way as

$$\int \frac{\delta H[p]}{\delta p}(x)\, \varphi(x)\, \mathrm{d}x = \lim_{\epsilon \to 0} \frac{H[p + \epsilon\varphi] - H[p]}{\epsilon}$$

for any arbitrary function φ. Essentially, you perturb the argument to the functional p by some epsilon and see how much the functional changes. The change in the functional is measured using the test function φ by integrating its changes $\frac{\delta H(p)}{\delta p}(x)$ at each point x in the domain. There may be certain conditions that the perturbation φ needs to satisfy, e.g., since $p+\epsilon\varphi$ should also be probability density, the functional derivative above should only consider test functions φ such that

$$\forall \epsilon \int (p(x) + \epsilon\varphi(x))\, \mathrm{d}x = 1 \Rightarrow \int \varphi(x) \mathrm{d}x = 0.$$

The KL-divergence between two probability densities,

$$\text{KL}(p \,||\, q) = \int p(x) \log \frac{p(x)}{q(x)}\, \mathrm{d}x,$$

is another such functional; it has two arguments p and q.

Variational optimization is concerned with minimizing functionals. For instance, if we have a problem

$$w^* = \underset{w \in \mathbb{R}^p}{\text{argmin}}\, \ell(w)$$

in standard optimization, a variational optimization problem with KL-divergence as the loss given a fixed density p looks like

$$q^* = \underset{q \in \mathcal{Q}}{\text{argmin}}\, \text{KL}(q \,||\, p). \tag{14.1}$$

The variable of optimization is the probability density q and we will denote the domain of the variable by $\mathcal{Q}$. Since we want q to be a legitimate probability density, we should choose

$$\mathcal{Q} \subseteq \mathcal{P}(\mathcal{X})$$

where $\mathcal{P}(\mathcal{X})$ denotes the set of all probability densities on some domain $\mathcal{X}$.

Picking the domain and objective in variational optimization Picking a good domain $\mathcal{Q}$ to minimize over is important. It is similar to the notion of the a hypothesis class in machine learning. If $\mathcal{Q}$ is too big, it is difficult to solve the optimization problem but we obtain a better value to KL$(q||p)$. If $\mathcal{Q}$

is too small, the optimization problem may be easy but we may not match the desired distribution p very well. Imagine if p is a mixture of two Gaussians and we pick $\mathcal{Q}$ to be a family of uni-modal Gaussian distributions. Since the KL-divergence is zero if and only if the two distributions are equal, we are never going to be able to minimize it completely. On the other hand, if we pick $\mathcal{Q}$ to be the family of distributions with 2 or more Gaussian modes, then we can perfectly match p. Essentially, the crux of variational inference boils down to picking a good family of distributions $\mathcal{Q}$ that makes solving (14.1) easy.

What functional should we use to measure the distance between q and p? The KL-divergence is popular and easy to use in practice but there are many others. For example, when we studied the Gibbs distribution we briefly talked about something called "Wassserstein metric": if one imagines a mountain of dirt given by distribution q and another mountain of dirt p, the Wassserstein distance $W_2(q, p)$ is the amount of work done in transporting the dirt from q to p; it is also called the "earth mover's distance". The Wassserstein metric is as legitimate a distance between two distributions as the Kullback-Leibler divergence.

14.2.2 Laplace approximation

Laplace approximation is a very useful trick that is similar to variational inference. Here is how it works: suppose we have to estimate approximately an expectation of our random variable $\varphi(w)$

$$\mathop{\mathbb{E}}_{w \sim e^{-n\ell(w)}} [\varphi(w)] = \int e^{-nf(w)}\, \varphi(w)\, \mathrm{d}w$$

for some large value of n. The above integral takes many values, some have small $\ell(w)$ and some have large $\ell(w)$. The values of w where $\ell(w)$ is small are the ones that have the highest $e^{-n\ell(w)}$, especially as $n \to \infty$, and therefore the ones that we should pay most attention while approximating the integral. For large n, Laplace approximation replaces the above integral by simply

$$\begin{aligned}\int e^{-n\ell(w)}\, \varphi(w)\, \mathrm{d}w &\approx \int \varphi(w)\, e^{-n\left(\ell(w^*) + \frac{1}{2}(w-w^*)^\top\, \nabla^2\, \ell(w^*)(w-w^*)\right)}\, \mathrm{d}w \\ &= e^{-n\ell(w^*)} \int \varphi(w)\, e^{-\frac{n}{2}(w-w^*)^\top\, \nabla^2\, \ell(w^*)(w-w^*)}\, \mathrm{d}w\end{aligned} \tag{14.2}$$

where $w^* = \text{argmin}\, \ell(w)$ is the global minimum of $\ell(w)$. The integral is now with respect to a Gaussian distribution and can be done more easily.

How does a variational approximation differ from the Laplace approximation? Let us look at an example.

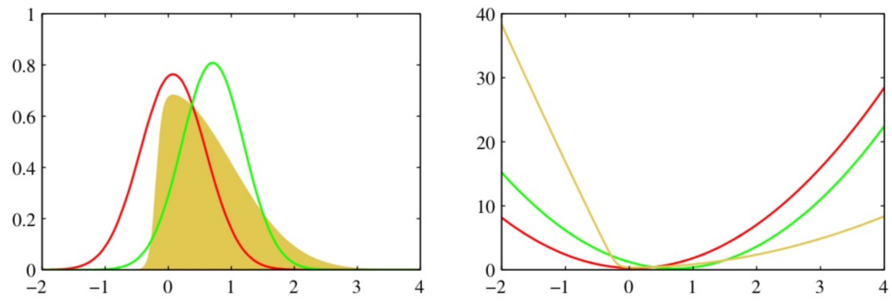


Figure 10.1 Illustration of the variational approximation for the example considered earlier in Figure 4.14. The left-hand plot shows the original distribution (yellow) along with the Laplace (red) and variational (green) approximations, and the right-hand plot shows the negative logarithms of the corresponding curves.

14.2.3 Digging deeper into KL-divergence

Let us take an example to understand KL-divergence better.

Figure 14.1 compares two forms of KL-divergence. The green contours represent equi-probability lines (1,2,3 standard deviations) for a two-dimensional correlated Gaussian $p(z_1, z_2)$. Red contours represent similar equi-probability lines for the variational approximation of this distribution using an uncorrelated Gaussian distribution

$$q(z) = q_1(z_1) q_2(z_2)$$

where both q_1, q_2 are one-dimensional Gaussians. The variational family $q \in \mathcal{Q}$ thus consists of factored uncorrelated Gaussians and we are trying to find the best member of this family that approximates the *correlated* true distribution $p(z)$.

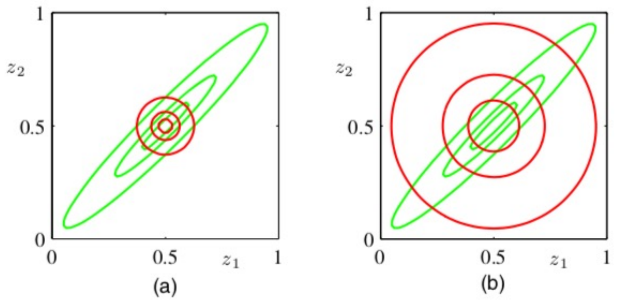


Figure 14.1: Comparison between the variational approximation of a correlated Gaussian using forward and reverse KL divergence and a factored Gaussian family.

Left panel (a) in Figure 14.1 shows the result using the forward KL-divergence minimization

$$q^* = \text{KL}(q \,||\, p).$$

while the right panel (b) shows the result for the reverse KL-divergence minimization

$$q^* = \text{KL}(p \,||\, q).$$

We see that both these forms capture the mean of the true distribution $p(z)$ correctly. The variance of the two approximations is quite different depending upon which form we employ.

❓ Use the expression of the KL-divergence to convince yourself why the forward KL under-estimates the variance while the reverse KL over-estimates the variance in Figure 14.1.

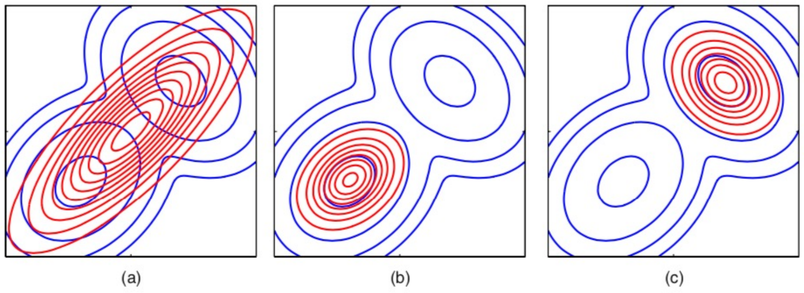


Figure 14.2: Approximating a multi-modal distribution using a uni-modal variational family.

We next consider the case when a multi-modal probability distribution $p(z)$ is approximated using a unimodal Gaussian distribution. Both these examples are very often seen in practice, the distribution of true data/latent factors is often correlated and multi-modal. We have seen one instance of this: the distribution of weights of a deep network in the Gibbs distribution is multi-modal because of multiple global minima.

In panel (a) of Figure 14.2, the reverse KL divergence $\text{KL}(p \,||q)$ is used to obtain the red contours of q^*. In contrast the forward KL divergence $\text{KL}(p \,||\, q)$ is used to obtain q^* in panels (b) and (c). Note that the distribution p is bi-modal and the variational problem is no longer convex in this case; depending upon the initial condition using q, one may get different solutions shown in panels (b) and (c).

KL-divergence is not the only distance used in variational inference and there are many many other ones. You should think of these different ways to measure distances between probability distributions in variational inference as different surrogate losses; which one we use is highly problem dependent although the forward KL-divergence $\text{KL}(q \,||\, p)$ is the most common.

14.3 Evidence Lower Bound (ELBO)

We now go back to the generative model.

We will formalize our goal in generative modeling as computing

Nature's true posterior distribution of latent factors

$$p(z|x).$$

We have access to a training dataset $D = \left\{(x^i)\right\}_{i=1}^n$. We do not know (i) what form Nature's posterior distribution takes, e.g., Gaussian, multi-modal distribution etc. and (ii) we do not know the true latent factors z that Nature uses. So we are going to approximate the true posterior using some variational family of our choice

$$\mathcal{Q} \ni q^*(z|x) \approx p(z|x).$$

This is the basic idea of variational inference: to approximate a complex distribution $p(z|x)$ using a member of from a simpler family of our choosing $\mathcal{Q}$. In practice, this variational family $\mathcal{Q}$ will be parameterized by a deep network.

With this background, the mathematical process of executing the above program is quite simple. We will simply minimize the KL-divergence

$$q^*(z|x) = \operatorname*{argmin}_{q \in \mathcal{Q}} \frac{1}{n} \sum_{i=1}^n \text{KL}\left(q(z|x^i) \mid\mid p(z|x^i)\right). \tag{14.3}$$

We next rewrite this KL-divergence above in a special form.

$$\begin{aligned}
0 &\leq \text{KL}\left(q(z|x^i) \mid\mid p(z|x^i)\right) \\
&= \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log \frac{q(z|x^i)}{p(z|x^i)}\right] \\
&= -\mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log p(z|x^i)\right] + \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log q(z|x^i)\right] \\
&= -\mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log p(z, x^i) - \log p(x^i)\right] + \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log q(z|x^i)\right] \\
&= \log p(x^i) - \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log p(z, x^i)\right] + \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log q(z|x^i)\right]. \\
\Rightarrow \log p(x^i) &\geq \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log p(z, x^i)\right] - \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log q(z|x^i)\right]
\end{aligned}$$

This is quite interesting. The left-hand side of this inequality is the log-likelihood of the data under Nature's distribution, i.e., it is fixed and independent of what we do. The left-hand side is also called the evidence. The right hand-side

$$\text{ELBO}(q, x^i) := \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log p(z, x^i)\right] - \mathop{\mathbb{E}}_{z\sim q(z|x^i)} \left[\log q(z|x^i)\right]. \tag{14.4}$$

is a lower bound on the evidence and therefore called the Evidence Lower Bound (ELBO).

Next comes a key step: a good generative model should be such that the evidence of the training data, i.e., the log-likelihood of this data under Nature's distribution, should be large under the model. We therefore want to maximize the ELBO on our training data

$$q^*(z|x) = \underset{q \in \mathcal{Q}}{\text{argmax}} \frac{1}{n} \sum_{i=1}^{n} \text{ELBO}(q, x^i). \tag{14.5}$$

to find the posterior distribution of the latent factors $q^*(z)$. Maximizing ELBO is equivalent to minimizing the KL-divergence $\text{KL}(q(z|x^i) \,||\, p(z|x^i)$.

We will again solve the optimization problem in (14.5) using stochastic gradient descent. Before we study how to do that, let us consider what model we have developed so far. The solution to this problem

$$q^*(z|x) \approx p(z|x)$$

approximates Nature's posterior distribution. If we maximize ELBO well, given an input x, samples $z \sim q^*(z|x)$ are likely to be the latent factors that Nature could have chosen while rendering this image. But we still do not know how to synthesize an image x for these latent factors. We now rewrite ELBO in a different form to understand this.

$$\begin{aligned}
\text{ELBO}(q, x^i) &= \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log p(z, x^i)\right] - \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log q(z|x^i)\right] \\
&= \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log p(x^i|z) + \log p(z)\right] - \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log q(z|x^i)\right] \\
&= \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log p(x^i|z)\right] - \text{KL}(q(z|x^i) \,||\, p(z)).
\end{aligned}$$

This form of ELBO

$$\text{ELBO}(q, x^i) = \underset{z \sim q(z|x^i)}{\mathbb{E}} \left[\log p(x^i|z)\right] - \text{KL}(q(z|x^i) \,||\, p(z)) \tag{14.6}$$

is very interesting. The first term is Nature's log-likelihood of datum x^i given the latent factor z sampled from *our* candidate posterior q. The second term is the discrepancy between our variational approximation of the posterior $q^*(z|x^i) \approx p(z|x^i)$ and Nature's true marginal distribution over latent factors $p(z)$. This alternative form of ELBO is conceptually very similar to what we do in standard classification, e.g.,

$$\underset{w}{\text{argmin}} \left\{ \ell(w) + \frac{\alpha}{2} \|w\|^2 \right\}.$$

We would like our $q(z|x^i)$ to be close to Nature's prior distribution $p(z)$ but at the same time be such that samples from $q(z|x^i)$ have a high log-likelihood $p(x^i|z)$ of synthesizing images in the training set. The KL-term is therefore a regularizer for the first data-fitting term.

14.3.1 Parameterizing ELBO

What variational family $\mathcal{Q}$ should we choose? Say we parametrized each distribution $q(z|x^i)$ by its mean and diagonal of the covariance.

$$\mathbb{R}^m \ni z \sim q(z|x^i) = N(\mu(x^i), \sigma^2(x^i) I) \in \mathcal{Q}(x^i)$$

where $\mu(x^i), \sigma^2(x^i) \in \mathbb{R}^m$. The ELBO in (14.6) is totally independent for each x^i in the training dataset, so all $i \in \{1, \dots, n\}$ we can solve for

$$\mu^*(x^i), \sigma^2(x^i) = \underset{\mu, \sigma^2}{\text{argmax}}\ \text{ELBO}\left(N(\mu(x^i), \sigma^2(x^i) I), x^i\right).$$

But this is not a good idea: the parameters μ, σ^2 are distinct for each input x^i and effectively they are being trained using a dataset of only input image x^i.

Amortized variational inference is a clever trick that ties together the variational families $\mathcal{Q}(x^i)$. We will be using a deep network with parameters $u \in \mathbb{R}^p$ that takes x^i as the input and gives $\mu(x^i; u), \sigma^2(x^i; u)$ as the outputs

$$\text{Encoder}: x^i \underbrace{\mapsto}_{\text{parameters } u} \mu(x^i; u), \sigma^2(x^i; u).$$

The variational family $\mathcal{Q}(x^i)$ that we are considering is therefore the set of distributions expressed by this deep network with p parameters. The family $\mathcal{Q}(x^i)$ is still distinct for each datum x^i but they are are all tied together by the same weights u.

Encoder. We will call this deep network the encoder because it takes in an input x^i and encodes it into $\mu(x^i; u), \sigma^2(x^i; u)$ which parameterize the distribution of the latent factors.

Decoder. Observe that although we have now parameterized the distribution $q(z|x^i)$ using a deep network with weights u, we still do not know how to model the term $p(x^i|z)$. After all, this is Nature's log-likelihood.

We have a dataset $\left\{(x^i, z^i)\right\}_{i=1}^n$ that consists of the images x^i and their corresponding latents z^i sampled from our encoder. We are going to model Nature's rendering process $p(x|z)$ using a deep network. This is a program that we have done many times in the past, e.g., we model the targets in classification y^i as samples from the softmax distribution with images x^i as the input and train the weights using maximum-likelihood (as you may recall, this is equivalent to the cross-entropy loss).

We can repeat that program here: we are going to learn a deep network

$$\text{Decoder}: p_v(x^i|z) \approx p(x^i|z).$$

with parameters $v \in \mathbb{R}^p$ that models Nature's likelihood $p(x^i|z)$.

Different possible decoders for MNIST Depending upon the type of data x^i, we will code up the deep network in different ways. For instance, if each pixel of $x^i \in \mathbb{R}^{28\times 28}$ is grayscale $[0, 255]$ like it is in MNIST, the output of the decoder is a multinomial with size $28 \times 28 \times 256$.

⚠ The distribution of labels y^i in classification was one-hot vectors, so the softmax layer created a multinomial distribution on the classes.

If we take the training dataset as binarized MNIST (if pixel jk is less than 128 set it to 0, else set it to 1), then the output of the decoder has size $28 \times 28 \times 2$ and we can fit this using a logistic distribution at each pixel

$$p_v(x^i|z) = \prod_{j,k=1}^{28} \underbrace{p_v(x^i_{jk}|z)}_{\text{logistic distribution for pixel } x^i_{jk} \in \{0,1\}}$$

The log-likelihood term in (14.6) will then correspond to the logistic loss as discussed in the Homework.

Using a mean-field prior $p(z)$. We do not know what the prior distribution $p(z)$ in (14.6) is. We will choose a simple prior

$$p(z) = \prod_{j=1}^{m} p_j(z_j) \tag{14.7}$$

where $p_i(z_i)$ is the distribution of the i^{th} latent factor z_i. Such distributions are called mean-field priors (where the distribution of a vector $z \in \mathbb{R}^m$ is modeled as independent distributions on its components). We will further choose each distribution

$$p_j(z_j) = N(0, 1)$$

to be a zero-mean standard Gaussian distribution. This is a Gaussian mean-field prior. Just like the choice of a regularizer is critical in machine learning for obtaining good generalization, the chose of a prior is critical in variational inference for synthesizing good images from the generative model.

14.4 Gradient of the ELBO

We now have all the ingredients in place for training a variational generative model. Let us summarize our setup.

1. Encoder parameters u are weights of a deep network that takes in x^i as input and outputs parameters $\mu(x^i), \sigma^2(x^i)$ of the latent distribution. We have tacitly assumed the latent posterior $p(z|x^i)$ to be a Gaussian here; if you have a problem where you wish to have a different latent, e.g., all the latent genes that could have caused a particular cancer, then you want to output the parameters of that distribution from the encoder.

2. The decoder models the likelihood $p_v(x^i|z)$ using parameters v.

3. The prior $p(z)$ will be a mean-field Gaussian distribution. The prior has no parameters in our case, although you may see research papers where the prior also has its own parameters. A popular choice is to use

$$\text{ELBO}_\beta(q, x^i) = \mathop{\mathbb{E}}_{z \sim q(z|x^i)} \left[\log p(x^i|z)\right] - \beta^{-1}\, \text{KL}(q(z|x^i) \mid\mid p(z))$$

in place of the standard ELBO. The hyper-parameter $\beta > 0$ gives more control over the strength of the prior; this is of course akin to picking the weight-decay coefficient.

⚠ The concept of variational inference and ELBO are much more general than generative models or the encoder-decoder structure that we have developed. Go through the assigned reading material to learn more.

The ELBO when rewritten in terms of the encoder and decoder parameters looks as follows.

$$\text{ELBO}(u, v; x^i) = \mathop{\mathbb{E}}_{z \sim q_u(z|x^i)} \left[\log p_v(x^i|z)\right] - \text{KL}(q_u(z|x^i) \mid\mid p(z)). \tag{14.8}$$

Our goal is to fit the weights u, v using

$$u^*, v^* = \mathop{\text{argmax}}_{u,v \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \text{ELBO}(u, v; x^i). \tag{14.9}$$

The number of parameters of the encoder and decoder can be different but for clarity we imagine them to be the same.

(14.9) is an optimization problem and in this section, we will see how to compute the gradient of the objective so that we can solve the problem using SGD.

14.4.1 The Reparameterization Trick

Focus on the gradient with respect to u of the first term of ELBO

$$\nabla_u \mathop{\mathbb{E}}_{z \sim q(z|x^i)} \left[\varphi(z)\right].$$

We have written $\log p_v(x^i|z) = \varphi(z)$ to keep the notation clear; we do not care about the exact form of the integrand in this section.

If we draw a computational graph for the forward propagation of this term, it looks as follows

$$u, x^i \to \text{sample } z \text{ from } q_u(z|x^i) \to \varphi(z).$$

The intermediate sampling step is troublesome, we do not really know how to use the chain rule of calculus across sampling, i.e., given

$$\overline{\varphi(z)} := \frac{\mathrm{d}}{\mathrm{d}_u} \varphi(z)$$

we need to compute $\overline{u} = \mathrm{d}\ell/\mathrm{d}_u\ u$. We only know how to apply the chain rule for *deterministic operations* of the form

$$u, x^i \to z = \text{some deterministic function } g(u, x^i) \to \varphi(z),$$

in which case we use the standard backprop across the function g.

The Reparameterization Trick enables us to obtain backpropagation gradients across sampling operations via a creative use of the Laplace approximation of the distribution $q_u(z|x^i)$.

We known from the Laplace approximation that we can compute an expectation over z using a Gaussian centered at the global maximum of the distribution $q_u(z|x^i)$ with variance equal to the inverse Hessian at that maximum. Motivated by this, the Reparameterization Trick *rewrites* the random variable z as

$$z = \mu(x^i; u) + \sigma(x^i; u) \odot \epsilon$$

where

$$\epsilon \sim N(0, I_{m\times m})$$

is a sample from a standard multi-variate Gaussian distribution and the notation $\odot$ denotes element-wise product. Effectively, we imagine that the encoder outputs

$$\begin{aligned}\mu(x^i; u) &= \operatorname*{argmax}_z q_u(z|x^i)\\ \sigma^2(x^i; u) &= \text{diag}\left(\left[\nabla_z^2\, q_u(z|x^i)\right]^{-1}\right).\end{aligned}$$

Just like the integral in (14.2) was performed over the Gaussian, the integral over z can be rewritten as an integral over ϵ

$$\begin{aligned}\nabla_u \mathop{\mathbb{E}}_{z\sim q_u(z|x^i)}[\varphi(z)] &= \nabla_u \mathop{\mathbb{E}}_{\epsilon\sim N(0,I)}\left[\varphi\left(\mu(x^i;u) + \sigma(x^i;u)\odot\epsilon\right)\right]\\ &= \mathop{\mathbb{E}}_{\epsilon\sim N(0,I)}\left[\nabla_u\,\varphi\left(\mu(x^i;u) + \sigma(x^i;u)\odot\epsilon\right)\right]\\ &\approx \frac{1}{N}\sum_{j=1}^N \nabla_u\,\varphi\left(\mu(x^i;u) + \sigma(x^i;u)\odot\epsilon^j\right),\ \text{where } \epsilon^j \sim N(0,I).\end{aligned}$$

We can take the gradient operator inside the expectation in this case because ϵ no longer depends on the weights u. The term $\nabla_u\,\varphi\left(\mu(x^i;u) + \sigma(x^i;u)\odot\epsilon^j\right)$ is a deterministic operation given a sample z^j and can be computed using standard backpropagation.

14.4.2 Score-function estimator of the gradient

Let us look at an alternative way to compute the same gradient.

$$\begin{aligned}\nabla_u \mathop{\mathbb{E}}_{z\sim q_u(z|x^i)}[\varphi(z)] &= \nabla_u \int \varphi(z)\; q_u(z|x^i)\,\mathrm{d}z\\ &= \int \varphi(z)\;\; \nabla_u\, q_u(z|x^i)\,\mathrm{d}z\\ &= \int \varphi(z)\; \frac{\nabla_u\, q_u(z|x^i)}{q_u(z|x^i)}\; q_u(z|x^i)\,\mathrm{d}z\\ &= \int \varphi(z)\; \nabla_u \log q_u(z|x^i)\; q_u(z|x^i)\,\mathrm{d}z\\ &= \mathop{\mathbb{E}}_{z\sim q_u(z|x^i)}\left[\varphi(z)\,\nabla_u \log q_u(z|x^i)\right]\\ &\approx \frac{1}{N}\sum_{j=1}^N \varphi(z^j)\,\nabla_u \log q_u(z^j|x^i)\,, \text{with } z^j \sim q_u(z|x^i).\end{aligned} \tag{14.10}$$

The term

$$\frac{\nabla_u\, q_u(z|x^i)}{q_u(z|x^i)} = \nabla_u \log q_u(z|x^i) \tag{14.11}$$

is called the score function of a probability distribution q_u. The above calculation is quite beautiful: calculating the gradient of the expectation of any quantity $\varphi(z)$ is equal to the expectation of the same quantity weighted by the score function

$$\nabla_u \mathop{\mathbb{E}}_{z \sim q_u} [\varphi(z)] = \mathop{\mathbb{E}}_{z \sim q_u} [\varphi(z) \, \nabla_u \log q_u] \, .$$

Due to this trick, we can compute the gradient using N samples

$$z^j \sim p_u(z|x^i) \tag{14.12}$$

from the encoder; this is easy if, say, the encoder outputs the mean and standard-deviation of the distribution of the latents. Given z^j, the gradient

$$\nabla_u \log q_u(z^j|x^i)$$

is just the standard back-propagation gradient of the quantity $\log q_u(z^j|x^i)$ with respect to weights u of the deep network and can be computed using autograd.

The key difference between the Reparameterization Trick and the score-function estimator is that in the latter, we do not need to make sure that the gradient $\mathrm{d}\ell/\mathrm{d}z^j$ can be back-propagated across the sampling operation. The score-function estimator directly computes the gradient of the entire expectation by a weighted average across the samples.

Having two different ways of computing the same gradient may seem redundant but they both are suited to very different applications. The Reparameterization Trick is not accurate in cases when the distribution $q_u(z|x^i)$ is multi-modal because we have only one mean $\mu(x^i)$ around which the samples are drawn. The score-function trick does not have this problem because so long as iid samples are drawn in (14.12) (using any method, e.g., importance sampling) we obtain true estimate of the gradient. The problem in score-function estimator lies in that the denominator $q_u(z|x^i)$ in (14.11) can take very small values if the particular sample z is unlikely. The summation (14.10) is a combination of many N, some very large in magnitude and some very small; the variance of score-function estimate of the gradient in (14.10) can therefore be quite large in most problems.

Typically, the Reparameterization Trick is commonly used in generative models while both the Reparameterization Trick and the score-function estimator are used widely in Reinforcement Learning.

14.4.3 Gradient of the remaining terms in ELBO

The gradient with respect to weights v of the decoder of the first term in ELBO

$$\nabla_v \mathop{\mathbb{E}}_{z \sim q_u(z|x^i)} \left[\log p_v(x^i|z)\right]$$

is simply the standard backpropagation gradient (the sampling distribution of the encoder does not depend on the weights of the decoder).

Let us focus on the second term

$$\mathrm{KL}\left(q_u(z|x^i) \,||\, \prod_{j=1}^{m} p_j(z_j)\right) . \tag{14.13}$$

where $p_j(z_j) = N(0,1)$ are terms of the mean-field prior. The gradient of this term with respect to weights of the decoder is zero

$$\nabla_v \text{ KL}\left(q_u(z|x^i) \;||\; \prod_{j=1}^m p_j(z_j)\right) = 0.$$

Following the reasoning in the Reparameterization Trick, we are positing that $q_u(z|x^i)$ is a Gaussian distribution:

$$q_u(z|x^i) = N\left(\mu(x^i;u), \sigma^2(x^i;u)I\right).$$

Notice that $\sigma^2(x^i;u) \in \mathbb{R}^m$ is the diagonal of the covariance and therefore the individual marginals $q_u(z_j|x^i)$ and $q_u(z_{j'}|x^i)$ for two indices j, j' are independent. We can therefore write

$$q_u(z|x^i) = \prod_{j=1}^m N(\mu_j(x^i;u), \sigma_j^2(x^i;u)). \tag{14.14}$$

The KL-divergence of a univariate Gaussian $N(\mu_1, \sigma_1^2)$ with respect to the standard Gaussian is

$$\text{KL}\left(N(\mu,\sigma^2) \;||\; N(0,1)\right) = \log\frac{1}{\sigma} + \frac{\sigma^2+\mu^2}{2} - \frac{1}{2}. \tag{14.15}$$

The general formula is

$$\text{KL}\left(N(\mu_1,\sigma_1^2) \;||\; N(\mu_2,\sigma_2^2)\right) = \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2+(\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}.$$

❓ Prove that

$$\text{KL}\left(\prod_{j=1}^m q_j(z_j) \;||\; \prod_{j=1}^m p_j(z_j)\right) = \sum_{j=1}^m \text{KL}(q_j(z_j) \;||p_j(z_j)).$$

Due to (14.14), the KL-divergence in (14.13) is a sum of the KL-divergences of the individual univariate Gaussians

$$\text{KL}(q_u(z|x^i) \;||p(z)) = -\frac{1}{2}\sum_{j=1}^m \left(\log\sigma_j^2(x^i;u) - \sigma_j^2(x^i;u) + \mu_j^2(x^i;u) + 1\right). \tag{14.16}$$

The right-hand side of this equation is only a function of u and its gradient can be calculated using standard back-propagation.

This completes our development of ELBO. Using the gradient calculated in this section, we can use SGD to maximize the objective in (14.5) and train a generative model.

14.5 Some comments

Although the mathematics of ELBO seems complicated, it is quite easy to implement generative models using variational inference in practice. You did for a simple MNIST problem in the homework/recitation but if the encoder and decoder are convolutional and deconvolutional architectures respectively, we can get very sophisticated generative models.

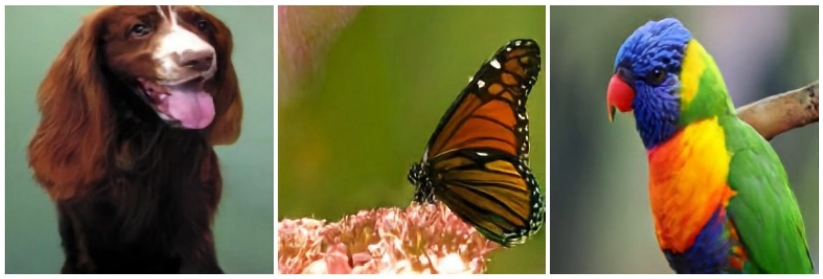


Figure 14.3: Samples from a state-of-the-art VAE trained on ImageNet (Razavi et al., 2019)

Variational inference and information-theoretic methods are a rich (and old) area of research and there are many modifications/innovations to ELBO, e.g., read Alemi et al. (2018) for some simple yet deep modifications.

Chapter 15

Generative Adversarial Networks

Reading

1. Andrew Ng's notes on generative models
   http://cs229.stanford.edu/notes/cs229-notes2.pdf
2. The original GAN paper by Goodfellow et al. (2014)
3. "The Numerics of GANs" by Mescheder et al. (2017)

In the previous chapter, we used variational methods to build a generative model for the data. In this case, we are given samples $D = \{x^i\}_{i=1}^n$ and would like to build a model that can synthesize new data. For every data x that a decoder synthesizes at test time using latent variables z, we can calculate the likelihood

$$x \sim p_v(x|z), \text{ for any } z \sim N(0, I).$$

This likelihood is an indicator of how unlikely the data x is under z. Models for which we can calculate such likelihood are called explicit generative models, i.e., they give a sample x and also report its likelihood. In this chapter, we will look an alternative class of generative models that are implicit, i.e., they only give a sample x but do not report its likelihood.

A Generative Adversarial Network (GAN) consists of two neural networks: a Generator and a Discriminator. The Generator works in the same way as the decoder in a variational auto-encoder. Given a sample z from some distribution, most commonly a standard normal, we train a neural network to generate a sample

$$x = g_v(z).$$

GANs differ from explicit models in how they train the generator, the discriminator is used for this purpose. We will look at this next.

15.1 Two-sample tests and Discriminators

We will first take a short trip into an area of statistics known as decision theory. Consider two datasets coming from two distributions $p(x)$ and $q(x)$

$$D_1 = \left\{x^1, \ldots, x^n, : \; x^k \sim p(x)\right\}$$
$$D_2 = \left\{x^1, \ldots, x^n, : \; x^k \sim q(x)\right\}.$$

We would like to check if these two distributions are the same given access to only their respective datasets D_1 and D_2. Let us define the *null hypothesis* which claims that the two distributions are the same.

⚠ The concept of a hypothesis here is different from what we saw in generalization/VC-theory. Hypothesis in decision theory simply means our hunch about a particular situation, e.g., $p = q$.

$$H_0 : p = q$$

The alternate hypothesis is
$$H_1 : p \neq q.$$

The goal of the so-called "two-sample test" is to decide whether H_0 is true or not. A typical two-sample test will construct a statistic (recall from Chapter 7 that a statistic is any function of the data)

$$\hat{t}$$

out of the two datasets, e.g., their individual means, their variances, and will use this statistic to *accept or reject* the null hypothesis, i.e., decide whether H_0 is true or false.

Let's say that we pick a threshold t_α, and the test statistic $\hat{t}$ is the difference of the means
$$\hat{t} = \left| \frac{1}{n} \sum_{x \in D_1} x - \frac{1}{n} \sum_{x \in D_2} x \right|.$$

Level of a test A statistician will then say that the null hypothesis is valid with *level* α if
$$\mathbb{P}_{D_1 \sim p, \; D_2 \sim p} \left(\hat{t} > t_\alpha\right) \leq \alpha. \tag{15.1}$$

In other words, if the null hypothesis were true (both D_1 and D_2 are drawn from the same distribution p) and if the probability of our empirical statistic $\hat{t}$ being larger than some *chosen* threshold t_α is smaller than some *chosen* probability α, then we know that the two distributions are the same despite only having finite data to check. The threshold α is called the p-value in the statistics literature and you will have seen statements like "gene marker XX is correlated with disease YY with p-value of 10^{-3}" or "smokers and non-smokers have different distributions of cancers with p-value of 10^{-3}".

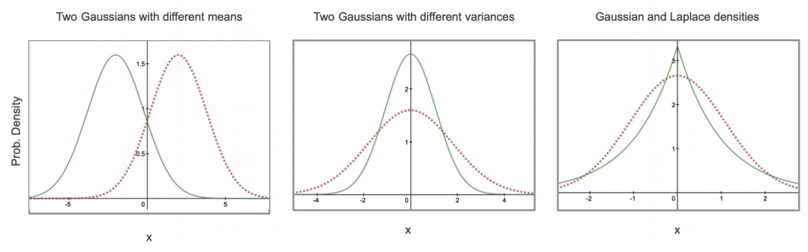
Power of a test The power of a two-sample test is the probability of rejecting the null hypothesis when it is actually false. We want tests with a large power, i.e., we like
$$\mathbb{P}_{D_1 \sim p, D_2 \sim q} \left(\hat{t} > t_\alpha\right) \tag{15.2}$$
being large if the two datasets D_1 and D_2 are drawn from two different distributions p and q respectively.

The key point to remember about two-sample tests is that they let us check if two distributions are the same without knowing anything about the distributions. We only need access to the samples and can run this test. This is fundamentally different than say

$$\mathrm{KL}(q \,||\, p) = \int q(x) \log \frac{q(x)}{p(x)} \,\mathrm{d}x$$

where we need to know the probabilities $q(x), p(x)$ to compute the distance between distributions.

Example 15.1. A two-sample test requires three things, a statistic $\hat{t}$, a level α and a threshold for the statistic t_α. The latter two are numbers that a statistician can pick, e.g., picking $\alpha = 0.05$ is an accepted standard in most biological studies.



15.2 Building the Discriminator in a GAN

Finding two-sample test statistics for arbitrary distributions is difficult, especially for high-dimensional problems where the samples are natural images. The key idea behind a Generator Adversarial Network (GAN) is to learn the statistic $\hat{t}$.

A good statistic is the one that lets us distinguish between data that comes from Nature's distribution and data that is synthesized by our generative model. This statistic, which is called the discriminator in GAN, is a critic of the generative model's results. It has a *high power* in (15.2) if the generated samples are different from those of Nature. Why? Because in this case for most thresholds t_α that we can pick, the power of the two-sample test in (15.2) will be large.

The discriminator should also be sound, i.e., if the two distributions are indeed the same (e.g., if our generator is as good as good as Nature's renderer), the discriminator should have a *low level* α in (15.1).

We are going to train a binary classifier

$$d_u : \mathcal{X} \mapsto [0, 1]$$

that will act as the discriminator in a GAN. You should think of the decision boundary of this binary classifier as the difference of the test statistic and our threshold $\hat{t} - t_\alpha$.

We next create a dataset to train this classifier. Given n images from Nature's distribution $p(x)$ and the distribution of our generator's images $q(x)$, we will label the former with $y = 1$ and the latter with $y = 0$ to create a joint dataset:

$$\begin{aligned} D_1 &= \left\{(x^i, 1)_{i=1,\dots,n} : x^i \sim p(x)\right\} \\ D_2 &= \left\{(x^i, 0)_{i=1,\dots,n} : x^i \sim q(x)\right\} \\ D &= D_1 \cup D_2. \end{aligned}$$

Fitting d_u on this problem can be done simply using the logistic loss wherein d_u is modeling the log-odds

$$\log \frac{\mathbb{P}(y = 1|x)}{\mathbb{P}(y = 0|x)} = d_u(x).$$

The logistic loss is therefore

$$u^* = \underset{u}{\text{argmin}} -\frac{1}{n} \sum_{x \sim D_1} \log d_u(x) - \frac{1}{n} \sum_{x \sim D_2} \log(1 - d_u(x)). \tag{15.3}$$

Observe that this is the same logistic loss that we are used to; the only difference being that the entire dataset has $2n$ samples with all the ones in D_1 having labels $y = 1$ and all the ones in D_2 having labels $y = 0$.

⚠ Notice how rigorous theory is used as an inspiration for developing GANs. This is a common theme that you will see in the deep learning literature; the models may seem *ad hoc* and sprung out of sheer intuition, but the reason they work well is often because there are sound theoretical principles behind them. Building this skill requires studying the classical curriculum (ML, statistics, optimization) but being creative in applying this curriculum with deep networks.

What is the ideal discriminator? The population risk corresponding to the discriminator's objective in (15.3) is

$$d^* = \underset{d}{\text{argmax}} \underset{x \sim p}{\mathbb{E}} [\log d(x)] + \underset{x \sim q}{\mathbb{E}} [\log(1 - d(x))] . \tag{15.4}$$

We can take the variational derivative of this objective (just like you did in HW 3 to compute the optimal classifier in the bias-variance tradeoff) to get

$$d^*(x) = \frac{p(x)}{p(x) + q(x)}. \tag{15.5}$$

⚠ For a functional

$$L[d] = \int \log d(x) p(x) \, \mathrm{d}x$$

the variational derivative is

$$\frac{\delta L}{\delta d}(x) = \frac{p(x)}{d(x)}.$$

Similarly, the variational derivative for

$$L[d] = \int \log(1 - d(x)) q(x) \, \mathrm{d}x$$

is

$$\frac{\delta L}{\delta d}(x) = \frac{q(x)}{1 - d(x)}.$$

Observe that the ideal discriminator is $1/2$ if the two distributions p and q are the same. The intuitive reason for this is that if the data D were really coming from the same distribution, we would never be able to fit a logistic classifier to get better than 50% error because D_1 and D_2 have different labels in spite of having similar input data.

Think of you would use our discriminator to build a two-sample test for a given dataset. If given two datasets D_1 and D_2 labeled as above

$$\hat{t} := \frac{1}{n} \sum_{x \in D_1} \mathbf{1}_{\{d_u(x) > 0\}} + \frac{1}{n} \sum_{x \in D_2} \mathbf{1}_{\{d_u(x) < 0\}}$$

and the threshold $t_\alpha = 1/2$. This construction is an example of what is called a "classifier-based two-sample test"; you can read more about it at Lopez-Paz and Oquab (2016).

It can be shown that if the two distributions are not the same, the

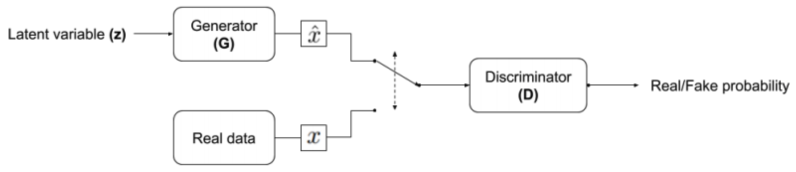


Figure 15.1: Schematic of the architecture in a GAN

power of the two-sample test is an increasing function of the statistic $\hat{t}$. Therefore if we wanted to maximize the power, maximizing the test statistic $\hat{t}$ of the discriminator is a good idea. This makes the discriminator more and more sensitive to the differences between samples from p and q.

15.3 Building the Generator of a GAN

The second key idea in a GAN is that the generator

$$g_v : \mathcal{Z} \to \mathcal{X}$$

that maps the latent space $\mathcal{Z} \subset \mathbb{R}^m$ to data space $\mathcal{X}$ is trained to *minimize* the power of the two-sample test.

The generator g_v wants to synthesize data that look like they came from Nature's distribution $p(x)$. As the generator's distribution q comes closer to p, the accuracy of the discriminator d_u will degrade (it cannot distinguish between them as easily) and thereby discriminator will be forced to make its test statistic more sensitive to subtle differences between the two distributions.

15.4 Putting the discriminator and generator together

The GAN objective combines two objectives: the discriminator updates its weights u to maximize the power and the generator updates its weights v to minimize the power. We will write the population version of the optimization problem as follows.

$$\min_v \max_u \; E_{x\sim p(x)} \; [\log d_u(x)] + E_{x\sim q(x)} \; [\log(1 - d_u(x))] \tag{15.6}$$

Let us fill in a few more details. The dataset of real images consists of samples from Nature's distribution $p(x)$, so we will write it as a finite sum over our dataset $D = \{x^i \sim p\}_{i=1}^n$. The generator uses samples z from some generic distribution, e.g., a standard Gaussian distribution.

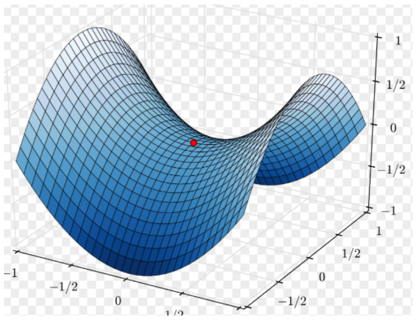
$$\min_v \max_u \frac{1}{n} \sum_{x\in D} \; [\log d_u(x)] + E_{z\sim N(0,I)} \, [\log(1 - d_u(g_v(z)))] \, . \tag{15.7}$$

Training a GAN The objective in (15.7) is an example of a min-max optimization problem. Such problems are quite difficult to solve and this is why training GANs is quite difficult. In practice, we typically resort to a few crude tricks. We sample a mini-batch of real images $\{x^1, \ldots, x^b\}$ and another mini-batch of noise vectors $\{z^1, \ldots, z^b\}$. Using these two mini-batches

1. we update the generator g_v using the gradient of the objective with respect to v.
2. update the discriminator d_u using the gradient of the loss with respect to u.

There is no need for the Reparametrization Trick here because there is no expectation being taken over parametrized distributions. This is a big benefit of the GAN formulation as compared to variational inference; the former does not have to be careful while picking a variational family and complex deep networks can be used as the generator or the discriminator easily. Let us next make a few comments about the objective in (15.7).

Solving min-max problems is difficult This is a min-max problem: the generator is minimizing the objective and the discriminator is maximizing the objective. Such problems are hard to solve in optimization especially with gradient descent techniques. Consider an example of a saddle point



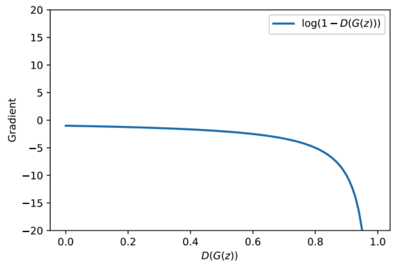
where the loss function increases in one direction and decreases in the other direction. Finding the solution of the min-max objective involves finding the saddle point. It is easy to appreciate that depending on how many steps of gradient descent we take for either of the min/max players we risk falling down or climbing up the hill. There are many many other other factors that make solving such problems hard, e.g., learning rate, momentum, stochastic gradients if we are using mini-batches. Hyper-parameters are very tricky to pick while training GANs and this is often called "instability of training".

A harsh discriminator inhibits the training of the generator The generator has a much more difficult task than the discriminator. During early stages of training, the generator needs to learn how to synthesize images whereas the discriminator can easily distinguish between bad images generated by the generator and good ones from our original dataset using very similar test statistics, e.g., an average of the RGB values all the pixels.

The gradient of the second term in the objective is

$$\nabla_v \log(1 - d_u(g_v(z))) = -\frac{\nabla_v \, d_u(g_v(z))}{1 - d_u(g_v(z))}.$$

As a function of $d_u(g_v(z))$ the second term in the objective thus looks like

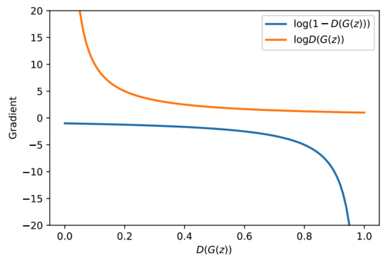


In other words, the gradient with respect to the generator's weights v is essentially zero if the generator is not working well (this is the case when $d_u(g_v(z))$ predicts a large negative value). This does not allow the generator to learn well; it is essentially like your advisor shooting down all your ideas.

Most GAN implementations therefore modify the second term in the objective to be

$$-\mathop{\mathbb{E}}_{z\sim N(0,I)}[\log d_u(g_v(z))]$$

which does not suffer from the small gradient problem.



Synthesizing new images from a GAN The generator samples latent factors $z \sim N(0, I)$ at test time to synthesize new images. The discriminator is not used at test time.

15.5 How to perform validation for a GAN?

For variational generative models, we can use the log-likelihood of synthesized images to obtain some understanding of whether the model is working well. If the log-likelihood of new images is similar to the log-likelihood of images in the training data then the new images are good at least as far as the model is concerned even if they may have perceptual artifacts.

Doing so is not so easy for implicit models because they do not output the likelihood of the generated samples. Run the generator a few times to eyeball the quality of images it generates.

But this is a very heuristic and qualitative metric.

Frechet Inception Distance (FID) A number of other metrics exist for evaluating generative models. One popular one is the so-called Frechet Inception Distance (FID) where we take any pre-trained model for classification, in this case people typically use the Inception architecture, and evaluate

$$\text{FID}(p, q) = \|\mu_p - \mu_q\|_2^2 + \text{trace}\left(\Sigma_p + \Sigma_q - 2\left(\Sigma_p \Sigma_q\right)^{1/2}\right).$$

where μ_p, Σ_p are the mean and covariace of the features of an Inception network when real images are fed to it and similarly μ_q, Σ_q are the mean/-covariance of the features when GAN-generated images are fed to the same network.

The above formula is the Wasserstein distance between the two densities p, q, There are many similar techniques such as the Maximum Mean Discrepency (MMD) that can be used to understand the discrepancy between the two distributions once the features are computed using some pre-trained model on their respective images.

Roughly speaking, the evaluation methodology in generative models, especially for images, is quite flawed. This is not a new phenomenon in machine learning/statistics because it is a intrinsically difficult problem to measure when two distributions are the same given only finite data from them. The problem is exacerbated in deep generative models because deep networks are very good at over-fitting, e.g., GANs can often end up memorizing the training data, they can generate very realistic images that are essentially the same as those in the training data. Nevertheless, a lot of techniques exist to make GANs synthesize high-quality images. See some examples at Brock et al. (2018); Karras et al. (2017).

The key behind the empirical success of GANs is to convert a problem about estimating distributions, sampling from them etc. into a classification problem. Deep networks are extremely good at classification as compared to other problems like regression, reconstruction etc. and GANs leverage this remarkably. This is a trick that you will do well to remember when you use deep networks in the future: typically you will always get better results if you manage to rewrite your problem as a classification problem. I suspect the real reason for this is that we do not have good regularization techniques for deep networks for non-classification problems.

15.6 The zoo of GANs

Due to the numerous issues with GANs, there have been a large number of variants and attempts to improve their empirical performance. They fall mainly into the following categories.

1. Optimization tricks to train the generator-discriminator pair in a more stable fashion.
2. New loss functions that change the binary cross-entropy loss of the discriminator to something else. A majority of papers, including the example we saw above, fall into this category.
3. Characterizing the kind of critical points, equilibria of the training process; this is a similar line of analysis as the study of the energy landscape of deep networks for standard supervised learning.
4. Connections with variational inference suggest that GANs and their training techniques are essentially variational inference in disguise (Nowozin et al., 2016).
5. Coming up with new ways of evaluating generative models.

In addition to the above lines, there are many other novel and interesting applications such as Cycle-GANs and conditional-GANs.

Bibliography

Aizerman, M. A. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837.

Alemi, A., Poole, B., Fischer, I., Dillon, J., Saurous, R. A., and Murphy, K. (2018). Fixing a broken elbo. In *International Conference on Machine Learning*, pages 159–168. PMLR.

Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58.

Bartlett, P. L., Harvey, N., Liaw, C., and Mehrabian, A. (2019). Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *J. Mach. Learn. Res.*, 20:63–1.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877.

Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer.

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311.

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Brock, A., Donahue, J., and Simonyan, K. (2018). Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*.

Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J., Sagun, L., and Zecchina, R. (2016). Entropy-sgd: Biasing gradient descent into wide valleys. *arXiv:1611.01838*.

Chaudhari, P. and Soatto, S. (2017). Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks. *arXiv preprint arXiv:1710.11029*.

Cortes, C. and Vapnik, V. (1995). Support vector machine. *Machine learning*, 20(3):273–297.

Efron, B. (1992). Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

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.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Kawaguchi, K. (2016). Deep learning without poor local minima. In *Advances in neural information processing systems*, pages 586–594.

Kidambi, R., Netrapalli, P., Jain, P., and Kakade, S. (2018). On the insufficiency of existing momentum schemes for stochastic optimization. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

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.

Kushner, H. and Yin, G. G. (2003). *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media.

Le Roux, N., Schmidt, M. W., Bach, F. R., et al. (2012). A stochastic gradient method with an exponential convergence rate for finite training sets. In *NIPS*, pages 2672–2680.

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.

Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. (2018). Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399.

Liu, C. and Belkin, M. (2018). Mass: an accelerated stochastic method for over-parametrized learning. *arXiv preprint arXiv:1810.13395*.

Lopez-Paz, D. and Oquab, M. (2016). Revisiting classifier two-sample tests. *arXiv preprint arXiv:1610.06545*.

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.

Mescheder, L., Nowozin, S., and Geiger, A. (2017). The numerics of gans. In *Advances in Neural Information Processing Systems*, pages 1825–1835.

Minsky, M. and Papert, S. A. (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

Nowozin, S., Cseke, B., and Tomioka, R. (2016). f-gan: Training generative neural samplers using variational divergence minimization. In *Advances in neural information processing systems*, pages 271–279.

Pedro, D. (2000). A unified bias-variance decomposition and its applications. In *17th International Conference on Machine Learning*, pages 231–238.

Pickering, A. (2010). *The cybernetic brain: Sketches of another future*. University of Chicago Press.

Polyak, B. T. (1990). A new method of stochastic approximation type. *Avtomatika i telemekhanika*, (7):98–107.

Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855.

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.

Razavi, A., van den Oord, A., and Vinyals, O. (2019). Generating diverse high-fidelity images with vq-vae-2. In *Advances in Neural Information Processing Systems*, pages 14866–14876.

Recht, B. and Ré, C. (2012). Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. *arXiv preprint arXiv:1202.4184*.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Ruppert, D. (1988). Efficient estimations from a slowly convergent robbins-monro process. Technical report, Cornell University Operations Research and Industrial Engineering.

Salakhutdinov, R. and Larochelle, H. (2010). Efficient learning of deep boltzmann machines. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 693–700.

Scholkopf, B. and Smola, A. J. (2018). *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Adaptive Computation and Machine Learning series.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv:1412.6806*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer.

Vapnik, V. (2013). *The nature of statistical learning theory*. Springer science & business media.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Wiener, N. (1965). *Cybernetics or Control and Communication in the Animal and the Machine*, volume 25. MIT press.