ESE 546

Principles of Deep Learning

Fall 2019

Instructor
Pratik Chaudhari pratikac@seas.upenn.edu

Teaching Assistants
Evangelos Chatzipantazis vaghat@seas
Yansong Gao gaoyans@sas
Kushagra Goel kgoel96@seas
Ashish Mehta ashishme@seas
Dewang Sultania dewang@seas

January 3, 2020

Contents

1 Logistics and Introduction 4

2 The perceptron algorithm and kernels 21

3 Back-propagation 34

4 Convolutional neural networks, Neural architectures 43

5 Neural architectures, Regularization 55

6 Early stopping, Data Augmentation 67

7 Dropout, Batch-Normalization 77

8 Gradient descent, stochastic gradient descent and accelerating techniques 91

9 Gradient descent (cont.), Nesterov's acceleration and Lyapunov functions 98

10 Nesterov's acceleration, Lyapunov functions and gradient flows 107

11 ODE for Nesterov's acceleration, stochastic gradient descent 114

12 Stochastic gradient descent, Markov chains 121

13 Acceleration of SGD, Markov chains 130

14 Markov chains, Gibbs distribution 139

15 Gibbs distribution **146**

16 Linear neural networks, stable manifold theorem, linear residual networks **156**

17 Linear neural networks, stable manifold theorem, linear residual networks **160**

18 Stable manifold theorem, linear residual networks, shape of local minima **169**

19 Binary Perceptron and Entropy-SGD **179**

20 Langevin dynamics, Markov Chain Monte Carlo **185**

21 Wrap up of Markov Chain Monte Carlo, Variational Inference **186**

22 Variational Inference, Auto-Encoders **190**

23 Auto-Encoders, Information Bottleneck **196**

24 Weight uncertainty in neural networks **198**

25 Weight uncertainty in neural networks, PAC-Bayes generalization bound **205**

26 Generative Adversarial Networks (GANs) **213**

Lecture 1

Logistics and Introduction

Reading

- Bishop 1.1-1.5
- Goodfellow Chapter 1
- "A logical calculus of the ideas immanent in nervous activity" by Warren McCulloch and Walter Pitts (McCulloch and Pitts, 1943).
- "Computing machinery and intelligence" by Alan Turing in 1950 (Turing, 2009).

Welcome to ESE 546: "Principles of Deep Learning". Deep networks are at the heart of modern algorithms for computer vision, natural language processing and robotics. Design of these networks requires a combination of intuition, theoretical foundation and empirical experience; this course discusses general principles of deep learning that cut across these three. It develops insight into popular empirical practices with a focus on the training of deep networks, builds the theoretical skills to develop new ideas in deep learning and to deploy deep networks in real world applications.

1.1 Pre-requisites

Required

- Proficiency in programming: ENGR105, CIS110, CIS120, or equivalent. Assignments in this course are based in Python but if you have used some other high-level language like MATLAB before, you should be able to pick up Python easily.

- Probability: ESE301, STAT430, CIS261, ENM503, or ESE530 or equivalent
- Linear Algebra: Math 312, EAS 205 or equivalent

Recommended

- Machine Learning or Data Analytics: ESE 305, ESE 402, ESE 542, ESE 545, CIS 519, or CIS 520.
- Optimization: ESE 204, ESE 504, or ESE 605.

Undergraduates: Permission of the instructor is required to enroll in this class. If you are registered, you asked for permission and were granted it at some time in the past. If you are unsure whether your background is sufficient for this class, talk to/email the instructor.

1.2 Material

The book "Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1). Cambridge: MIT press" (available online at https://www.deeplearningbook.org) will be used as a reference and reading material. This will be mentioned at various places in the notes as "Goodfellow xx".

The book "Bishop, C.M., 2006. Pattern recognition and machine learning. Springer" will be another reference. This will be mentioned in the notes as "Bishop xx"

Detailed instructor notes will be provided and they will be the primary text for this course. Suggested reading material from the above textbooks and scientific literature will be provided for each class.

"Dive into Deep Learning" (http://www.d2l.ai) is a good reference for the applied parts of the class. If you want to brush up your Python/Numpy in a way that is useful for the class, this is a good place to start. The recitation session on Fri 8/30 will go through these basics.

1.3 Administration

Units: 1 (3 hours of instruction, 1 hour of recitation and 6 hours of homework per week)

Location: Lectures will be held in the ARCH 208 Auditorium (https://goo.gl/maps/EuKyG6MRy3oYkK9Z6) on Monday and Wednesday from 1.30p-3p. Recitation sessions will be held in Towne 100 (https://goo.gl/maps/CwhuedemPYs47VMZ7) on Fridays from 11a-12noon.

Canvas: https://canvas.upenn.edu/courses/1474815. This is the main webpage for the class. All course material will be disseminated from here.

Piazza will be used for discussions and clarifications. Sign up for it at https://piazza.com/upenn/fall2019/ese546.

Email address:

Pratik Chaudhari pratikac@seas
Evangelos Chatzipantazis vaghat@seas
Yansong Gao gaoyans@sas
Kushagra Goel kgoel96@seas
Ashish Mehta ashishme@seas
Dewang Sultania dewang@seas

Instructor office hours Tuesday-Wednesday 4-5p in Levine Hall 470 (https://goo.gl/maps/GmuSvo2VtaS3hYB66).

No instructor office hours on Tuesday Sept. 3, 2019.

TA office hours TBD, will be announced on Canvas.

1.4 Grading policy

40% 4 problem sets, each contributing to 10% of the total grade

20% Midterm (closed book)

20% Project

20% Final Exam (closed book)

Late policy Each student will have 5 "late days" to use during the semester. You can use these late days to submit problem sets/project after the due date without any penalty. Assignments that are submitted late, after exhausting the quota of late days will result in 50% credit deducted per day, i.e., zero credit after 2 late days. Do not exhaust all the late days on the first problem set.

Assignments will be submitted through Gradescope.

DO NOT turn in Problem Set 0. It is only provided for you to get a feel for what the assignments in this course will look like.

1.5 Project

This accounts for 20% of your grade. Form teams of 4 students. The timeline for the project is as follows.

10/30 Project proposals are due. The proposal will consist of the title, team members and an abstract. The abstract can be at most 1000 words.

11/6 Feedback and approval from the instructors.

12/4 Project report and source code due. This can be at most 4 pages in NeurIPS 2019 Latex format, excluding references.

The instructor will summarize all the projects in the "Final Remarks" class on 12/9. Three teams will be invited to give a 10-minute talk each on their project on 12/9.

Some pointers in picking project topics: This class focuses on fundamentals of deep learning. Do not pick a project where you will spend significant amount of time collecting/curating data, this is not aligned with the objectives of the course. You *can* pick a project that is of the form "I had data from XX in my lab, I am training a deep network on this data to do YY". You can also do a project of the form "I really like paper XX, I re-implemented it". However note that the latter kind of projects will be judged carefully; in particular it is not okay to significantly exploit existing implementations on the Internet and submit that as a part of your project. You can pick a theoretical project which involves reading and understanding a few related papers; it is however advised that there be an implementation component in addition to the literature review.

1.6 Computation

All problem sets and the project will involve a component of programming. For instance, you may be asked to write code for training a neural network on a given dataset and submit plots/results given by your code. You can use the following resources to do these assignments.

- Your personal computer if you have.
- Google Colaboratory (https://colab.research.google.com) gives you access to one GPU/TPU for 5 hours at a time. This should be sufficient for doing homeworks. The first recitation session will provide starter code for you to use Google Colaboratory effectively.
- Each registered student for this class will get $50-100 worth AWS Educate credits after the drop date (Sept 10). The TAs will tell you how to use "spot" instances. Please

manage these credits judiciously, if you run out of them we will not be able to provide you more and this will affect your coursework. You may want to preserve these credits for the project. Lastly, AWS credits is not an incentive for sticking around in the class if you do not intend to take it.

- New Google Cloud (GCP) accounts get $300 worth starter credits. If you have not exhausted these already, it is a great resource. The TAs will not be able to provide support for Google Cloud but GCP is very similar to the Amazon Cloud (AWS). The recitation on Fri 9/6 will cover the basics of using AWS.

1.7 Academic Integrity

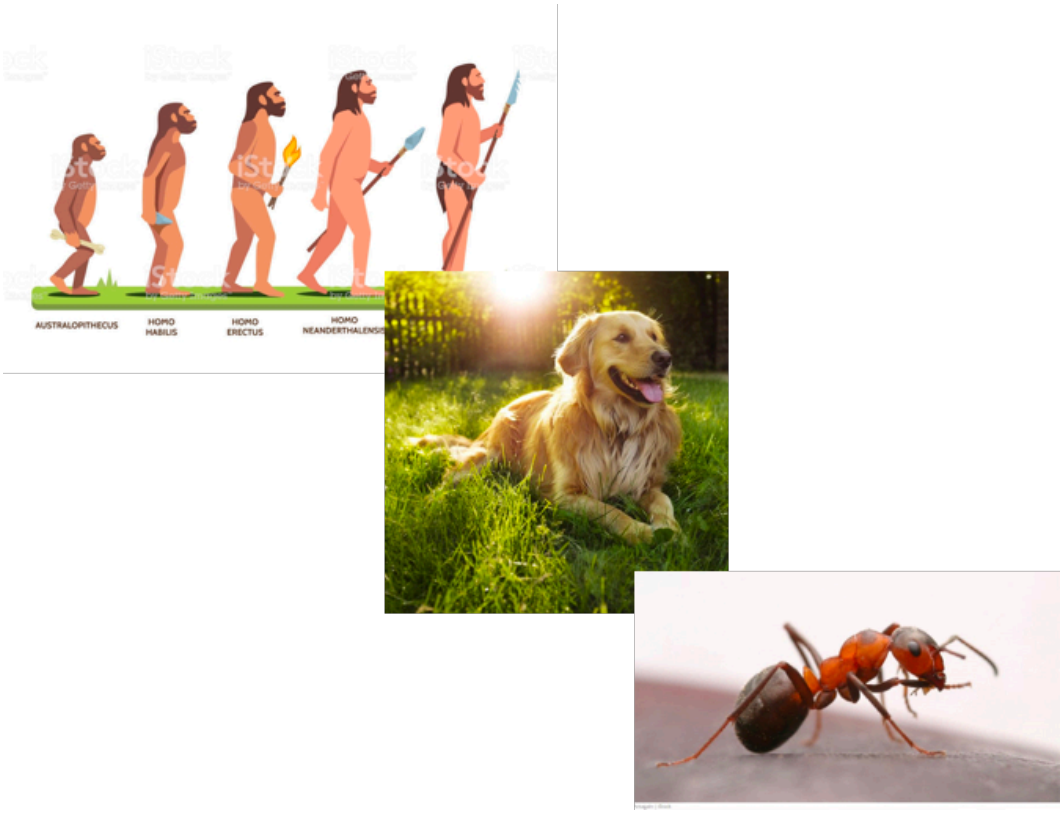
You are encouraged to collaborate with your peers for solving problem sets, read books and other instructional materials both online and off-line to help you understand the concepts taught in this class. While doing so, you might come across code or pseudo code for a problem set/project. When you begin to write your submission (problem set/code) you should keep aside all these materials (including your friends) and do things from "from scratch". In short, everything you write/code and submit should be your own work and done independently.

You should disclose all collaborations in your submission at the top. If you came across some code as a part of your problem set/project you must mention it.

Collaboration is different from cheating. The latter will have serious consequences. Cheating is defined as attempting, abetting or using unauthorized assistance (knowledgeable senior who is not taking the class) or material (e.g., online code). Some examples of cheating are: copying problems sets/exams, handing in someone else's work as your own and plagiarism. This will not be tolerated and will be reported to the university.

1.8 What is intelligence?

What is intelligence? It is hard to define, I don't know a good definition. But we know it when we see it. All humans are intelligent, you are intelligent. Dogs are plenty intelligent. A house fly or an ant is perhaps less intelligent than a dog.



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.

Here is a fun plant however. Tunicates are invertebrates. When they are young they roam aroud the ocean floor in search of nutrients, and they indeed have a nervous system (ganglion cells) at this point of time that helps them do so. Once they find a nice 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 after they attach to the rock, they develop a thick covering (shown above) or a "tunic" to protect themselves.

You need mobility to be called intelligent. With this comes the ability to affect your environment, pre-empt antagonistic agents in the environment and take actions that achieve your desired outcomes. We can now write down the three key components an intelligent, autonomous agent possesses.

$$\text{Perception} \Rightarrow \text{Cognition} \Rightarrow \text{Action}$$

Perception refers to the sensory mechanisms to gain information about the environment, control. Action refers to your hands, legs, motors, engines 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 the actions.

The flowchart above is not a mere feed-forward process. Your sensory inputs depend on the previous action you took. Time is an integral component.

You should not think of learning 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.

Remark 1. If you agree with the above definition of intelligence, would you say AlphaGo demonstrates intelligence?

This class will focus on "Learning". It is a component, not the entirety, of cognition. Examples of other classes that address various aspects of this "loop of intelligence" are:

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

Remark 2 (Why is learning essential to cognition?)**.** In principle, a supreme agent which is infinitely fast and clever can interpret its sensory data and compute optimal actions for any task it wishes. One would think learning is not essential to cognition, certainly not for this supreme agent. However, an autonomous agent does require learning. Learning is essential for the following reasons:

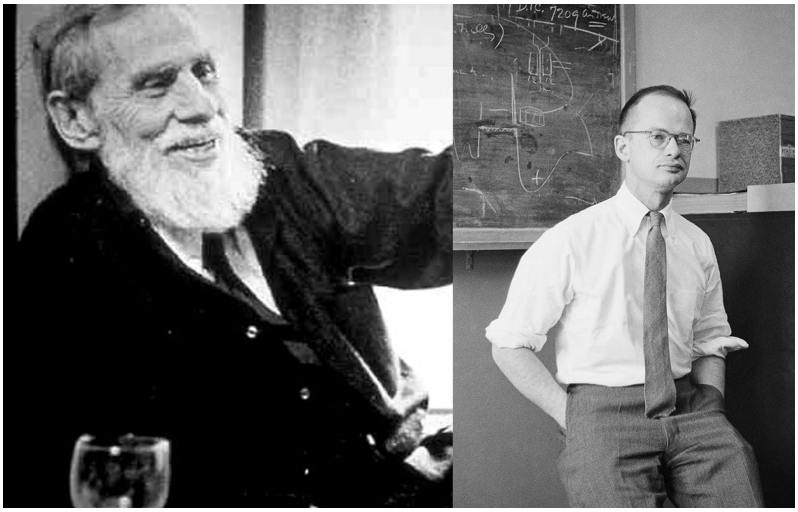
- if you are not as fast as the supreme agent or if you want to save some compute time/energy during decision making.
- the big problem with the supreme agent is that it does not have any memory. It tries to make up for it by being very good at computation. This does not work if the future data is slightly different than the model of the environment it has been creating causally. Priors, i.e., models that were learnt on past data, may help make up for the gap in such cases and help predict more accurately.

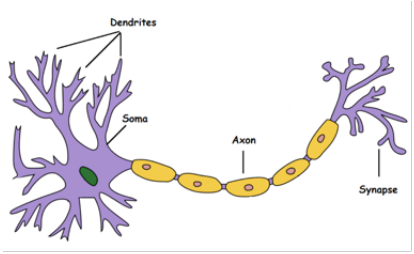
The second bullet above is the key reason why learning is essential. You should not think of a deep network or a machine learning model as a mechanism that directly undertakes the actions. It is better suited to provide a prior on the possible actions that an autonomous agent should take; other algorithms 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.

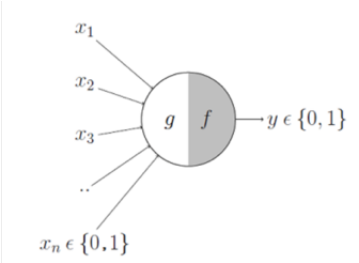
1.9 Intelligence: The Beginning (1942-50)

With that background, I want to give you a short glimpse into how these ideas have developed, roughly over the past 75 years.

The story roughly begins in 1942. 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.

This action was happening in Chicago. 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 (Turing, 1937).

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.*

This paper (Turing, 2009) is the second assigned reading for this lecture. 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.

Back to the Cambridge in the US, Norbert Wiener in about 1942 had created a little club of enthusiasts. They would coin the term "Cybernetics" which is exactly what one would call "Artificial Intelligence" today. You can read more in the original book (Wiener, 1965) whose table of contents is shown below.

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.

CONTENTS

Preface to the Second Edition vii

PART I

ORIGINAL EDITION

1948

Introduction 1

I Newtonian and Bergsonian Time 30

II Groups and Statistical Mechanics 45

III Time Series, Information, and Communication 60

IV Feedback and Oscillation 95

V Computing Machines and the Nervous System 116

VI Gestalt and Universals 133

VII Cybernetics and Psychopathology 144

VIII Information, Language, and Society 155

PART II

SUPPLEMENTARY CHAPTERS

1961

IX On Learning and Self-Reproducing Machines 169

X Brain Waves and Self-Organizing Systems 181

Index 205

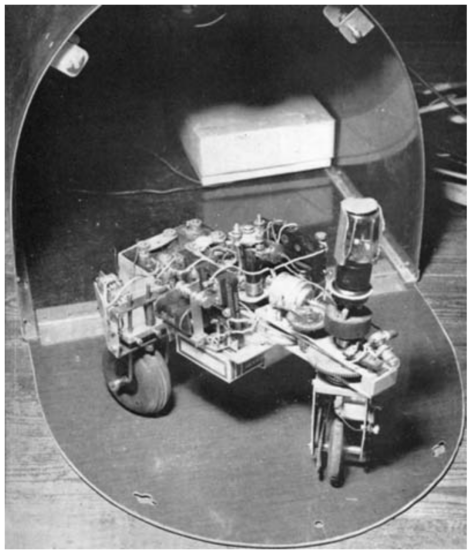
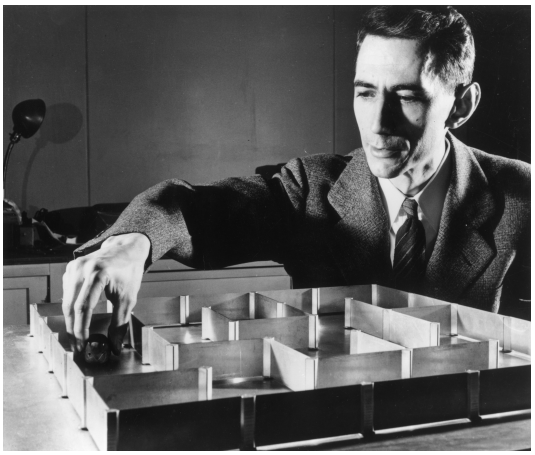


Figure 3.3. Anatomy of a tortoise. Source: de Latil 1956, facing p. 50.

You can also look at the book "The Cybernetic Brain" (Pickering, 2010) to learn more.

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 concepts. 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.

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. We will take a look at all these aspects in this class.

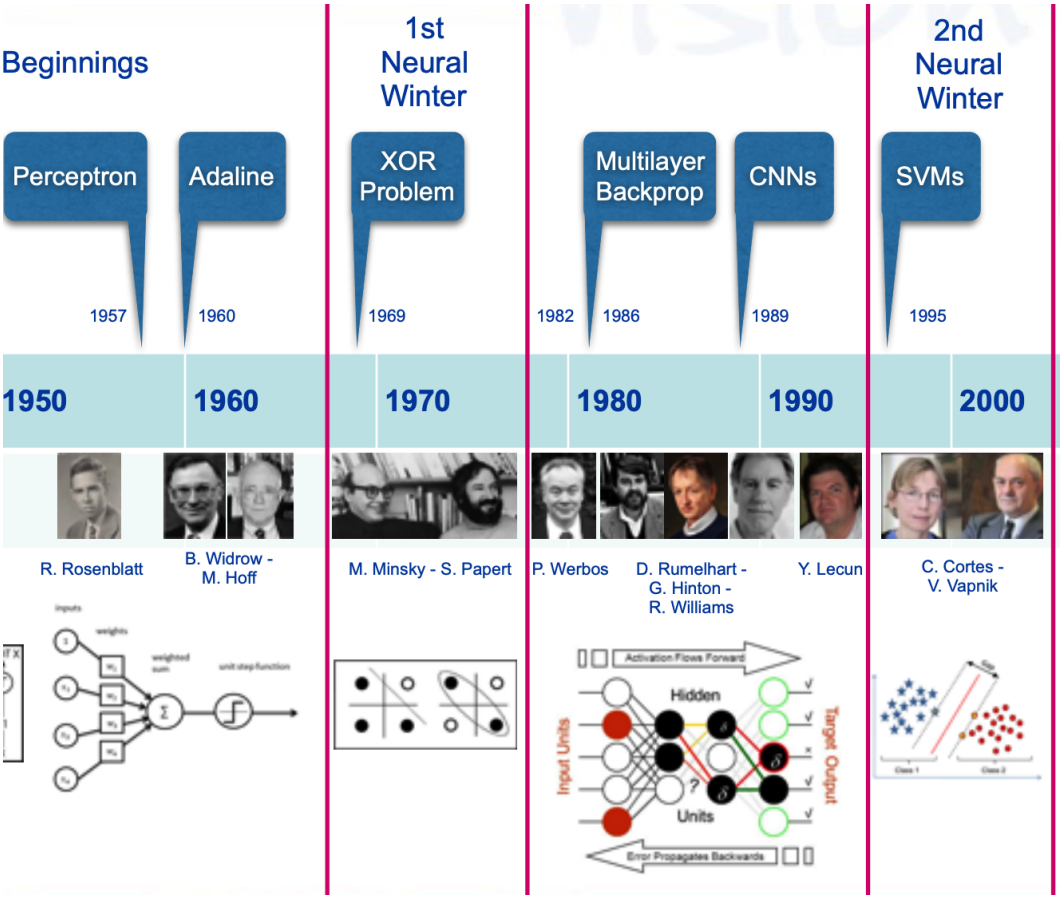
1.10 Intelligence: Reloaded (1960-2000)

The early period created interest in intelligence and devised some basic ideas. The first major progress of what I would call the second era was made by Frank Rosenblatt in 1957. Rosenblatt's perceptron is a model with a single binary neuron. The inputs 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 function

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

Rosenblatt perceptron has a single neuron, it cannot distinguish between complex data. This is what Marvin Minsky and Seymour Papert discussed in Minsky and Papert (2017). This

book was widely perceived as a death knell for the perceptron which coupled with the rise of “symbolic reasoning” (as opposed to the connectionist approach we’ve discussed here) resulted in what one would call the first AI winter; research on neural networks slowed down. You can read about it here (Rosenblatt, 1958).



1.11 Intelligence: Revolutions (2006-)

Lecture 2

The perceptron algorithm and kernels

Reading

- Bishop Chapter 3
- Goodfellow Chapter 5.1-5.4
- Random Features for Large-Scale Kernel Machines (Rahimi and Recht, 2008).

A few more reference texts that I did not mention last time:

- Machine Learning: A Probabilistic Perspective, by Kevin P. Murphy
- Trevor Hastie, Robert Tibshirani, Jerome Friedman (2009) The Elements of Statistical Learning
- David MacKay (2003) Information Theory, Inference, and Learning Algorithms

These books are intended as advanced reference material. We will not use them in the course but you are encouraged to peruse through them to improve your understanding of the material.

2.1 Admin

Problem Set 0 has been uploaded to Canvas. Try to solve the problems, they are a combination of analytical calculations and coding assignments. The problem set covers basic probability,

optimization and SVMs, you will have seen similar problems in your previous courses in these areas. This problem set is roughly representative of the length and hardness of the problem sets in this course. Do NOT turn in problem set 0; the instructors will upload the solutions to Canvas in a few days and you can verify them yourself.

2.2 Setup

Nature gives us data. On this data, it provides us with a prediction problem:

$$\underbrace{\text{input data}}_{X} \rightarrow \underbrace{\text{predictions}}_{Y}.$$

Nature often does not 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: 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 simply 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 $\mathcal{P}$ over (X, Y). It draws n iid samples from this distribution

$$D_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{n}$$

where

$$(x_i, y_i) \sim \mathcal{P} \quad \forall i \leq n$$

and hands D_{train} to us as the "training set". If we wish to test our predictor on new data then nature gives us a "test set"

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

The assumption that the distribution $\mathcal{P}$ generates both D_{train} and D_{test} is very important. This distribution provides coherence between past and future samples: past samples that were used to train and future samples that we will wish to predict upon.

What is the task in machine learning? Imagine 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 of possible intensities 255^3. It might happen that we discover one particular pixel, say at location $(25, 45)$, which takes distinct values in all images inside D_{train}. Here is a predictor: given a test datum the predictor outputs the label of the first

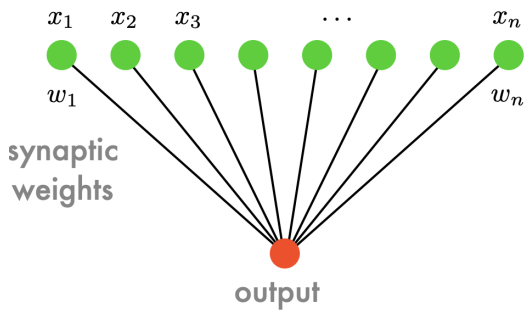
image in the training set whose pixel $(25, 45)$ matches that of the test datum. Observe that this predictor achieves zero error on D_{train}. Why do you think this will not work well for images outside D_{train}?

> 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}.

How to generalize? If we never see data from outside D_{train} why should we hope to do well on it? The key is the distribution $\mathcal{P}$. It is to constrain the class of predictors we entertain in our search. If this class is too big we risk finding rules similar to the one constructed above: they are good on the training data but cannot generalize. If this class is too small we never even predict on the training data well, surely the performance on the test data will be equally bad. Finding the right class of functions to fit the data is known as the model selection problem. Regularization in machine learning is a technique that will help constrain large model classes. In this sense, regularization in ML is different from regularization in, say, image processing.

2.3 Linear classifiers: Perceptron

First built by Frank Rosenblatt in 1957 at Cornell to identify pictures of triangles.



We will first consider a classification problem, i.e., $y_i \in \{-1, 1\}$. The perceptron is a linear classifier

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

Key points:

- we have a weighted linear combination of the inputs x
- combined using a nonlinear activation function sign
- an offset (bias)

What is the dimensionality of the vector w and the bias b? A perceptron creates a hyper-plane that separates the data, what direction does w specify in this hyper-plane? What is the intercept for the plane?

Let's see how we can train a perceptron. Training involves finding w and b that make the fewest mistakes on the training set. This is formally written down as a loss function

$$\ell(w, b) = \frac{1}{n} \sum_{i=1}^{n} \big(1 - \delta\left(y_i, f(x; w, b)\right)\big). \tag{2.1}$$

where $(x_i, y_i) \in D_{\text{train}}$. The delta function $\delta(y, y') = 1$ iff $y = y'$. This loss simply counts the number of mistakes made by the perceptron on the training set D_{train}.

2.3.1 Stochastic gradient descent

Let us consider a linear regression problem where $y_i \in \mathbb{R}$. The objective is to minimize the sum of the squares of the errors made by a linear predictor on the data. We can write down

this loss as

$$\ell_{\text{ls}}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(y_i - w^\top x_i - b \right)^2 . \tag{2.2}$$

Let us ignore the bias term for now, this can be easily fixed by appending a 1 to the data, i.e., feeding $x' = [x, 1]$ as the data. The last element of w becomes the bias in that case. The ordinary least squares loss function looks like

$$w^* = \operatorname*{argmin}_w \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(y_i - w^\top x_i \right)^2 .$$

We can solve for w directly, i.e., by equating the derivative of ℓ_{ls} to zero. The minimizer w^* of ℓ_{ls} is such that

$$\frac{\mathrm{d}\ell_{\text{ls}}}{\mathrm{d}w} = 0.$$

Is the converse true? Let's go through the steps once:

$$\frac{\mathrm{d}\ell_{\text{ls}}}{\mathrm{d}w} = -\frac{1}{n} \sum_{i=1}^{n} \underbrace{\left(y_i - w^\top x_i \right)}_{\text{residual}} x_i = 0.$$

How many equations are these? How many variables are we solving for? The key point for this linear regression is that we seek to find values for $d + 1$ variables and have exactly $d + 1$ equations at hand to get them. We are done at this point: we give this problem to Numpy.

Remark 3. When does the above solution not work? It does not work if two equations above are linearly dependent and we do not have enough equations to find all variables. What is the condition on the data x_i that characterizes this?

Remark 4. What is yet another problem with this? Does anyone know how does Numpy solve systems of linear equations?

Instead of giving the analytical derivative to Numpy, we can solve the problem ourselves. Let us take a simple algorithm:

(i) initialize w arbitrarily, say set it to zero.

(ii) run through the samples one by one and update w in the direction of the optimal w for that sample

$$\begin{aligned} w_{t+1} &= w_t - \eta \, \partial_w \, \ell_{\text{ls}} \\ &= w_t + \frac{\eta}{n} \big(y_t - w^\top x_t \big) \, x_t . \end{aligned} \tag{2.3}$$

(iii) start step (ii) from the first sample again at $t = n$.

This is the simplest instantiation of stochastic gradient descent (SGD). What do we know about this? The loss function for least squares is convex. The SGD algorithm converges for convex losses (more on this later). After a few iterations over the dataset D_{train}, we obtain an approximation for w^*.

2.3.2 Back to fitting the perceptron

The decision function of the perceptron is nonlinear; we cannot solve for the parameters using a linear solver. We can however use SGD.

What loss function should we use? Remember that the perceptron is trying to solve the classification problem, the labels are $y_i \in \{-1, 1\}$. Use the same trick as before to forget about the biases b. Consider the "hinge loss"

$$\begin{aligned} \ell_{\text{hinge}}(w; x, y) &= \max\left(0, -yw^\top x\right) \\ &= \begin{cases} > 0 & \text{for incorrect prediction,} \\ 0 & \text{for correct prediction.} \end{cases} \end{aligned} \tag{2.4}$$

How does the hinge loss look? Plot it as a function of $yw^\top x$. Let us try to compute the gradient of ℓ_{hinge}.

$$\partial_w \, \ell_{\text{hinge}}(w; x, y) = \begin{cases} -yx & \text{for incorrect prediction} \\ 0 & \text{else.} \end{cases} \tag{2.5}$$

The iterations of SGD for fitting the real-valued perceptron look like

$$w_{t+1} = \begin{cases} w_t + \eta \, y_t x_t & \text{if previous prediction was wrong} \\ w_t & \text{else.} \end{cases} \tag{2.6}$$

We will set

$$\eta = 1.$$

Is this the best choice? One keeps taking these updates the error on the training dataset goes to zero. In other words the parameters are changed only if we make a mistake. The updates tend to correct mistakes: suppose the perceptron w_t was incorrect on sample x_t. The updated

parameters satisfy

$$\begin{aligned} y_t w_{t+1}^\top x_t &= y_t \left(w_t + y_t x_t\right)^\top x_t \\ &= y_t w_t^\top x_t + y_t^2 x_t^\top x_t \\ &= y_t w_t^\top x_t + \|x_t\| \\ &\geq y_t w_t^\top x_t. \end{aligned}$$

The value of $y_t w_t^\top x_t$ always becomes more positive at each update on the sample. If we consider the same input repeatedly, it will be classified correctly eventually. Samples may fight with each other for capacity in the perceptron, i.e., other samples in (2.6) can cause the perceptron to become incorrect. This is the perceptron algorithm.

Remark 5 (What is the problem with the hinge loss as we have defined it above?)**.**

Remark 6 (When does the perceptron stop?)**.** The training algorithm stops after a finite number of mistakes if the original data is linearly separable, i.e., a solution to the perceptron exists. You have probably done this proof in an earlier course. In short, if there exists a w^* with unit length that makes zero mistakes on the training set

$$y_i {w^*}^\top x_i \geq \rho \quad \forall i \leq n$$

then the perceptron algorithm converges to a linear separator after a number of updates at most

$$\text{number of updates of perceptron} \leq \frac{r^2}{\rho^2}.$$

where $r > 0$ is such that $\|x_i\| \leq r$. There are a few key properties to remember from this formula above:

- $\frac{r^2}{\rho^2}$ 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.
- there are no constant factors, this is also the worst case number of updates; this will be rare.
- 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.

How do the parameters w for a perceptron look after training? Note that the parameters are a linear combination of the data.

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

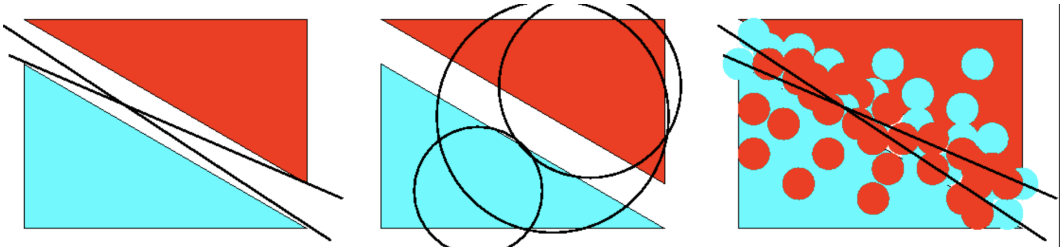
where the set $\alpha_i \in \{0, 1, \ldots\}$ indicates how many mistakes were made on that sample during training. The perceptron computes the function

$$f(w; x) = \left(\sum_{i=1}^{n} \alpha_i y_i x_i \right)^\top x$$
$$= \sum_{i=1}^{n} \alpha_i y_i \, x_i^\top x.$$

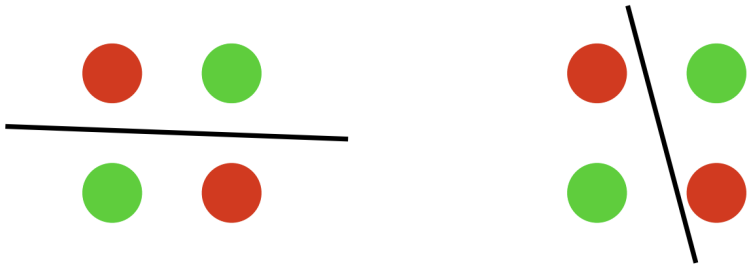
Remember the special form: the inner product of the sample x with all other samples in the training dataset x_i.

2.4 Kernels

The class of functions we search on matters. The perceptron works for data that can be linearly separated; it separates such data using hyper-planes.



Consider the third figure on the right: sometimes data may just be hard to separate cleanly into two classes. It may not be *separable*. Next consider the second figure: imagine if instead of hyper-planes you only had access to balls in $\mathbb{R}^d$, circles in $\mathbb{R}^2$. You would not be able to separate the same dataset using such a classifier. The reason is that you are not searching over the correct function class. This is often hard to distinguish in practice. Is your training error large because you are not searching over the correct function class or is your data simply too hard?



This dataset is not linearly separable. Is it non-linearly separable? This example in the book Minsky and Papert (2017) essentially killed the research on perceptrons in 1969. Here is a simple trick to classify such datasets: modify each input sample to be

$$x' = \begin{bmatrix} x_i & y_i \end{bmatrix}.$$

See that the XOR function on 4 data points drawn above is linearly separable in this new space.

Remark 7. Why can't you use the above trick of appending the labels to the data as a classifier?

Kernels extend this basic idea of projecting data into higher dimensions. They map data into some feature space of our choice

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

and fit the perceptron in this new space. The updates of the original perceptron algorithm are modified to

$$w_{t+1} = \begin{cases} w_t + y_t \phi(x_t) & \text{if mistake} \\ w_t & \text{else.} \end{cases} \tag{2.7}$$

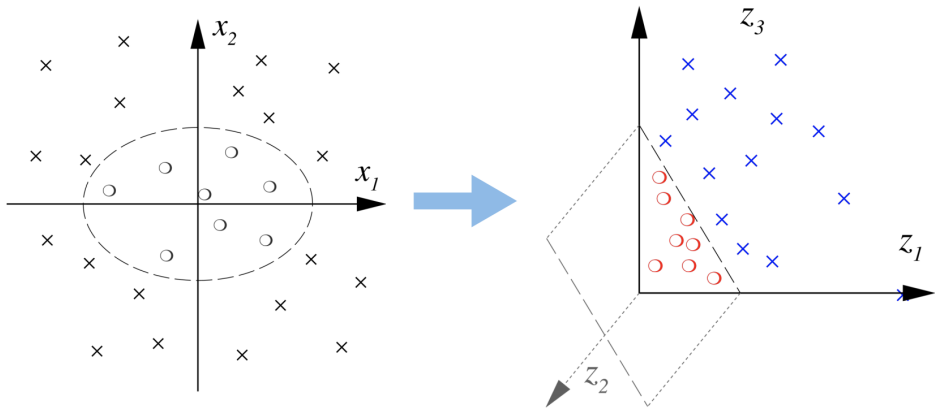
the parameters are given by

$$w^* = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)$$

and the function computed by the perceptron is

$$f(w; x) = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)^\top \phi(x);$$

it is now a linear combination of the features, not the data.

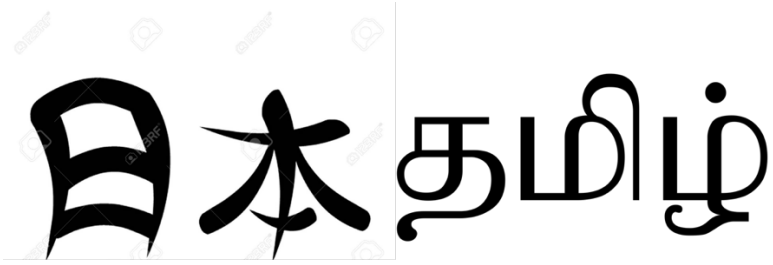


Consider this example, the original dataset is not linearly separable. We can however create, say, a quadratic feature space

$$\begin{bmatrix} x_1 & x_2 & x_1^2 + x_2^2 \end{bmatrix}$$

show on the right. This is easily linearly separable. What are the problems with kernels?

- Often we do not know what features to use. This is where machine learning researchers spent most of their effort until 2010 or so. If you are identifying different written scripts, e.g., Japanese Kanji vs Hindi, you need experts from to create features.



- Features are expensive to compute. Your homework zero contains an example where we compute the responses of Gabor filters. You need lots of filters and even for MNIST images, the dimensionality quickly becomes prohibitive.
- If you are at Google and wish to optimize your search prediction algorithm, you need to redo all optimizations you made for Japanese for classifying Hindi. Engineering systems built using hand-made features are hard to transfer.
- Curse of dimensionality: separating data becomes easier if you project it to higher dimensions. However there are many more candidate classifiers in the higher dimension. You need more data to find the best one. Best in the sense of what?

You can work around some of the above complaints by using lots of features from a class of features and hope that they capture relevant properties of the data. This is the "kitchen sink" approach to machine learning and *it often works great*. The key to deploying this in practice is to do the computations efficiently. A great example of this very popular idea is today's assigned reading (Rahimi and Recht, 2008).

Definition 8 (Kernel)**.** A kernel $k : X \times X \to \mathbb{R}$ is a symmetric function in its arguments for which the following property holds

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

for *some* feature map ϕ.

What other properties hold for a kernel? Given a kernel, is the feature map unique? Can you give an example?

If a kernel is much cheaper to compute than $\phi(x)$ you can kernelize your algorithm. Let us build the "kernel perceptron". Consider our original perceptron algorithm in the feature space.

$$w_{t+1} = \begin{cases} w_t + y_t \phi(x_t) & \text{if there was a mistake,} \\ w_t & \text{else.} \end{cases}$$

The perceptron was given by

$$\begin{aligned} w &= \sum_{i=1} \alpha_i y_i \phi(x_i) \\ &= \sum_{i=1}^{n} \alpha_i \; y_i \phi(x_i). \end{aligned}$$

These α_i are very useful. We can rewrite the problem of finding the best w into the problem of finding the correct α_i. The kernel perceptron goes as follows:

- Initialize $\alpha_i = 0$ for all $i \leq n$
- Repeat for $t = 1, \ldots$. It there is a mistake

$$\begin{aligned} 0 &\geq y_t \left(\sum_{i=1}^{n} \alpha_i y_i \phi(x_i)^\top \phi(x_t) \right) \\ &= y_t \left(\sum_{i=1}^{n} \alpha_i y_i k(x_i, x_t) \right) \end{aligned}$$

  update

$$\alpha_t \leftarrow \alpha_t + 1.$$

Note that at no point of time we compute $\phi(x)$ so it does not matter what the dimensionality of ϕ is so long as we can efficiently compute the kernel $k(x_t, x_i)$. The dimensionality can be infinite also.

Example 9 (Quadratic kernel)**.** Consider the simple quadratic kernel

$$k(x, x') = \left(x^\top x' \right)^2$$

What is the corresponding $\phi(x)$?

Kernels look like dot-products of feature vectors. As the previous example showed, it may often be easier to come up with kernels than feature vectors. But are they equivalent? We know that a feature vector defines a kernel. Is every kernel an inner product?

Theorem 10 (Mercer's Theorem)**.** *For any symmetric function $k : X \times X \to \mathbb{R}$ which is square integrable in $X \times X$ and satisfies*

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

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

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

for all $x, x' \in X$.

This theorem shows that given any kernel that satisfies some regularity properties we can rewrite it as an inner product. The summation over λ_i can actually be infinite. Can you give an example of a kernel that has an infinite-dimensional feature vector?

Define the kernel matrix K as

$$K_{ij} = \phi(x_i)^\top \phi(x_j)$$

for $x_i, x_j \in D_{\text{train}}$. What is the dimensionality of K?

The most important property of kernels is that they are **positive semi-definiteness** The kernel matrix of any dataset is positive semi-definite:

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

This is easy to show.

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

The double integral in Theorem 10 is really just the continuous analogue of the vector-matrix-vector multiplication above.

Lecture 3

Back-propagation

Reading

- Goodfellow Chapter 6.3-6.5
- Course notes from Stanford's CS 231n http://cs231n.github.io/optimization-2/ and http://cs231n.github.io/convolutional-networks/

3.1 Admin

1. Pratik is traveling, his office hour on Tue (09/10) 4pm will be held on Thu (09/12) 4pm (only for this week).
2. Problem Set 1 will go online on Wed 09/11, will be due on 09/25. It will be on the material in Lectures 2-7. Do not worry if you can't solve all the problems right-away; we have not covered all the material yet.

3.2 Finish/recap material from previous lecture

Question 11 (What are the key problems with kernels?)**.**

3.3 Learning the feature vector

Consider the perceptron: the perceptron had the mapping

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

that goes from the input x to the outputs before the sign function. The perceptron acting on features had the mapping

$$\begin{aligned} f(w; x) &= \text{sign}\left(w^\top \phi(x)\right) \\ &= \text{sign}\left(\sum_{i=1}^{p} w_i\, \phi_i(x)\right) \end{aligned}$$

The kernel perceptron had the mapping

$$f(w; x) = \text{sign}\left(\sum_{i=1}^{n} w_i\, k(x_i, x)\right)$$

where $k(\cdot, \cdot)$ is any positive semi-definite, symmetric function. Let us create a special bank of random features which has the mapping

$$f(w; x) = \text{sign}\left(\sum_{i=1}^{p} w_i\, \sigma\left(S_i^\top x\right)\right) \tag{3.1}$$

where

$$\sigma : \mathbb{R} \to \mathbb{R}$$

is some nonlinear function and $(S_1, \ldots, S_p) = S \in \mathbb{R}^{p \times d}$ is a matrix of random entries. This is a special feature vector where we have defined

$$\phi_i(x) = \sigma(S_i^\top x)$$

as the feature. This is the random features approach in (Rahimi and Recht, 2008).

Remark 12 (Abuse of notation)**.** The nonlinearity $\sigma : \mathbb{R} \to \mathbb{R}$. For convenience, we will use the same symbol for point-wise nonlinearities, i.e.,

$$\sigma(x) = \begin{bmatrix} \sigma(x_1) & \sigma(x_2) & \ldots & \sigma(x_d) \end{bmatrix}^\top.$$

We can now rewrite (3.1) as

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

If S and σ are identity, we are back to the linear perceptron; if σ is identity and $\phi_i(x) = S_i^\top x$ are have the feature-based perceptron for linearly sketched features. If σ is a positive definite nonlinearity and S_i is the training set, we get the kernel perceptron.

Two key ideas behind multi-layer (deep) neural networks are:

1. features are learnt instead of being hand-designed.
2. multiple compositions of the features are used to create new features

Take the random features in (3.1). The parameters we seek are $w_1, \ldots, w_p$; everything else, the nonlinearities σ and the sketching matrix S, are fixed. Neural networks would instead learn both w and S from the data. Let us instantiate this idea.

Remark 13 (Space of functions spanned by neural network)**.** If we compose the functional mapping for L layers, we get

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

Each intermediate output corresponds to a different feature space. This is a nonlinear mapping in both the inputs x and the parameters $S_1, \ldots, S_L$. Remember the mapping of the perceptron

$$f(w; x) = \sum_{i=1}^{n} \alpha_i y_i\, x_i^\top x$$

which had a nice inner product structure. A deep neural network is not a linear operator, so we cannot check if the trained network has such a nice inner product structure; we cannot easily write down what values $S_1, \ldots, S_L$ have after training completes.

Remark 14 (Some jargon)**.** The nonlinearities in neural networks are often called activation functions. This is because they are imagined to be similar to the activation function of the neurons in the brain. The McCulloch and Pitts neuron had a threshold nonlinearity

$$\sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else.} \end{cases}$$

This is not continuous and is hard to train. Popular activation functions therefore are:

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

Non-linearities affect the (i) function computed by the neural network and (ii) learning process. Typically the second one is more difficult to get right.

3.4 The back-propagation algorithm

For a perceptron, fitting the training data involves solving an optimization problem of the form

$$\begin{aligned}&\min_{w} \sum_{i=1}^{n} \ell_{\text{hinge}}(w; x_i, y_i) \\ &= \min_{w} \sum_{i=1}^{n} \max(0, -y_i w^\top x_i).\end{aligned}$$

This does not look very different for a neural network

$$\begin{aligned}&\min_{w, S_1, \ldots, S_L} \sum_{i=1}^{n} \ell_{\text{hinge}}(w, S_1, \ldots, S_L;\ x_i, y_i) \\ &= \min_{w, S_1, \ldots, S_L} \sum_{i=1}^{n} \max\left(0, -y_i \left(w^\top \sigma\left(S_L \ldots \sigma\left(S_2\, \sigma(S_1 x)\right) \ldots\right)\right)\right).\end{aligned} \tag{3.2}$$

Note that the minimization is over all parameters of the neural network: this involves the layer-wise mappings $S_1, \ldots, S_L$ as well as the weights of the final layer which we denoted by w.

Like the previous lecture, we will solve this problem by taking gradient steps on one input at each time-step. Let us however consider a simpler loss so that we can write everything down clearly. Consider a one hidden layer neural network being trained for a regression task.

$$\begin{aligned}\ell(w, S) &= \sum_{i=1}^{n} \left(y_i - f(w, S; x_i)\right)^2 \\ &= \sum_{i=1}^{n} \left(y_i - w^\top \sigma(S x_i)\right)^2\end{aligned} \tag{3.3}$$

The gradient descent equations for each parameter look as follows:

$$\begin{aligned}w^{t+1} &= w^t - \eta\, \partial_w \ell(w^t, S^t) \\ S^{t+1} &= S^t - \eta\, \partial_S \ell(w^t, S^t).\end{aligned}$$

This is a bit hard to understand: we are taking a partial derivative with respect to a matrix S or vector w. Let us unroll the above two equations.

$$\begin{aligned}w_j^{t+1} &= w_j^t - \eta\, \partial_{w_j} \ell(w^t, S^t) \\ S_{jk}^{t+1} &= S_{jk}^t - \eta\, \partial_{S_{jk}} \ell(w^t, S^t).\end{aligned}$$

When you expand out the partial derivative, you will realize that the compositionality of the mapping gives you a way to compute these derivatives in a specific order cheaply, namely from the outermost round bracket in (3.3) to the innermost round bracket. Let's write down the gradient for the parameters in the outermost round bracket

$$\partial_{w_j} \ell(w^t, S^t) = -2 \sum_{i=1}^{n} \underbrace{\Big(y_i - w^\top \sigma(S x_i)\Big)}_{\text{residual} \quad \Delta_{ji}} \sigma(S_j^\top x_i)$$

Let us define the residual term, i.e., the difference between the targets y_i and the output of the neural network $w^\top \sigma(S x_i)$ as Δ_{ji}. Of course to compute this output, you need access to all the weights w, S. The second derivative goes as follows:

$$\partial_{S_{j,k}} \ell(w^t, S^t) = -2 \sum_{i=1}^{n} \underbrace{\Big(y_i - w^\top \sigma(S x_i)\Big) w_j \sigma'(S_j^\top x_i)}_{\nu_{ik}} x_i^k.$$

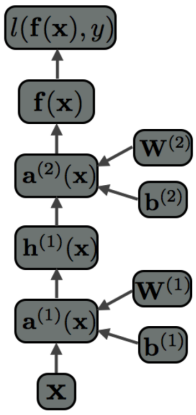
Let us define the stuff under the curly bracket as ν_{ik}. This is an interesting quantity. It is the product of the residual Δ_{ji} and the derivative of the *layer-specific* mapping $w^\top \sigma(S_j^\top x_i)$ with respect to $S_{j,k}$. We can exploit this observation to write

$$\nu_{ik} = \Delta_{ji} \, w_j \sigma'(S_j^\top x_i). \tag{3.4}$$

This gives a natural way to understand back-propagation. Mathematically, it is exactly the chain rule in calculus. The power of back-propagation comes from its implementation in code (you will implement this in your problem set).

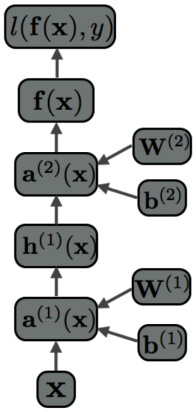
Computational Flow Graph

- 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



Each layer is equipped with two member functions

```
def forward(self, h_{k-1}, S_k):
    # computes the output of the layer
    # given output of previous layer h_k and
    # parameters of current layer S_k
    h_k = sigma(dot(S_k, h_{k-1}))
    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}}
```

The implementation can be separated into two phases.

1. Forward propagation: where given an input x_i we run through the function $w^\top \sigma(S_L \dots \sigma(S_1 x_i) \dots)$ to compute the output and the loss $\ell(w, S_1, \dots, S_L;\ x_i, y_i)$. This is a simple computation.

2. Backward propagation: each layer takes as input quantities like Δ_{ji} from the layer above it and computes ν_{ik} and a quantity similar to $\Delta_{\ell i}$ for the layer below it.

Pseudo-code for these operations is given above. The caching of quantities like h_k makes back-prop efficient. The backward function does not have to compute h_k again.

Example 15 (Convenient notation for doing backprop calculations)**.** The biggest benefit of deep learning libraries is that the backward function is defined automatically, you rarely have to write the backprop gradients yourself. It is however important to develop intuition of how the gradients "flow". Let us introduce a simple example and some notation. We will consider the linear regression problem with one layer and one data sample:

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

It is useful to draw a computational graph of the above expression

$$w, x \longmapsto z \overset{\sigma}{\longmapsto} h \overset{y}{\longmapsto} \ell.$$

where $h = \sigma(z)$ and $z = w^\top x$. Each node in this graph will have a forward and backward function. Denote

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

for any quantity v. After the forward propagation which computes the values $y - w^\top x$ and

ℓ, we are interested in computing the values of $\overline{w}, \overline{x}$.

$$\begin{aligned}
\overline{\ell} &= 1 \\
\overline{h} &= \overline{\ell}\, \frac{\mathrm{d}\ell}{\mathrm{d}h} \\
&= \overline{\ell}(h - y) \\
\overline{z} &= \overline{h}\, \frac{\mathrm{d}h}{\mathrm{d}z} \\
&= \overline{h}\, \sigma'(z) \\
\overline{w} &= \overline{z} \frac{\partial z}{\partial w} \\
&= \overline{z}\, x \\
\overline{x} &= \overline{z} \frac{\partial z}{\partial x} \\
&= \overline{z}\, w.
\end{aligned}$$

What is the complexity of computing all the gradients? A key observation is that backprop gives the derivative of the loss with respect to the input image x, for free. This is used in many places, for instance to compute adversarial samples.

Remark 16 (Hebbian learning)**.** Is back-propagation the only way to train neural networks?

Lecture 4

Convolutional neural networks, Neural architectures

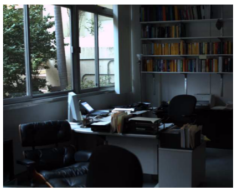
Reading

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

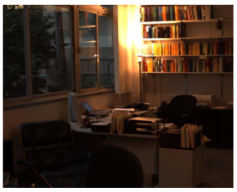
4.1 Convolutional layers

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

- They require a lot of parameters. If an input image is of size $100 \times 100 = 10^4$ grayscale pixels and we would like to classify it as belonging to one out of 1000 classes, we need 10M parameters. It is difficult to perform so many add-multiply operations quickly even on sophisticated hardware. The curse of dimensionality never goes away; we need lots of data fit these many parameters.
- Natural data is full of "nuisances" that are not useful for tasks such as classification. E.g., illumination, viewpoint, and occlusions

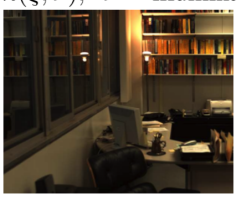


$I = h(\xi, \nu)$



$\tilde{I} = h(\xi, \tilde{\nu}),\ \ \tilde{\nu} = \text{illumination}$

$\tilde{\nu} = \text{visibility}$



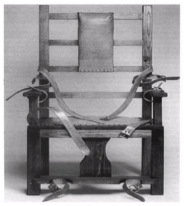
$\tilde{\nu} = \text{viewpoint}$

$\tilde{I} = h(\tilde{\xi}, \tilde{\nu}),\ \ \tilde{\xi} \neq \xi$

or semantic ones shown below.

I_1

I_2



similar shape?

similar appearance?

$\ldots$

I_n

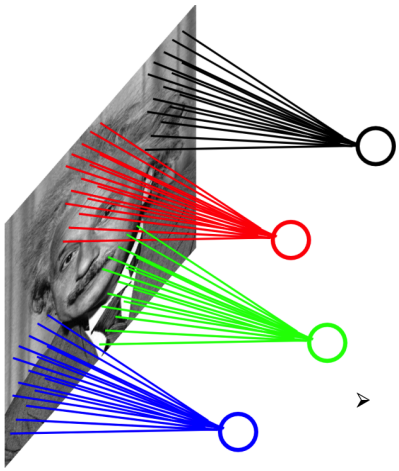
similar function?

??

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 have a group structure, e.g., image of the same chair taken from different vantage points. Some others do not have a group structure, e.g., occlusions. Convolutional operators provide a simple way to build "translational equivariance". We will expand upon this shortly.

An example using local connections



How many connections will this have?

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

$$\sum_{j=1}^{p} w_j \sigma(S_j^\top x).$$

The basic unit of a convolutional neural network is

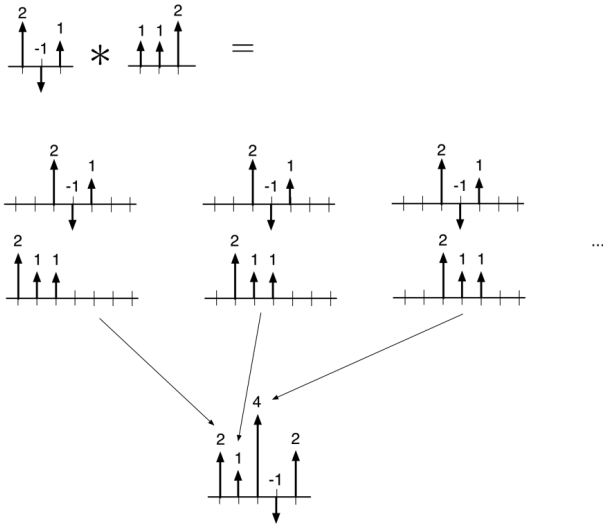
$$\sum_{j=1}^{p} w_j \sigma(x * S_j)$$

where the convolution between two one-dimensional vectors $x \in \mathbb{R}^d$ and $s \in \mathbb{R}^K$ is defined as

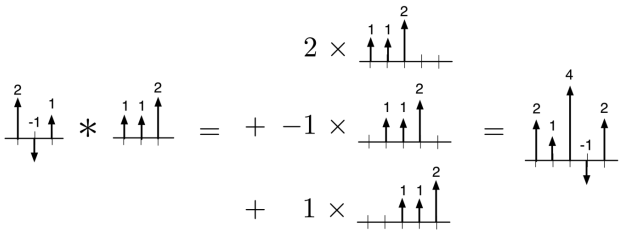
$$(x * s)_k = \sum_{k'=-\infty}^{\infty} x_{k'} \; s_{k-k'}$$

with the convention that s_{-k} is the k^{th} index from the right of the flipped signal s.

Example 17 (Flip and filter)**.** This is the straightforward way to implement the above sum.



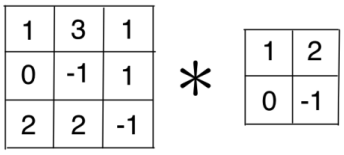
Example 18 (Translate and scale)**.** Let us look at convolution using the "translate and scale" style of computation.

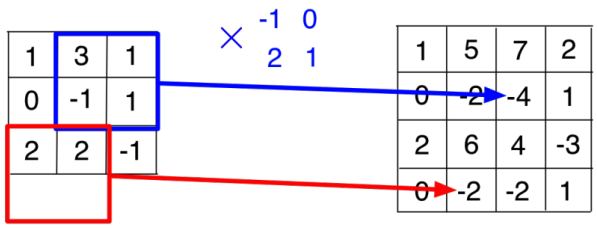


Convolutions work in the same way for two-dimensional or three-dimensional signals. The kernel will be a matrix of size $K \times K$ in the former case and a tensor of size $K \times K \times K$ in the latter.

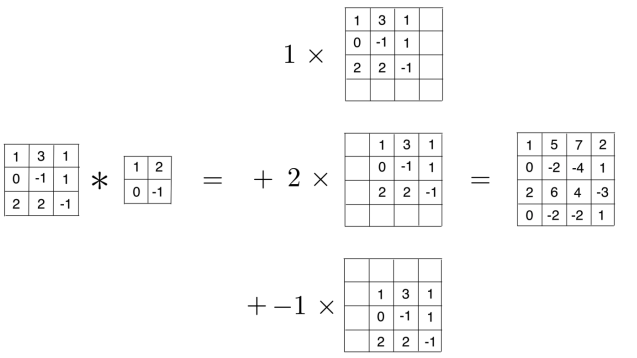
$$(x * k)_{i,j} = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} x_{s,t}\, k_{i-s,j-t}.$$

Flip and filter style, notice the flipping of the kernel.





Translate and scale style.

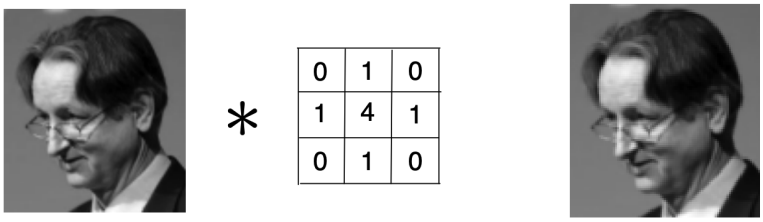


Note that

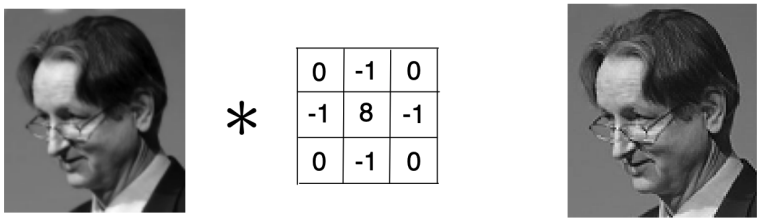
1. Convolution is a linear operator; how can we write it as a matrix-vector multiplication? We take the kernel, flip it and sweep it left and right to get the rows of the matrix.

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

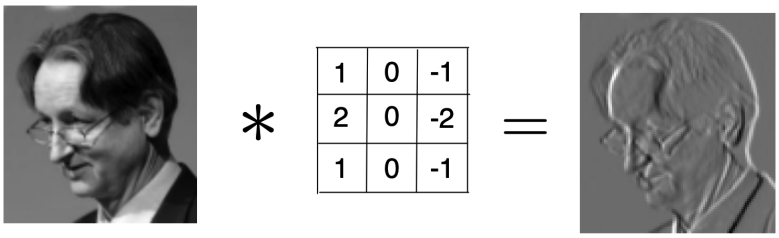
2. Lots of non-trivial transformations of the image are possible using slight changes in the weights. E.g., blurring



or sharpening using a slight change in the weights



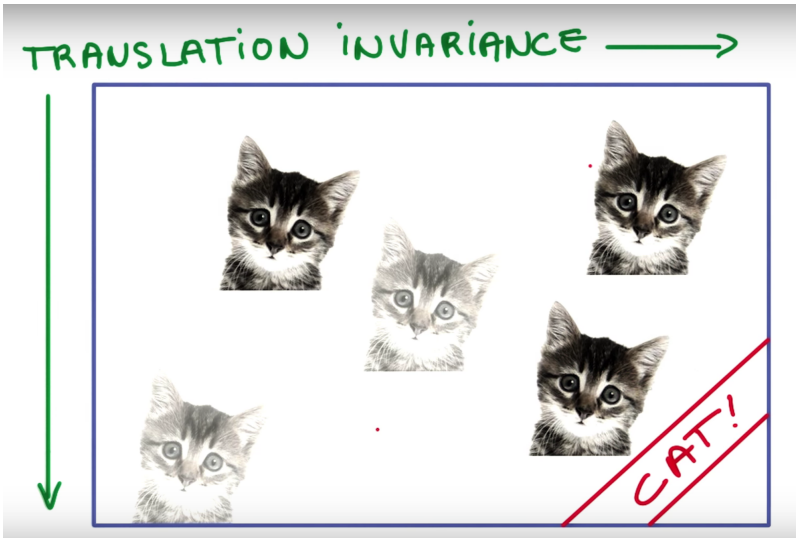
We can also detect edges



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. A dot product which is what we saw for the perceptron takes a vector and returns a real number. A convolution operator instead returns an entire vector. The number of parameters has been reduced: it is pK right now instead of pd. The important observation is however that there is parameter sharing that takes place among different elements of the vector x. This was not happening for the fully-connected network. Here is what it buys us



5. What if the size of the cat's head was very large? How should we pick the kernel size to identify objects are different scales?

You can find some animations at https://colah.github.io/posts/tags/convolutional_neural_networks.html.

4.1.1 How are convolutions implemented in practice?

The most heavily used operator in a deep network. Need to implement it as efficiently as possible. There are a few different ways of implementing it.

1. Write a for loop. This works well if the kernel is small in size.

2. Expand out the kernel as a matrix. This is the one that is mostly popularly implemented. Works well for sizes up to 5×5.

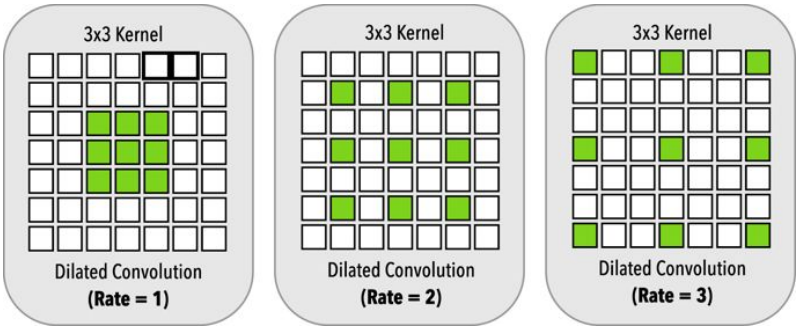
3. Using the Fast Fourier Transform (FFT)

$$(x * s) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{x\right\}.\mathcal{F}\left\{s\right\}\right\}$$

   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. But fact remains that large convolutional kernels which we would like are more expensive to compute than small convolutional kernels.

Remark 19 (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.

Remark 20 (Separable convolutions)**.** There are 9 weights in a 3×3 kernel. Convnets 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

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

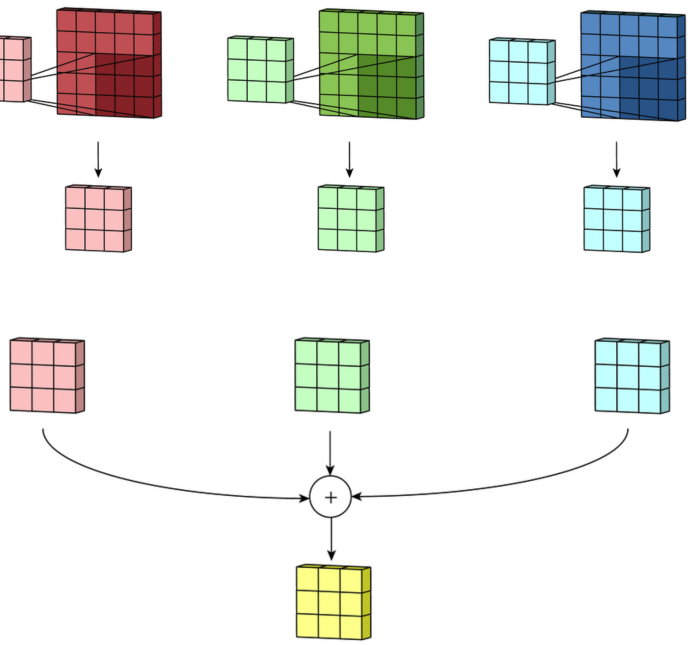
Using the original kernel requires 9 multiply operations to compute each pixel value. Using the split kernels requires only 6, it also has fewer parameters (6). These are called separable convolutions. The Sobel filter which we saw before can be written as a separable convolution

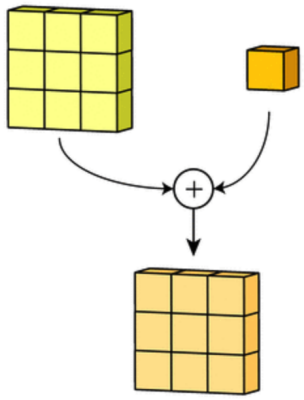
$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}.$$

Question 21. Can any kernel be written as a separable convolution?

4.1.2 Convolutions for multi-channel images

How do convolutions work for RGB images? We have a separate kernel for each input channel





Question 22. Is the convolutional operation in a standard CNN implemented in the "separable" along the channels?

Question 23. How many parameters does one convolutional layer have?

Remark 24 (1×1 convolutions)**.** Let's imagine that we created 1×1 convolutions, what would be the utility of these?

4.1.3 Pooling

Convolutions only achieve translational equivariance. Can you guess the difference between equivariance and invariance? Think about how the last layer looks if the cat is the top left corner of the image versus if the cat is in the botton right part of the image. The top layer of the network w was trained to detect cats in the top left part of the images. Even if the convolutional filter picked up the features of the cat in the bottom half because of weight sharing, the top-most layer will not be able to identify it has a cat, its weights for that part of the image are different.

This is why neural networks perform the pooling operation. The average pooling operation is given by a convolutional kernel

$$K_{\text{avg-pool}} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}. \tag{4.1}$$

It is still a linear operator. It does not solve our problem entirely, as we saw above, it kind of blurs the cat. We can stack up one average pooling layer after each convolutional layer and

increase the effective receptive field but we can also use a max-pooling operator.

$$(\text{max-pool}(x))_{ij} = \max_{-K \le s \le K} \max_{-K \le t \le K} x_{i-s,j-t} \tag{4.2}$$

How does this build invariance to small transformations of the input?

Typically, max-pooling is done with a "stride" equal to the size of the kernel. This reduces the size of the image by a factor of K. It loses lots of information and should be used sparingly.

4.1.4 Backpropagation for convolutional and pooling layers

Let us write the output of a convolutional layer in full detail.

$$h_{k',i',j'} = b_{k'} + \sum_{i=1}^{K} \sum_{j=1}^{K} \sum_{k=1}^{K} w_{i,j,k,k'} \, x_{i'+i-1,j'+j-1,k}$$

Use the backpropagation notation from the previous lecture to get

$$\begin{aligned} \overline{w_{i,j,k,k'}} &= \sum_{i',j',k'} \overline{h_{k',i',j'}} \, \frac{\partial h_{k',i',j'}}{\partial w_{i,j,k,k'}} \\ &= \sum_{i',j',k'} \overline{h_{k',i',j'}} \, x_{i'+i-1,j'+j-1,k}. \end{aligned}$$

Question 25. What is the gradient

$$\overline{x_{i,j,k}}$$

for a convolutional layer?

The max-pooling layer is a much simpler operation. We can write it as

$$\begin{aligned} h_{i',j'} &= \max_{i,j \in \Omega(i',j')} x_{i,j} \\ &= \max_{i,j} \mathbf{1}\left\{ij \in \Omega(i',j')\right\} \, x_{i,j} \end{aligned}$$

This layer has no parameters of its own. So we only need to compute

$$\begin{aligned} \overline{x_{i,j}} &= \sum_{i',j'} \overline{h_{i',j'}} \, \frac{\partial h_{i',j'}}{\partial x_{i,j}} \\ &= \overline{h_{i',j'}} \, \mathbf{1}\left\{ ij = \operatorname*{argmax}_{i'',j'' \in \Omega(i',j')} x_{i'',j''} \right\} \end{aligned}$$

In other words only the winning input pixel gets the backprop gradient back. All others get zero gradient.

Lecture 5

Neural architectures, Regularization

Reading

- Goodfellow 7.1-7.5, 7.8, 7.12
- Bishop 3.3, 5.5

5.1 Neural architectures

Let us build a CNN. Our input images are of size $3 \times 32 \times 32$.

```
import torch as th
import torch.nn as nn

def conv_relu(cin, cout, ksz, stride=1, padding=0):
    return nn.Sequential(
        nn.Conv2d(cin, cout, ksz, stride, padding),
        nn.ReLU())

model = nn.Sequential(
    conv_relu(3, 8, 5, 1, 2),        # output size is 1x8x32x32
    nn.MaxPool2d(3),                 # output size is 1x8x10x10
    conv_relu(8, 8, 5, 1, 2),        # output size is 1x8x10x10
    nn.MaxPool2d(3),                 # output size is 1x8x3x3
    conv_relu(8, 8, 5, 1, 2),        # output size is 1x8x3x3
    nn.AvgPool2d(3),                 # output size is 1x8x1x1
    )
```

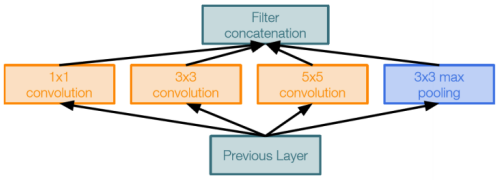
```
x = th.randn(1,3,32,32)      # batch-size of 1
yh = model.forward(x)
```

Remark 26 (Adaptive average pooling)**.** What happens if we feed images of a different size to the network? The convolutional layers are still well-defined. But we get an output of a different size. The adaptive average pooling operator is convenient to use in these cases. It adapts the size of the averaging kernel to always give an output of a fixed size.

```
# target output size of 5x7
m = nn.AdaptiveAvgPool2d((5,7))
input = torch.randn(1, 64, 8, 9)
output = m(input)
```

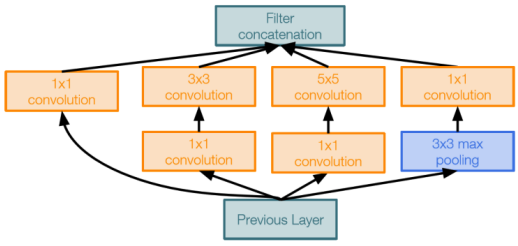
5.1.1 Inception module from Google LeNet

Large convolutional kernels are desirable because they have a large receptive field. They are however computationally expensive. Further, a large kernel can certainly identify objects that are smaller than the kernel size but it leads to a large waste of the parameters. Consider the computational block below.



Naive Inception module

This block splits the computation using filters of different sizes and concatenates the output at the top. It works, but what could be a problem with this?



Inception module with dimension reduction

5.1.2 Some general advice about picking architectures

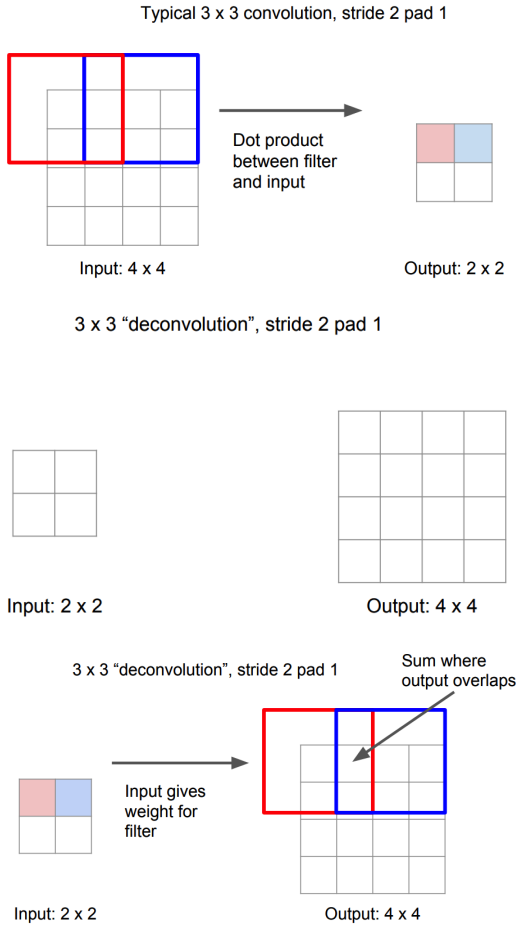
- Designing good architectures is hard. One has to pick many parameters: kernel sizes, strides, number of channels, pooling sizes and strides etc. Each candidate configuration needs to be ideally tested against multiple datasets with good training methodology before we can claim that we have discovered a new architecture.
- Many architectures have been discovered to be performing well over the past years. To name a some popular ones
  - LeNet, AlexNet, VGG, Inception, ResNet, WideResNet, DenseNet
  - MobileNet, SqueezeNet
  - SiameseNet

  You should use these architectures are building blocks when you solve new problems. Instead of going for the latest and fanciest architectures, start from the simplest existing architecture and build up, e.g., today's reading (Springenberg et al., 2014) is a very simple architecture with about 1.5M weights that often works better than architectures $10\times$ its size.
- Roughly speaking, a good architecture (pick any of the above) is more or less going to work for a new problem. It is rare that architecture A gives 40% accuracy on your dataset and architecture B gives 90% accuracy, other factors such as training a network well, regularization (which we will cover next) matter much more.

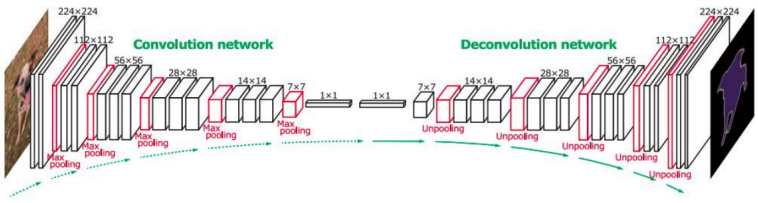
5.1.3 Upsampling (Deconvolutional) operator

A typical classification network looks like a pyramid. The input image size starts at, say, 100×100 and sequentially decreases due to max-pooling, or strided convolutions to give an output that is equal to the number of classes, sy 1000. What if we want to predict larger objects? Say you'd like to train a model for solving image segmentation

The input data is a raw image, the annotations/labels in this case are “dense”, e.g., for each pixel in the image you have a label that indicates which class the pixel belongs to (person A, person B, road, car 1, car 2, background etc.) We would like to train a network that predicts a similar segmentation on new images. How should we do this?



This operator where the convolutional kernel is such that it increases the size of the image upon which it operates is called a “deconvolutional” operator in the literature. This is an incorrect name because no real deconvolution is performed here. You can instead call it transposed convolution (think of the matrix-vector implementation), “upsampling”, convolution with 1/2 strides etc. Nevertheless, this is an important operator in computer vision. You can read more about it at https://distill.pub/2016/deconv-checkerboard.



5.2 Regularization

Neural networks have lots of parameters. The architectures we have considered above have of the order of 10M parameters. We are fitting them on datasets that are as small as 50,000 CIFAR-10 images.

5.2.1 Bias-Variance Decomposition

Remember the finite dataset setting that we have been discussing till now. Given a dataset $D = \{(x_i, y_i)\}_{i=1,\dots,n}$, a loss function $\ell(x, y)$ we train a model $h \in \mathcal{H}$ where $\mathcal{H}$ is some hypothesis class, say the set of all neural networks of a given architecture, that minimizes

$$\widehat{\mathcal{R}}(h) = \frac{1}{n}\sum_{i=1}^{n} \ell(h(x), y). \tag{5.1}$$

This problem is called emprical risk minimization and the objective is called the empirical risk of a classifier $h \in \mathcal{H}$. We however really care for the performance of h on new data so we would like h^* to also have good population risk

$$\mathcal{R}(h) = \mathop{\mathbb{E}}_{(x,y)\in\mathcal{P}} \left[\ell(h(x), y)\right]. \tag{5.2}$$

Let us understand the difference between the two quantities in the case when the loss is ℓ is quadratic:

$$\mathcal{R}(h) = \int |h(x), y'|^2 \; p(x, y') \; \mathrm{d}x\mathrm{d}y'.$$

The minimizer of the population risk (you can take the derivative with respect to $h(x)$) is the conditional expectation

$$h^*(x) = \mathop{\mathbb{E}}_{y'} \left[y'|x\right].$$

Question 27 (Can we use the conditional distribution directly in practice?)**.**

Now perform some algebra

$$\begin{aligned}(h(x) - y)^2 &= \left(h(x) - \mathop{\mathbb{E}}_{y'} \left[y'|x\right] + \mathop{\mathbb{E}}_{y'} \left[y'|x\right] - y\right)^2 \\ &= \left(h(x) - \mathop{\mathbb{E}}_{y'} \left[y'|x\right]\right)^2 + 2\left(h(x) - \mathop{\mathbb{E}}_{y'} \left[y'|x\right]\right)\left(\mathop{\mathbb{E}}_{y'} \left[y'|x\right] - y\right) + \left(\mathop{\mathbb{E}}_{y'} \left[y'|x\right] - y\right)^2\end{aligned}$$

When we substitute this into the formula for $\mathcal{R}(h)$ we get

$$\mathcal{R}(h) = \mathop{\mathbb{E}}_{x}\left[(h(x) - \mathop{\mathbb{E}}_{y'}\left[y'|x\right])^2\right] + \mathop{\mathbb{E}}_{x}\left[(\mathop{\mathbb{E}}_{y'}\left[y'|x\right] - y)^2\right]. \tag{5.3}$$

This decomposition is very telling. The first term is how far our hypothesis $h(x)$ is from the true minimizer of the population risk, the conditional expecatation $\mathbb{E}_{y'}\left[y'|x\right]$. The second term does not even have $h(x)$, it is not in our hands really; it determines variance of the data itself. It is a lower bound for the population risk for any classifier. It is called the Bayes error

$$\text{Bayes error} = \mathop{\mathbb{E}}_{x}\left[(\mathop{\mathbb{E}}_{y'}\left[y'|x\right] - y)^2\right]$$

and it is zero if the true model is deterministic.

Let us write the conditional expectation $\mathbb{E}_{y'}\left[y'|x\right]$ as simply $\mathbb{E}\left[y'|x\right]$. The first term in (5.3) for a given classifier looks like

$$\left(h(x;D) - \mathbb{E}\left[y'|x\right]\right)^2;$$

we have made it clear that this quantity depends on our training dataset D. Consider the following computation

$$\begin{aligned}
(h(x;D) - \mathbb{E}\left[y'|x\right])^2 = \\
&= \left(h(x;D) - \mathop{\mathbb{E}}_{D}\left[h(x;D)\right] + \mathop{\mathbb{E}}_{D}\left[h(x;D)\right] - \mathbb{E}\left[y'|x\right]\right)^2 \\
&= \left(h(x;D) - \mathop{\mathbb{E}}_{D}\left[h(x;D)\right]\right)^2 + \\
&+ \left(\mathop{\mathbb{E}}_{D}\left[h(x;D)\right] - \mathbb{E}\left[y'|x\right]\right)^2 + \\
&+ 2\left(h(x;D) - \mathop{\mathbb{E}}_{D}\left[h(x;D)\right]\right)\left(\mathop{\mathbb{E}}_{D}\left[h(x;D)\right] - \mathbb{E}\left[y'|x\right]\right).
\end{aligned}$$

Take the expectation with respect to D to get

$$\begin{aligned}
&\mathop{\mathbb{E}}_{D}\left[(h(x;D) - \mathbb{E}\left[y'|x\right])^2\right] = \\
&\underbrace{\left(\mathop{\mathbb{E}}_{D}\left[h(x;D)\right] - \mathbb{E}\left[y'|x\right]\right)^2}_{\text{bias}} + \underbrace{\mathop{\mathbb{E}}_{D}\left[\left(h(x;D) - \mathop{\mathbb{E}}_{D}\left[h(x;D)\right]\right)^2\right]}_{\text{variance}}
\end{aligned} \tag{5.4}$$

The first term is called the bias: it is the gap between the optimal classifier and the per-

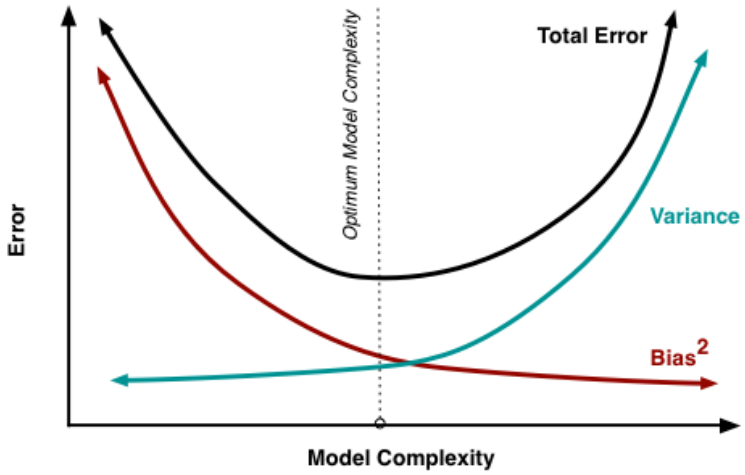
formance of our candidate $h(x)$ when averaged over all datasets D. If we have a large hypothesis class, we can make the bias small. The second term is called the variance: it measures how sensitive our candidate classifier is to the dataset it was trained on. If our training dataset is small, the variance can be large because we do not get a good

(Picture of bias-variance tradeoff)

Remark 28 (Some caveats)**.** The bias-variance decomposition comes with certain caveats.

(a) It is of limited practical value because it is based on averages with respect to ensembles of datasets, whereas in practice we have only the single observed data set. If we had a large number of independent training sets of a given size, we would be better off combining them into a single large training set, which of course would reduce the level of over-fitting for a given model complexity.

(b)



(c) It has not been observed to be true for neural networks. There are large debates on what the correct notion of model complexity is for neural networks which makes plotting this curve difficult. We will look at a few different arguments around this trade-off for deep networks in the module on generalization.

5.2.2 Weight decay

Let's say we picked a really large hypothesis class and are unsure whether the variance in the bias-variance trade-off will be large. We can either collect more data, or we can regularize the classifier to reduce its effective capacity. We will denote all regularizers as modifying our original loss function as

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

Weight decay then sets

$$\Omega(w) = \frac{\lambda}{2} w^\top w. \tag{5.5}$$

Remark 29 (Motivation from using a Bayesian prior on the parameters)**.**

Question 30 (Should we use weight decay on the biases of a neural network?)**.**

Remark 31 (Homotopy continuation)**.** Homotopy continuation is an approach to solving a system of equations by tracking the solutions of "nearby" systems of equations. Weight decay keeps the parameters close to the origin in its mathematical formulation. How does this reflect in the updates of the gradient descent? Instead of the original updates

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

we now have

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

The original

Remark 32 (Scaling of the weights in a neural network)**.** The multi-layer nature of the deep network needs to be understood together with weight decay. Let us consider a simple two-layer network given by

$$h_j = \sigma(\langle w^j, x \rangle + w^{j,0})$$
$$y_k = \left\langle v^k, h \right\rangle + v^{k,0}.$$

where $x \in \mathbb{R}^d$ is the data, h are the activations of the first layer and the vector y are all the output neurons. If we scale the input x by a scalar constant c

$$\tilde{x} = cx + d$$

we can change the network in a way that the activations do not change. We simply consider the weights w^j scaled down by c

$$\tilde{w^j} = \frac{w^j}{c}$$
$$\tilde{w^{j,0}} = w^{j,0} - \frac{d}{c} \sum_i w_i^j$$

Similarly, if the labels y were scaled in our dataset by a constant c and a bias d, we can predict equally well by using

$$\tilde{v^k} = cv^k$$
$$\tilde{v^{k,0}} = cv^{k,0} + d.$$

as the weights of the second layer instead of the original weights. It is reasonable to ask that the training process is consistent, i.e., if the dataset transforms the inputs or outputs in some simple way, in this case linearly, the classifier is also a simple transformation of the original classifier, in this case a linear transformation.

> Regularizers should reduce the total number of solutions, they should not arbitrarily favor one equivalent solution over the other.

Weight-decay in particular treats weights and biases exactly the same way. Even if we knew that our dataset was scaled, we cannot change the regularization parameter λ easily to keep the output of the learning process consistent. We would therefore like weight decay to be

invariant to the scaling of weights and shifts of the biases. It is simple to achieve this:

$$\begin{aligned}\Omega(w) = \frac{\lambda_1}{2} & \sum_{w_i \in \text{first layer weights, not biases}} w_i^2 \\ + \frac{\lambda_2}{2} & \sum_{w_i \in \text{second layer weights, not biases}} w_i^2 \\ + \ldots &\end{aligned} \tag{5.6}$$

If we now knew the input data was scaled by c, we would simply set

$$\tilde{\lambda_1} = c^2 \, \lambda_1.$$

which will give us the scaled down weights $\tilde{w^j}$. In other words, only apply weight decay to the convolutional weights, do not apply them to the biases. In practice, we do not want to pick all the coefficients $\lambda_1, \lambda_2, \ldots$

Lecture 6

Early stopping, Data Augmentation

Reading

- Bishop 5.5.2
- "Dropout: a simple way to prevent neural networks from overfitting" by (**?**)

Let us list down all the different regularizations that we have heard of on the Internet:

1. Weight decay and variants (ℓ_2/ℓ_1 regularization, weight smoothing, soft weight sharing)
2. Early stopping, weight averaging
3. Dropout (Maxout, Bayesian dropout, drop-connect, noise on synapses, standout)
4. Ensemble methods (boosting, bagging)
5. Batch-normalization (layer norm, group norm)
6. Data augmentation (tangent prop, mirror flips, scale, contrast normalization, color jitter, adversarial samples, test-time augmentation)
7. Multi-task learning, self-supervised learning
8. Some really funny ones: Cutout, Shake-shake

There are many others and new ones crop by every month. In the previous lecture we made a pedagogical distinction between regularization and other techniques (hacks, ways to accelerate optimization etc.). Keep in mind that if a technique does not "reduce the size of

the hypothesis class to get better generalization" we prefer not to call it regularization. For instance, if there is an algorithm that is a faster variant of stochastic gradient descent (we will see some in the later lectures), this is simply a faster optimization problem, it is not a regularizer.

6.1 Early stopping

Early stopping is a heuristic wherein we observe the validation error during the training process and stop training when the validation error starts increasing. This is very standard in machine learning.

(picture of training and validation error curves)

(Early stopping as a way of achieving weight decay)

Question 33 (Can you list down caveats of early stopping? What makes it hard to do it in practice?)**.**

Let's now see where early stopping comes from. We will resort to least squares in which case early stopping has strong connections to the minimum-norm solution.

Consider a linear model

$$Xw = Y$$

where we have strung up the data $x_i \in \mathbb{R}^d$ as rows of X and the labels $y_i \in \mathbb{R}$ as rows of Y. The least squares problem is then to find

$$\min_{w \in \mathbb{R}^d} \frac{1}{2n} \|Y - Xw\|^2.$$

For $n > d$ the solution is

$$w^\dagger = (X^\top X)^{-1} X^\top Y = X^\dagger Y. \tag{6.1}$$

The matrix $X^\dagger$ is called the left pseudo-inverse of X.

Remark 34 (Under-determined least squares)**.** It will be useful to remember that for an under-determined system with $n < d$, there are many possible solutions to the least squares problem. We have to pick one of these many solutions. We have to pick one among them, say the one with the minimum norm. The problem of finding the minimum norm solution is now formulated as

$$\min_{w \in \mathbb{R}^d} \frac{1}{2} \|w\|^2$$
$$\text{such that } Xw = y,$$

with the solution given by

$$w^\dagger = X^\top \left(X X^\top \right)^{-1} Y = X^\dagger Y. \tag{6.2}$$

Question 35. Show that the solution $w^\dagger$ is such that

$$(w^\dagger - w) \perp w^\dagger$$

for any vector $w \in \text{Null}(X)$, i.e., $Xw = 0$.

Answer: Consider any other solution w to the minimum-norm optimization problem above. We have $X(w - w^\dagger) = 0$. Now

$$\begin{aligned}
(w - w^\dagger)^\top w^\dagger &= (w - w^\dagger)^\top X^\top \left(X X^\top \right)^{-1} Y \\
&= \left(X(w - w^\dagger) \right)^\top \left(X X^\top \right)^{-1} Y \\
&= 0.
\end{aligned}$$

This shows that $(w - w^\dagger) \perp w^\dagger$. Further we can also check that the solution $w^\dagger$ indeed the minimum-norm solution.

$$\begin{aligned}\|w\|^2 &= \|w - w^\dagger + w^\dagger\|^2 \\ &= \|w - w^\dagger\|^2 + \|w^\dagger\|^2 \\ &\geq \|w^\dagger\|^2.\end{aligned}$$

In summary, for the least squares problem, we always write the solution as

$$w^\dagger = X^\dagger Y. \tag{6.3}$$

with $X^\dagger$ defined appropriately if X is a tall matrix or a fat matrix.

6.1.1 Tikhonov regularization

If some of the data are linearly correlated, the matrix $X^\top X$ can become singular in which case we perform (regularized least squares) ridge regression

$$w_\lambda^\dagger = (X^\top X + \lambda I)^{-1} X^\top Y = X_\lambda^\dagger Y. \tag{6.4}$$

This is a special case of Tikhonov regularization. Picking a good λ gives a way of massaging the original data in cases when they are noisy, correlated etc.

As the regularization coefficient $\lambda \to 0$, Tikhonov regularization reduces to

$$\lim_{\lambda \to 0} w_\lambda^\dagger = (X^\top X)^{-1} X^\top Y.$$

On the other hand, for large λ we have

$$w_\lambda^\dagger \approx \frac{1}{\lambda} X^\top Y.$$

Question 36. Draw the training and test curves as a function of time for two cases (i) where the pseudo-inverse solution is good, i.e., the data is good and (ii) where the pseudo-inverse solution is bad, so the regularized solution is better.

6.1.2 Gradient descent for linear regression

Now let's see what gradient descent does. If we minimize the original least squares objective using gradient descent, we have updates of the form

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{n} X^\top \left(X w^{(t)} - Y \right).$$

You will see the convergence rate of gradient descent for a fixed step-size in the first problem set. We know that least squares is convex and therefore these iterates of gradient descent converge to the same solution $X^\dagger Y$ obtained from the direct method. What happens if the solution is non-unique in the under-determined case? Note that if $w^{(0)} = 0$, the weight update of GD can be written as

$$w^{(t)} = X^\top \alpha$$

for some α. In other words, the residuals $Xw^{(t)} - y$ couple the original data linearly to get the iterate. Substitute this in (6.2) to see that

$$\begin{aligned} w^\dagger &= X^\top \left(X X^\top \right)^{-1} Y \\ &= X^\top \left(X X^\top \right)^{-1} \left(X w^{(t)} \right) \\ &= X^\top \left(X X^\top \right)^{-1} \left(X X^\top \alpha \right) \\ &= X^\top \alpha. \end{aligned}$$

In other words, if $w = X^\top \alpha$ for some α it is the minimum-norm solution to the least squares problem.

Gradient descent converges to the minimum-norm solution for least squares.

Now you can stop here and say, voila gradient descent regularizes machine learning. But What is the regularization parameter? And how can we control it?

Draw the different kinds of functions fitted by the optimization as a function of time.

Let us formalize this intuition.

Fact 37. For a scalar $a \in \mathbb{R}$ and $|a| < 1$ we have

$$\sum_{k=0}^{\infty} a^k = (1-a)^{-1}, \quad \sum_{k=0}^{t} a^k = (1-a^t)\,(1-a)^{-1},$$

Similarly for $|b| < 1$, we have

$$\sum_{k=0}^{\infty} (1-b)^k = b^{-1}, \quad \sum_{k=0}^{t} (1-b)^k = (1-(1-b)^t)\,b^{-1}.$$

The exact same identity is true if a and b are matrices.

Fact 38. For a matrix A, if $I - A$ is invertible and $\|A\| < 1$ we have

$$\sum_{k=0}^{\infty} A^k = (I - A)^{-1}, \quad \sum_{k=0}^{t} A^k = (1 - A^t)\,(1 - A)^{-1},$$

Similarly for an invertible matrix B with $|B| < 1$, we have

$$\sum_{k=0}^{\infty} (1 - B)^k = B^{-1}, \quad \sum_{k=0}^{t} (1 - B)^k = (I - (I - B)^t)\,B^{-1}.$$

Lemma 39 (Closed form solution of gradient descent updates for linear regression)**.** *The updates*

$$w^{(t+1)} = w^{(t)} - \frac{\eta}{n} X^\top (Xw - Y)$$

can be rewritten as

$$\begin{aligned} w^{(t+1)} &= \frac{\eta}{n} \sum_{k=0}^{t} \left(I - \frac{\eta}{n} X^\top X\right)^k X^\top Y \\ &= \left(I - \left(I - \frac{\eta}{n} X^\top X\right)^t\right) \left(X^\top X\right)^{-1} X^\top Y. \end{aligned}$$

Proof. By induction. Assuming that the lemma is true for $w^{(t)}$, use the gradient descent updates to show that it is true for $w^{(t+1)}$. ■

Now note that a magical thing happens. If the step-size of gradient descent η is such that

$$\|I - \frac{\eta}{n} X^\top X\| < 1$$

then for large time t

$$\frac{\eta}{n} \sum_{k=0}^{t} \left(I - \frac{\eta}{n} X^\top X\right)^k X^\top \approx X^\dagger.$$

We therefore have shown that

$$\lim_{t \to \infty} w^{(t)} = w^\dagger.$$

Similarly for small t,

$$w^{(t)} \approx \frac{\eta}{n} X^\top Y.$$

Now compare this with Tikhonov regularization: for large time t, gradient descent gives us

$$\text{Small } \lambda: \quad w^{(t)} \approx \left(X^\top X\right)^{-1} X^\top Y, \quad \text{Large } \lambda: \quad w^{(t)} \approx \lambda X^\top Y.$$

Early stopping has an implicit regularization effect.

The same result holds for kernel regression.

6.2 Data augmentation

Data augmentation helps the classifier capture invariances in the data. Deep networks are already invariant to translations so we do not have to worry about them. Let us look at a few others at https://docs.fast.ai/vision.transform.html.

Question 40 (What kind of augmentations should we use?)**.**

Tips for good data augmentation:

- data augmentation is a way of amplifying the amount of available data.
- this extra data has to be similar to the test data that you expect.
- else we regularize the model but waste capacity of the model by forcing it to handle unimportant nuisances.

Lecture 7

Dropout, Batch-Normalization

Reading

- "Dropout: a simple way to prevent neural networks from overfitting" by **?**
- "Batch normalization: Accelerating deep network training by reducing internal covariate shift" by **?**

We are running roughly two lectures behind the syllabus but we have remained more or less within the realms of systematic derivations. We now cross-over into the murky waters. Regularization in deep networks is a very poorly understood subject and there are some operations that (i) seem absurd but work great, and (ii) seem reasonable but are extremely challenging to get to work. We will look at Dropout and Batch-Normalization in this lecture which fit the bill on these two counts. We will end the lecture with a look at convex optimization to prepare us for the next few lectures.

7.1 Ensembles and Dropout

The key idea behind all of machine learning is "averaging always improves things". If a loss function is convex with respect to its predictions, i.e., if the prediction of the model is $\hat{y}(x; h) = h(x)$.

$$\ell(\hat{y}, y) \text{is convex with respect to } \hat{y},$$

we can always add the predictions of multiple models together, e.g.,

$$\ell\left(\lambda_1\hat{y}(x;h_1)+\ldots+\lambda_n\hat{y}(x;h_n),\quad y\right)\leq\lambda_1\ell(\hat{y}(x;h_1),y)+\ldots+\lambda_n\ell(\hat{y}(x;h_n),y)$$

for $\lambda_1+\ldots+\lambda_n=1$ and $\lambda_k\geq 0$ for all k.

It does not really matter where the different predictions came from. They can be different neural networks, a combination of random forests and neural networks, or anything else that you wish. The loss function $\ell(\hat{y},y)$ is convex in $\hat{y}$ for almost all the losses that we use in practice, e.g., squared loss, logistic regression, cross-entropy error, hinge loss etc. Combinations of multiple models are called ensembles.

Remark 41 (A trip back to bagging (bootstrap aggregating))**.** For convex machine learning models, e.g., SVMs, we can train multiple classifiers on slightly different datasets. Consider learning multiple

$$\begin{aligned}w^1&=\operatorname*{argmin}_w\sum_{(x,y)\in D^1}\ell(y,\sigma(w^\top x))+\Omega(w)\\&\vdots\\w^m&=\operatorname*{argmin}_w\sum_{(x,y)\in D^m}\ell(y,\sigma(w^\top x))+\Omega(w)\end{aligned}$$

SVMs where the datasets $D^1,\ldots,D^m$ are slight perturbations of each other, e.g., subsampling the data, adding noise to certain features, getting new data a few months down the future etc. The combined classifier where each model is weighed equally

$$w^*=\frac{1}{m}\sum_{i=1}^m w^m$$

has a lower variance in the bias-variance trade-off than the individual classifiers. This is easy to understand because we are averaging multiple models. Such bagging, which averages the classifiers without paying attention to their individual errors, does not improve the bias, the final model is exactly as complex as the individual models.

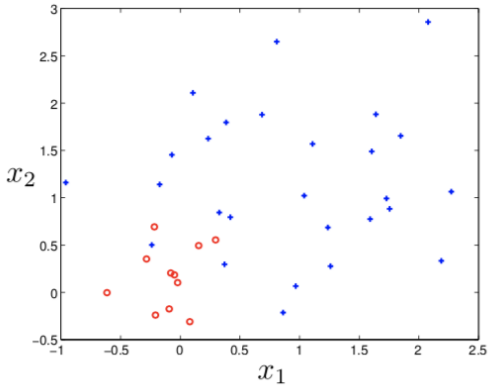
Remark 42 (A trip back to boosting)**.** Boosting is another way to combine multiple classifiers into one. It learns the classifiers in an online fashion to build an ensemble. After m rounds of boosting, the ensemble classifier has the form

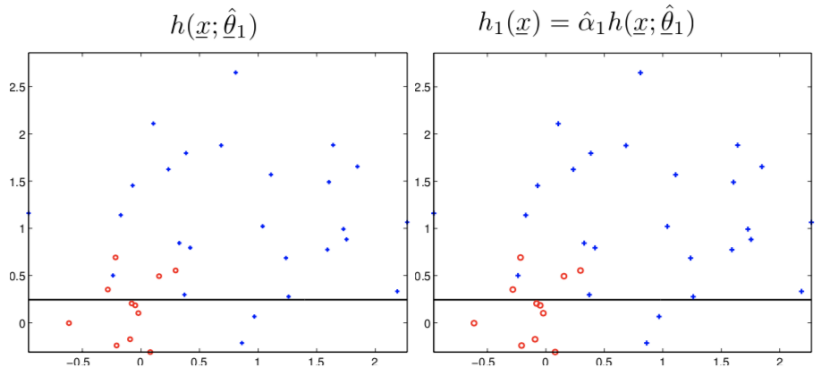
$$h_m(x)=\sum_{i=1}^m\alpha_i h(x,\theta_i)$$

where each individual model comes from our hypothesis class $h_i \in \mathcal{H}$. The weights α_i are the votes that the base classifier gets in the ensemble; these votes are a function of the number of mistakes made by the ensemble on the dataset at the i^{th} step. The ensemble can also be viewed as a linear classifier on the feature vector

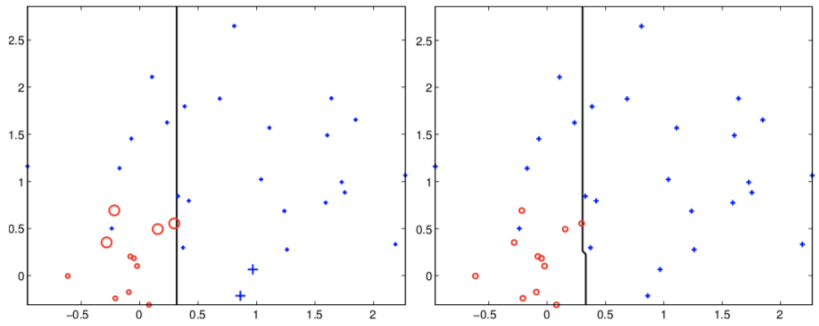
$$\phi(x) = \begin{bmatrix} h(x,\theta_1) & h(x,\theta_2) & \dots & h(x,\theta_m) \end{bmatrix}.$$

Both the individual base classifiers (weak learners) and the mixture parameters α_i are learnt in the boosting process. Let us look at the execution of Adaboost on a dataset.

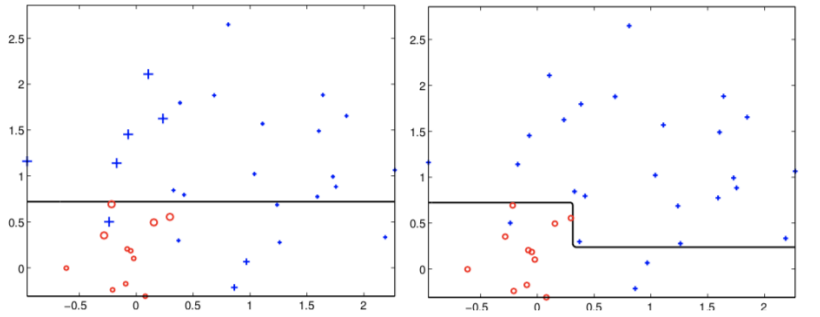




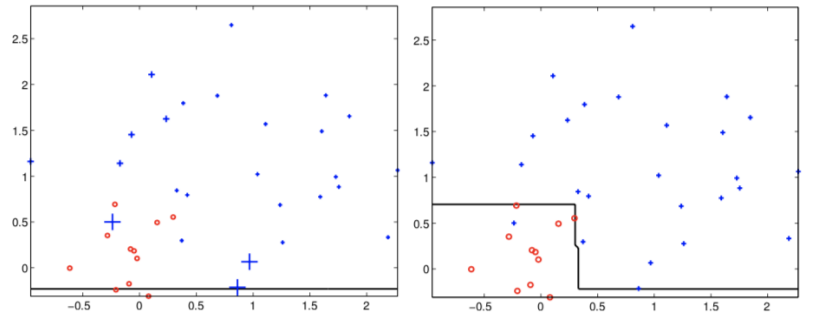
$h(\underline{x};\hat{\underline{\theta}}_2)$ $h_2(\underline{x}) = \hat{\alpha}_1 h(\underline{x};\hat{\underline{\theta}}_1) + \hat{\alpha}_2 h(\underline{x};\hat{\underline{\theta}}_2)$



$h(\underline{x};\hat{\underline{\theta}}_3)$ $h_3(\underline{x}) = \hat{\alpha}_1 h(\underline{x};\hat{\underline{\theta}}_1) + \cdots + \hat{\alpha}_3 h(\underline{x};\hat{\underline{\theta}}_3)$



$h(\underline{x};\hat{\underline{\theta}}_4)$ $h_4(\underline{x}) = \hat{\alpha}_1 h(\underline{x};\hat{\underline{\theta}}_1) + \cdots + \hat{\alpha}_4 h(\underline{x};\hat{\underline{\theta}}_4)$



Question 43. How can we train an ensemble of neural networks?

- Start from different initializations
- Train on slightly different data
- Average models after training(?)
- Prune weights after training to get multiple models
- many more tricks...

But we do none of these tricks in practice, why?

> Dropout is a clever way of constructing a really large ensemble. There is very heavy weight sharing: members of the ensemble are essentially the same as each other and pull individuals towards each other during training. Clever test-time computation is the trick that enables using such an ensemble.

Consider a single-layer hidden layer network performing regression for a datum (x, y)

$$y = a^\top \text{dropout}\left(\underbrace{\sigma(Wx)}_{\text{activations } h}\right).$$

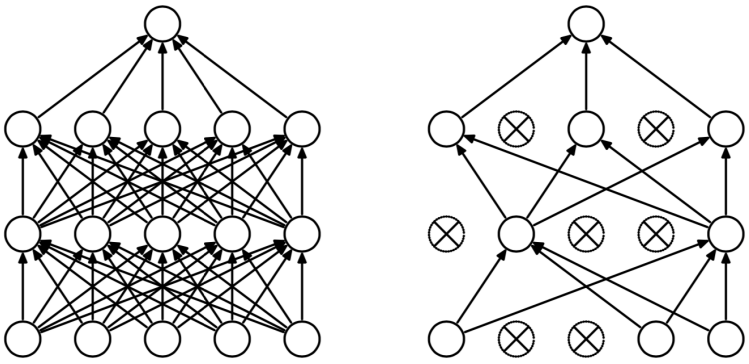
Dropout is an operation that sits in between two layers. It looks like

$$\text{dropout}_p(h) = h \odot r$$

where $r \in \{0,1\}^d$ is a binary mask. Each element of this mask r_k is a Bernoulli random variable Bernoulli(p), it is zero with probability p and one with probability $1-p$

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

A new mask is chosen for each datum that is forward-propped through the network.



(a) Standard Neural Net (b) After applying dropout.

You can change the parameter of the Bernoulli to be something else, in one of the problem

sets we used $p = 0.2$ right at the input layer.

Question 44. For a Bernoulli parameter $p = 0.5$ and $h \in \mathbb{R}^d$, at most how many single-hidden layer networks do we get in the above example?

Question 45. How does backprop work for the dropout layer?

Let us look at possibly the most important idea behind dropout: that is how to use dropout at test time. **The following is a very heuristic but beautiful argument**. Building upon the intuition from bagging and boosting, we would like to average the output of all the individual sparse models created during training. Consider the case where we have a model without any non-linearities

$$f(w; x, r) = \sum_{k=1}^{d} w_k \ (x_k \odot r_k)$$

which predicts the log-probability of belonging to a class given a mask, i.e., $f(w; x, r) = \log \mathbb{P}(x \text{ is a cat}, r)$. Each neuron corresponding to weights w_k sees its input zero-ed out with probability p during training. Equivalently, we can zero-out the weights of the neuron with probability p.

$$\sum_{k=1}^{d} w_k \ (x_k \odot r_k) = \sum_{k=1}^{d} (w_k \odot r_k) \ x_k.$$

At test time we would like to average over all possible masks r. We would like to compute

$$\begin{aligned}
&\log \mathbb{P}(\text{ensemble predicts } x \text{ to be a cat}) \\
&= \frac{1}{2^d} \sum_{i=1}^{2^d} \log \mathbb{P}(x \text{ is a cat}; r_i) \\
&= \frac{1}{2^d} \sum_{i=1}^{2^d} \sum_{k=1}^{d} (w_k \odot r_k) \ x_k \\
&= \sum_{k=1}^{d} \left(w_k \odot \frac{1}{2^d} \sum_{i=1}^{2^d} r_k \right) \ x_k \\
&= \sum_{k=1}^{d} (w_k \ (1-p)) \ x_k.
\end{aligned}$$

In other words, we can

- train with dropout of probability p
- scale all the weights by $1 - p$ at test-time to get the prediction of the ensemble.

This is powerful because we never have to compute the predictions of the exponentially large number of models explicitly at test time. Note that it is not true for multi-layer networks but we do it anyway.

Question 46. We wanted to create an ensemble of different models. If you observe closely, we are zero-ing out activations of the layer. Dropout seems more like a non-linearity when written like this. The weights of the layer W remain the same. Is this appropriate?

Remark 47 (Inverted Dropout)**.** It is cumbersome to change the model at test-time depending on how it was trained. E.g., after you train your model with dropout of $p = 0.25$, you would have to give the weights of the model to someone who deploys this model in real-life along with the probability $1 - p$ that they should be using to make predictions. We can avoid this hassle by *scaling down the weights* $1 - p$ *during training*. This way the person deploying the model at test-time can simply use the weights as is. This mechanism is called inverted dropout and is the one typically implemented in all deep learning libraries.

Remark 48 (Using dropout as a heuristic for getting confidence of the classifier)**.** What happens if we apply dropout at test time? We can for instance make multiple predictions on the same datum with slightly different models, each obtained using a different dropout mask. While we know from the previous computation that the average of these predictions is more or less the same as the prediction of the ensemble, we can get a heuristic estimate of the confidence of the classifier as

$$\text{Var}\left(f(w, x, r_1), f(w, x, r_2), \ldots, f(w, x, r_m)\right).$$

Is this the same as the variance in the variance in the bias-variance tradeoff

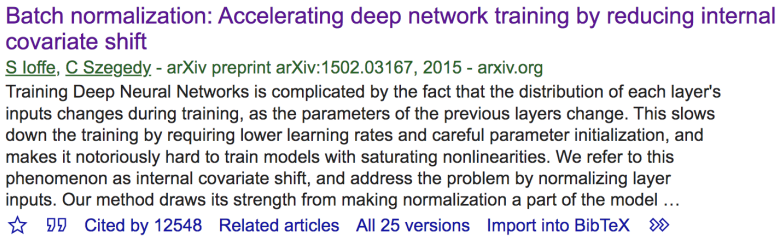
$$\text{Var}(h) = \mathop{\mathbb{E}}_{D}\left[\left(h(x, D) - \mathop{\mathbb{E}}_{D}\left[h(x, D)\right]\right)^2\right]?$$

7.2 Batch-normalization (BN)

This is a multi-faced beast.

- BN is one of the most fragile operations in the practice of deep learning. It is crucial to make things work in practice. After it has done its job, it will invariably come back to bite you.

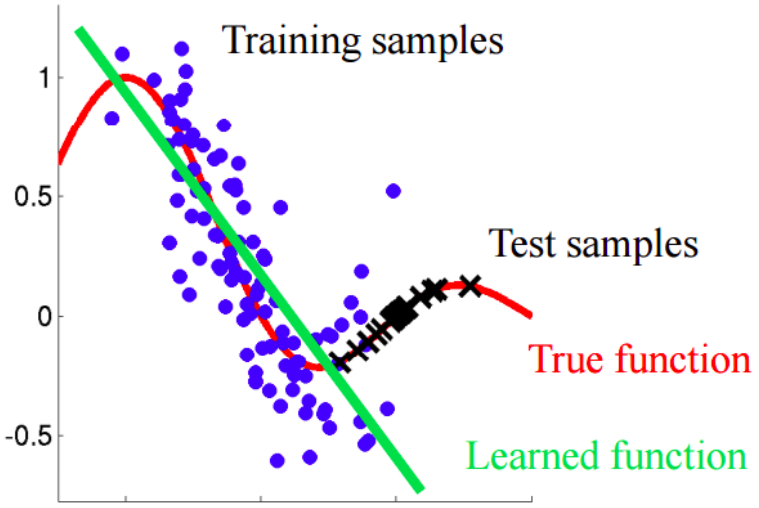
- Deep Learning is full of such mysterious operations. Bringing order in this mess is a great avenue for research.



7.2.1 What is 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, the shifted distribution of forces the classifier to work in a data regime that is different from what it was trained for.



Can we get an estimate of the robustness of a classifier to covarite shift by looking at the variance in the bias-variance trade-off? This is a subtle point: the expectation in the tradeoff is computed over all possible datasets which is impossible to compute in practice. This is the reason why algorithms to *tackle covariate shift* are popular as opposed to techniques for estimating covariate shift.

Remark 49 (Ways to mitigate covariate shift in standard machine learning?)**.** Here is a standard technique to fight covarite shift. Let us imagine that the training data D_{train} are drawn from the distribution $q(x)$. The test data as we said may not be drawn from the same distribution, say they come from some distribution $p(x)$. We will assume that the way the labels depend on the data does not change, i.e.,

$$q(y|x) = p(y|x).$$

This is reasonable because if a feature so and so was the one predicting cancer in one country, it is also potentially going to be a good predictor of cancer in another country.

A short derivation of the importance ratio for a finite dataset follows.

introduce the importance ratio or the propensity score β.

Question 50. All of this seems a bunch of complications brought about by the real world. If we stick to our standard setup where the training data and the test data come from the same distribution, there is no covariate shift. Then why does the BN paper have the words "internal covariate shift" in its title? The answer lies in the observation that deep networks are hierarchical. We can think of each layer as taking in data that is input activations and

predicting the output activations. As the network is being trained, the distribution of the activations also changes. This is the reason each layer suffers from covariate shift.

7.2.2 Recap of BN equations from the recitation

For $D_{\text{train}} = \{(x_i, y_i)\}_{i=1,\dots,n}$ where each $x_i \in \mathbb{R}^d$ one can do normalization on each dimension $k \in \{1, \dots, d\}$ as

$$\tilde{x}_i^k = \frac{x_i^k - \mathbb{E}_{i\in\{1,\dots,n\}}\left[x_i^k\right]}{\sqrt{\text{Var}_{i\in\{1,\dots,n\}}\left(x_i^k\right)}}.$$

Note that the mean and variance are computed over the entire dataset D_{train}. But the weights of the network change after each mini-batch update. It is cumbersome to evaluate the mean/variance over the entire dataset each time. The Batch-Normalization operation evaluates these quantities only over a mini-batch. Precisely, given a mini-batch of data $\{(x_i, y_i)\}_{i\in\mathcal{b}}$ where $\mathcal{b}$ denotes the indices of the images that we sampled in the particular mini-batch, the BN operation is

$$\tilde{x}_i^k = \frac{x_i^k - \mathbb{E}_{i\in\mathcal{b}}\left[x_i^k\right]}{\sqrt{\text{Var}_{i\in\mathcal{b}}\left(x_i^k\right) + \epsilon}}.$$

There is small $\epsilon > 0$ that is added to the denominator for numerical stability. Note that now the normalized data $\tilde{x}_i^k$ depend on the other images that form the mini-batch. This makes BN very sensitive to the size of the mini-batch: for large batch-sizes the statistics are close to those of the entire training set whereas for small batch-sizes the statistics are quite noisy. This is why it is often said that BN "adds noise" during training.

In pseudo-code, this goes as follows:

```
# t is the incoming tensor of shape [B, H, W, C]
# mean and stddev are computed along 0 axis and have shape [H, W, C]
mean = mean(t, axis=0)
stddev = stddev(t, axis=0)
for i in 0..B-1:
    out[i,:,:,:] = normalize(t[i,:,:,:], mean, stddev)
```

Typically, the BN layer also performs an affine operation upon the normalized inputs

$$W\tilde{x}_i^k + w_0.$$

Remark 51 (BN for convolutional layer)**.**

Question 52 (Should you add BN layer right after convolutions or do ReLU first?)**.** This is a subject of much debate. But the answer is not that hard. BN removes the effect of the bias that is added at each output channel of a convolutional layer because it normalizes everything else except the channel dimension. If we use BN without the affine transform, this is problematic because the different channels are forced to remain at the same bias, namely, zero. The affine transform of BN helps by adding this extra degree of freedom. In other words, it is okay to use BN before or after the non-linearity.

Remark 53 (BN makes the weights scale invariant)**.** Consider a linear layer given by

$$h_i = Wx_i;$$

as usual $x_i \in \mathbb{R}^d$, $W \in \mathbb{R}^{c\times d}$ and $h_i \in \mathbb{R}^c$. If all weights W are multiplied by a scalar λ the output changes to

$$h_i^\lambda = \lambda W x_i.$$

Notice that

$$\begin{aligned} h_i^\lambda &= \lambda h_i \\ \mathop{\mathbb{E}}_{i\in\mathcal{b}}\left[h_i^\lambda\right] &= \lambda \mathop{\mathbb{E}}_{i\in\mathcal{b}}\left[h_i\right] \\ \mathop{\mathrm{Var}}_{i\in\mathcal{b}}\left(h_i^\lambda\right) &= \lambda^2 \mathop{\mathrm{Var}}_{i\in\mathcal{b}}\left(h_i\right). \end{aligned}$$

In other words, disregarding the effect of the ϵ in the denominator the output of batch norm remains the same:

$$\text{batch-norm}(\lambda W x_i) = \text{batch-norm}(W x_i).$$

It is almost as if we did not scale the weights at all. Note that this property does not depend on the batch-size. This property is also true for the convolutional layers exactly because we normalized on all axes except the channels.

It is reasonable to expect that the scale invariance of the weights is a good thing. We have

introduced a symmetry in the space of the hypothesis class and have therefore *regularized*.

Question 54 ("BN stabilizes the growth of the parameter scale" - BN paper)**.** Consider the following

$$\begin{aligned}\frac{\partial \text{batch-norm}(\lambda W x)}{\partial(\lambda W)} &= \frac{\partial \text{batch-norm}(Wx)}{\partial(\lambda W)} \\ &= \frac{1}{\lambda}\frac{\partial \text{batch-norm}(Wx)}{\partial W}.\end{aligned}$$

You can do the same derivation with our $\overline{(\cdot)}$ from the lecture to see that the same is true for the backprop gradients $\overline{W}$. In other words, if an SGD update caused the scale of weights to increase on one layer, BN scales down the gradient in the next iteration leading to an effectively smaller learning rate and decreased rate of scale explosion.

Question 55 (BN versus weight-decay: fight, fight, fight)**.** We saw that BN scales down the gradients. But this is not the end of the story. The scale invariance in the weight space caused by BN acts fundamentally against what weight decay tries to do: the latter wants parameters to stay close to the origin while the former is happy to scale them up, or down, without changing the loss. Further, thanks to the scale invariance of BN, the ℓ_2 penalty on the weights

$$\|w\|^2$$

can be made arbitrarily small by the network. Effectively, using BN on a layer disables weight decay for the parameters of that layer. This seems like a bad thing to do. Let us dig deeper into this problem.

You will show in the following problem set that

$$v^{(t+1)} = v^{(t)} - \frac{\eta}{\|w^{(t)}\|^2}\left(I - v^{(t)}{v^{(t)}}^\top\right)\nabla\ell(v^{(t)}) + \mathcal{O}(\eta^2)$$

where $v = w/\|w\|$ if the loss $\ell(w)$ is invariant to the scale of w. This indicates that reducing the norm of the weights $\|w^{(t)}\|^2$ leads to an effective increase in the learning rate for gradient descent. This is quite intuitive.

7.2.3 Some tips to work with BN

1. BN statistics are implemented in tricky ways in the libraries.
2. BN statistics change enormously during training. Often 10-20 weight updates are enough to change them for small batch-sizes.

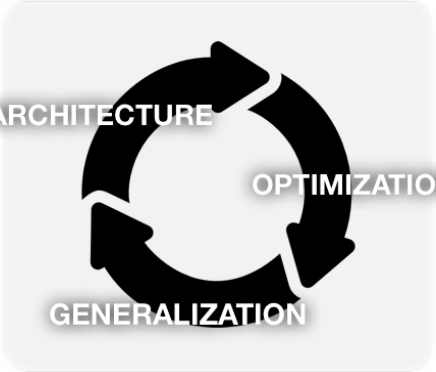
7.3 Stochastic convex optimization

Lecture 8

Gradient descent, stochastic gradient descent and accelerating techniques

We have covered the cliff-notes of the practice of deep learning in the previous 6 lectures. It is by no means a complete overview. The practice of deep learning is an enticing, mysterious, and sometimes frustrating art. 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.

The next few lectures will have a very different flavor. We will forget deep networks and understand some basic ideas in optimization. Depending on your personal style, some of you might find this material utterly drab, uber-exciting, or anything in between. But here is why it is important.



There are three main concepts in machine learning. First, the hypothesis class that you search for classifiers in, this is the “architecture in the following picture”. Second, the algorithm

you use to perform this search. This would have ideas from optimization theory which is all about doing the best job one can on minimizing the empirical risk. Third is the generalization performance of your classifier which is about getting a good number for the population risk. We work on machine learning to make our classifiers generalize.

All this is well and good for simple models like SVMs. In your previous class, you picked a kernel... The story is quite murky for deep networks. It is never clear how you should change the architecture, the optimization algorithm, the numerous little bells and whistles like regularization techniques. If your model generalizes well, you are done and happy. If it does not, it is hard to find one part of the pipeline that gets the blame.

Disentangling this viscious cycle is what "understanding deep learning" is all about. We have covered the first part, namely architecture. We are now moving to the second one: optimization.

8.1 Convexity

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

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

I remember this (Jensen's inequality) as "square of averages is smaller than average of the squares". (Draw a picture). A function $f(x)$ is called concave if $-f(x)$ is convex.

There are many species of convex functions:

1. Strictly convex functions where we have that for all $x \neq y$ and $\lambda \in (0, 1)$

$$f(\lambda x + (1 - \lambda)y) < \lambda f(x) + (1 - \lambda)f(y).$$

2. Strongly convex functions which have the property that there exists an $\alpha > 0$ such that

$$f(x) - \alpha\|x\|^2 \text{ is convex.}$$

It is easy to see that strict convexity implies convexity: the former is more restrictive than the latter. Let us show quickly that strong convexity implies strict convexity. If a function $f(x)$ is strongly convex we know that there exists an $\alpha > 0$ such that $f(x) - \alpha\|x\|^2$ is convex. Apply the definition of convexity to this new function now

$$f(\lambda x + (1-\lambda)y) - \alpha\|\lambda x + (1-\lambda)y\|^2 \leq \lambda f(x) - \lambda\alpha\|x\|^2 + (1-\lambda)f(y) - (1-\lambda)\alpha\|y\|^2.$$

But

$$\lambda\alpha\|x\|^2 + (1-\lambda)\alpha\|y\|^2 - \alpha\|\lambda x + (1-\lambda)y\|^2 > 0$$

for $\lambda in (0,1)$ for all $x \neq y$ because $\|x\|^2$ is strictly convex.

In other words, we have

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

Strongly convex functions are easier to optimize for algorithms. It will also always be easier to prove a result in optimization on strongly convex functions.

If $f(x)$ is convex and differentiable, as you saw in the problem set, we can write

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle .$$

For strictly convex functions this inequality will look like

$$f(y) > f(x) + \langle \nabla f(x), y - x \rangle$$

Question 56. What do we do for non-differentiable functions? The gradient at the point of non-differentiability is not defined. If we look at the property that gradients always lie below convex functions, it is easy to see that there are lots of vectors that satisfy the bill. In this case, we define what is called a sub-gradient. A vector v is a sub-gradient at x iff

$$f(y) \geq f(x) + \langle v, y - x \rangle .$$

The sub-differential for a function f at x is defined to be

$$\partial f(x) := \left\{ v \in \mathbb{R}^d : \ v \text{ is a sub-gradient of } f \text{ at } x \right\}.$$

Note that the sub-differential is a set and it is has only one element in it if $f(x)$ is differentiable. It has many elements in it if f is not differentiable at x. The sub-differential set is never empty for convex functions.

8.1.1 Some typical assumptions in convex optimization

1. **Lipschitz continuity/Bounded gradients (B).**

$$|f(x) - f(y)|_2 \leq B\|x - y\|_2$$

which is how you will see it written in most papers/books. This is equivalent to assuming that gradients of $f(x)$ are uniformly bounded on compact sets (say, $\|x\| \leq D$)

$$\|\nabla f(x)\| \leq B.$$

2. **Smoothness/Lipschitz continuity of gradients (L)** We like functions that are smooth, i.e., the function $f(x)$ is differentiable and its gradient is also Lipschitz continuous

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

If $f(x)$ is twice differntiable, this is equivalent to assuming that

$$\nabla^2 f(x) \preceq LI.$$

3. **Strong convexity (μ)** The function $f(x)$ is convex and differentiable. Sometimes we may even assume strong convexity. For differentiable functions you can re-write the definition of strong convexity to look as

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2} \|x - y\|^2.$$

Note again that if $f(x)$ is twice differentiable, this is equivalent to assuming

$$\nabla^2 f(x) \succeq \mu I.$$

4. **Co-coercivity of the gradient**. If the gradient is L-Lipschitz, we have

$$\langle \nabla f(x) - \nabla f(y), x - y \rangle \geq \frac{1}{L} \|\nabla f(x) - \nabla f(y)\|^2.$$

8.2 Gradient descent

Let us now think of solving the following problem

$$x^* = \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x) \tag{8.1}$$

for a given function $f(x)$. A point x^* is a local minimum for a function $f(x)$ if $f(x^*) \leq f(x)$ for all x in a neighborhood of x^*. It is a global minimum of $f(x)$ if $f(x^*) \leq f(x)$ for all x in the domain.

Lemma 57. *For convex functions, local minima are global minima.*

Proof. We can do an argument by contradiction. If x^* is a local minimum of $f(x)$ that is not a global minimum, there exists a point y such that $f(y) < f(x^*)$. The domain of $f(x)$ is convex, so pick a point

$$z = \lambda y + (1 - \lambda)x^*$$

to see that

$$f(z) - f(x^*) \leq \lambda \left(f(y) - f(x^*) \right).$$

Since x^* is a local minimizer, we can now make λ small enough to get that the left hand side is non-negtive

$$f(y) \geq f(x^*).$$

But this means that x^* is a global minimum. ■

A similar result that can be proved using the definition of continuity and the previous result.

Lemma 58. *If x^* is a local minimum of a continuously differentiable function f it satisfies first-order optimality condition, i.e.,*

$$\nabla f(x^*) = 0.$$

If $f(x)$ is convex, then $\nabla f(x^*) = 0$ is a sufficient condition for global optimality.

For an iterative scheme

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

optimization typically concerns itself with two notions of convergence.

1. Convergence in function value

$$f(x^{(t)}) - f(x^*)$$

2. Convergence of the actual iterates

$$\|x^{(t)} - x^*\|^2.$$

(picture of a contour plot of a function f. mention picking the step-size is important, descent direction)

So here is one iterative algorithm. It updates the weights as

$$x^{(t+1)} = x^{(t)} + \alpha_t \, d^{\,(t)} \tag{8.2}$$

where α_t is the step-size (also called the learning rate in the machine learning literature) and $d^{\,(t)}$ is some descent direction that satisfies

$$\left\langle \nabla f(x^{(t)}), d^{\,(t)} \right\rangle < 0.$$

Question 59 (How do we pick α_t and $d^{\,(t)}$?)**.** There are numerous ways to do so and this gives a family of descent algorithms. Typically we at least desire monotonic progress, i.e.,

$$f(x^{(t+1)}) < f(x^{(t)}).$$

Choices for $d^{\,(t)}$:

1. Gradient descent (steepest descent) picks

$$d^{\,(t)} = -\nabla f(x^{(t)}).$$

2. Newton's method picks

$$d^{\,(t)} = -\left[\nabla^2 f(x^{(t)})\right]^{-1} \nabla f(x^{(t)}).$$

3. The descent direction at time t does not always have to depend only on $x^{(t)}$, it can also depend on the past iterates.

Choosing α_t: Say you picked a $d^{\,(t)}$ what is the best step-size to use? It is something that makes the best progress at this iteration, i.e.,

$$\alpha_t = \underset{\alpha \geq 0}{\text{argmin}} \; f(x^{(t)} + \alpha d^{\,(t)}).$$

(back to the picture of the contour plot).

Question 60. Can anyone tell me an algorithm for optimization that does not look like gradient descent?

8.2.1 Convergence of gradient descent

Lemma 61 (Descent lemma)**.** *For an L-smooth function $f(x)$*

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2.$$

we have

$$f(y) \leq f(x) + \langle \nabla f(x), y - x\rangle + \frac{L}{2}\|x - y\|^2. \tag{8.3}$$

Compare this with

$$f(y) \geq f(x) + \langle \nabla f(x), y - x\rangle$$

that we saw for convex functions.

Proof. We use Taylor's theorem with some direction $z_t = x + t(y - x)$ to get

$$f(y) = f(x) + \int_0^1 \langle \nabla f(z_t), y - x\rangle \ \mathrm{d}t.$$

Add and subtract $\langle \nabla f(x), y - x\rangle$ on the right hand size to get

$$\begin{aligned}
f(y) - f(x) &= \int_0^1 \langle \nabla f(z_t), y - x\rangle + \langle \nabla f(x), y - x\rangle - \langle \nabla f(x), y - x\rangle \ \mathrm{d}t \\
f(y) - f(x) - \langle \nabla f(x), y - x\rangle &= \int_0^1 \langle \nabla f(z_t) - \nabla f(x), y - x\rangle \ \mathrm{d}t \\
|f(y) - f(x) - \langle \nabla f(x), y - x\rangle| &= |\int_0^1 \langle \nabla f(z_t) - \nabla f(x), y - x\rangle \ \mathrm{d}t| \\
|f(y) - f(x) - \langle \nabla f(x), y - x\rangle| &\leq \int_0^1 |\langle \nabla f(z_t) - \nabla f(x), y - x\rangle| \mathrm{d}t \\
&\leq \int_0^1 \|\nabla f(z_t) - \nabla f(x)\| \|y - x\| \mathrm{d}t \\
&\leq L \int_0^1 t\|y - x\|^2 \mathrm{d}t \\
&= \frac{L}{2}\|x - y\|^2.
\end{aligned}$$

This bounds $f(y)$ around x with room for quadratic variations. ■

Lecture 9

Gradient descent (cont.), Nesterov's acceleration and Lyapunov functions

9.1 Convergence of gradient descent

Lemma 62 (Descent lemma)**.** *For an L-smooth function $f(x)$*

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2.$$

we have

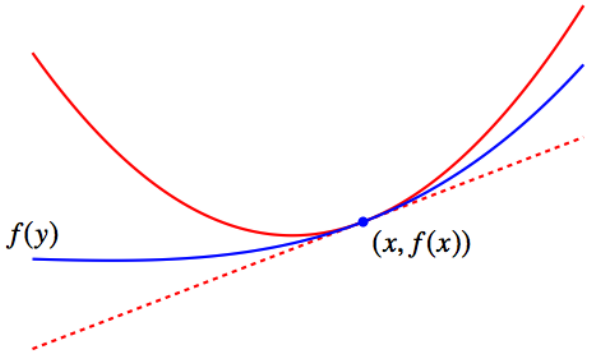
$$f(y) \leq f(x) + \langle \nabla f(x), y - x\rangle + \frac{L}{2}\|x - y\|^2. \tag{9.1}$$

Compare this with

$$f(y) \geq f(x) + \langle \nabla f(x), y - x\rangle$$

that we saw for convex functions.

Proof. Was done in the previous lecture. ■



Corollary 63. *A consequence of the descent lemma is that*

$$\frac{1}{2L}\|\nabla f(z)\|^2 \leq f(z) - f(x^*) \leq \frac{L}{2}\|z - x^*\|^2$$

for all z.

Proof. The right hand side follows from (9.1) directly by substituting $x = x^*$ and $y = z$. The left hand side follows by minimizing the quadratic upper bound in (9.1). Substitute $x = z$ and

$$\begin{aligned}
\inf_y f(y) &\leq \inf_y \left(f(z) + \langle \nabla f(z), y - z \rangle + \frac{L}{2}\|z - y\|^2 \right) \\
&= \inf_{\|v\|=1} \inf_t \left(f(z) + t \langle \nabla f(z), v \rangle + \frac{L}{2} t^2 \right) \\
&= \inf_{\|v\|=1} \left(f(z) - \frac{1}{2L} \left(\nabla f(z)^\top v \right)^2 \right) \\
&= f(z) - \frac{1}{2L}\|\nabla f(z)\|^2.
\end{aligned}$$

■

Corollary 64. *Substitute* $y = x - \alpha \nabla f(x)$ *in the descent lemma to show that if*

$$\alpha \leq \frac{1}{L}$$

then we get

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{1}{2\alpha}\|x - y\|^2. \tag{9.2}$$

We have actually just shown that steepest descent makes monotonic progress is the step-size

is small enough. Observe that we can set $x := x^{(t)}$ in the above corollary to get that for

$$x^{(t+1)} := x^{(t)} - \alpha \nabla (f(x^{(t)}))$$

we have

$$f(x^{(t+1)}) \leq f(x^{(t)}) - \frac{\alpha}{2} \|\nabla f(x^{(t)})\|^2. \tag{9.3}$$

This also explains why gradient descent can diverge in practice if the step-size α is too large.

(Picture of divergence of gradient descent)

Let us again use the definition of convexity

$$f(x^*) \geq f(x) + \langle \nabla f(x), x^* - x \rangle \Rightarrow f(x) \quad \leq f(x^*) + \langle \nabla f(x), x - x^* \rangle .$$

Substitute this in (9.3) to get

$$\begin{aligned} f(x^{(t+1)}) - f(x^*) &\leq \left\langle \nabla f(x^{(t)}), x^{(t)} - x^* \right\rangle - \frac{\alpha}{2} \|\nabla f(x^{(t)})\|^2 \\ &= \frac{1}{2\alpha} \left(\|x^{(t)} - x^*\|^2 - \|x^{(t)} - x^* - \alpha \nabla f(x^{(t)})\|^2 \right) \\ &= \frac{1}{2\alpha} \left(\|x^{(t)} - x^*\|^2 - \|x^{(t+1)} - x^*\|^2 \right). \end{aligned} \tag{9.4}$$

This shows that during gradient descent, if $\nabla f(x^{(t)}) \neq 0$, the loss decreases

$$f(x^{(t+1)}) < f(x^{(t)})$$

and the distance to the optimum also decreases

$$\|x^{(t+1)} - x^*\|^2 < \|x^{(t)} - x^*\|^2.$$

Lemma 65 (Convergence rate for gradient descent with constant step-size)**.** *The inequality in (9.4) gives*

$$f(x^{(t)}) - f(x^*) \leq \frac{1}{2t\alpha} \|x^0 - x^*\|^2. \tag{9.5}$$

Proof. Sum up the inequalities in (9.4) and use the fact that the sequence decreases, so the last iterate $f(x^{(t+1)})$ is smaller than the average. ■

The above lemma indicates that number of iterations to reach a certain accuracy $f(x^{(t)}) - f(x^*) \leq \epsilon$ is $\mathcal{O}(1/\epsilon)$. This is known as a sub-linear convergence rate in the optimization literature, longer you run gradient descent slower is its progress.

This is quite bad, but notice that we never used strong convexity of the function $f(x)$ to get this rate. So $\mathcal{O}(1/\epsilon)$ is the convergence rate for convex functions with Lipschitz-continuous gradients and fixed step-size.

Remark 66. In the optimization literature a way to measure the rate of convergence is by using the limit of the successive errors. An sequence $f(x^1), \ldots, f(x^{(t)}), f(x^{(t+1)}), \ldots$ has a convergence rate ρ if

$$\lim_{t \to \infty} \frac{f(x^{(t+1)}) - f(x^*)}{f(x^{(t)}) - f(x^*)} = \rho.$$

If

- $\rho = 1$ we call it a sub-linear rate
- $\rho \in (0, 1)$ we call it a linear rate
- $\rho = 0$ we call it a superlinear rate.

For the expression in (9.5) we have

$$\lim_{t\to\infty} \frac{f(x^{(t+1)}) - f(x^*)}{f(x^{(t)}) - f(x^*)} \leq \frac{t}{t+1}$$

9.1.1 Gradient descent for strongly-convex functions

Lemma 67. *For a m-strongly convex and L-smooth function f, we have*

$$\langle \nabla f(x) - \nabla f(y), x - y\rangle \geq \frac{mL}{m+L}\|x-y\|^2 + \frac{1}{m+L}\|\nabla f(x) - \nabla f(y)\|^2$$

for all x, y.

Proof. We will use the co-coercivity condition for the convex function $h(x) := f(x) - \frac{m}{2}\|x\|^2$. First we show that the function $h(x)$ is $L - m$ smooth; it is of course convex from the definition of strong convexity. Note that for a convex function we have from the descent lemma that

$$f(y) \leq f(x) + \langle \nabla f(x), y - x\rangle + \frac{L}{2}\|x-y\|^2.$$

We can also create a lower bound for $f(y)$ for smooth convex functions:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x\rangle + \frac{1}{2L}\|\nabla f(x) - \nabla f(y)\|^2.$$

To prove this, set some $z = y - \frac{1}{L}\left(\nabla f(x) - \nabla f(y)\right)$ and write

$$\begin{aligned} f(x) - f(y) &= f(x) - f(z) + f(z) - f(y) \\ &\leq \langle \nabla f(x), x - z\rangle + \langle \nabla f(y), z - y\rangle + \frac{L}{2}\|z-y\|^2 \\ &= \langle \nabla f(x), x - y\rangle + \langle \nabla f(x) - \nabla f(y), y - z\rangle + \frac{1}{2L}\|\nabla f(x) - \nabla f(y)\|^2 \\ &= \langle \nabla f(x), x - y\rangle - \|\nabla f(x) - \nabla f(y)\|^2. \end{aligned}$$

We can add the two to get a very useful if and only if condition for smooth convex functions:

$$\frac{1}{L}\|\nabla f(x) - \nabla f(y)\|^2 \leq \langle \nabla f(y) - \nabla f(x), y - x\rangle \leq L\|x-y\|^2$$

for any smooth convex function $f(x)$.

$$\begin{aligned}\langle \nabla h(x) - \nabla h(y), x - y\rangle &= \langle \nabla f(x) - \nabla f(y), x - y\rangle - m\|x-y\|_2^2 \\ &\leq (L-m)\|x-y\|_2^2.\end{aligned}$$

Now we use the co-coercivity condition for $h(x)$ which is $L-m$ smooth and convex.

$$\begin{aligned}\langle \nabla h(x) - \nabla h(y), x - y\rangle &\geq \frac{1}{L-m}\|\nabla h(x) - \nabla h(y)\|^2 \\ \langle \nabla f(x) - \nabla f(y), x - y\rangle - m\|x-y\|^2 &\geq \frac{1}{L-m}\left(\|\nabla f(x) - \nabla f(y)\|^2 + m^2\|x-y\|^2\right) \\ &\quad - \frac{2m}{L-m}\langle \nabla f(x) - \nabla f(y), x - y\rangle \\ \langle \nabla f(x) - \nabla f(y), x - y\rangle \left(1 + \frac{2m}{L-m}\right) &\geq \frac{\|\nabla f(x) - \nabla f(y)\|^2}{L-m} + \left(\frac{m^2}{L-m} + m\right)\|x-y\|^2 \\ \langle \nabla f(x) - \nabla f(y), x - y\rangle &\geq \frac{\|\nabla f(x) - \nabla f(y)\|^2}{L+m} + \frac{mL}{m+L}\|x-y\|^2.\end{aligned}$$

■

Theorem 68. *The convergence rate of gradient descent for*

$$\alpha \leq \frac{2}{m+L}$$

for m-strongly convex function f with L-Lipschitz gradients is

$$\|x^{(t)} - x^*\| \leq c^t \|x^0 - x^*\|$$

where

$$c = 1 - \alpha\left(\frac{2mL}{m+L}\right).$$

This implies a linear convergence rate (the plot is linear on a semi-log scale). If we pick the largest allowed step-size

$$\alpha = \frac{2}{m+L},$$

we have

$$c = \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 \quad \text{where } \kappa = L/m$$

Proof.

$$\begin{aligned}
\|x^{(t+1)} - x^*\|^2 &= \|x^{(t+1)} - \alpha\nabla f(x^{(t)}) - x^*\|^2 \\
&= \|x^{(t+1)} - x^*\|^2 - 2\alpha\nabla f(x^{(t)})^\top (x^{(t)} - x^*) + \alpha^2\|\nabla f(x^{(t)})\|^2 \\
&\leq \left(1 - \frac{2mL\alpha}{m+L}\right)\|x^{(t)} - x^*\|^2 + \alpha\left(\alpha - \frac{2}{m+L}\right)\|\nabla f(x^{(t)})\|^2 \\
&\leq \underbrace{\left(1 - \frac{2\alpha mL}{m+L}\right)}_{c}\|x^{(t)} - x^*\|^2.
\end{aligned}$$

If the step-size α is such that quantity in the round brackets is less than 1, this is a contraction. Iterate upon this to get

$$\|x^{(t)} - x^*\| \leq c^t\|x^0 - x^*\|. \tag{9.6}$$

■

Remark 69. Notice that

1. the largest step-size α is limited by $m + L$ by the convergence rate only depends on $\kappa = L/m$. Smaller the value of c faster the convergence (c does not affect the rate, only the constant in front of the rate).

2. if the function is not very smooth, i.e., m is small, c is close to 1 and the iterates converge slowly. Thus increasing the value of m for your function is a good trick to accelerate gradient descent. We will see such algorithms very soon.

3. larger the L smaller the ideal step-size α.

4. you will also see the above convergence rate written many times in papers as

$$\|x^{(t)} - x^*\| \leq e^{-4t/\kappa}\,\|x^{(0)} - x^*\|.$$

   You can get this by using the inequality $1 + x \leq e^x$ for any x and $t > 0$ from (9.6).

Corollary 70. *Theorem 68 gives a bound on the convergence of the function value as well by plugging in the quadratic upper bound of Corollary 63*

$$f(x^{(t)}) - f(x^*) \leq \frac{L}{2}\|x^{(t)} - x^*\|^2 \leq \frac{c^t L}{2}\|x^0 - x^*\|^2.$$

Note that the number of iterations to reach an accuracy $f(x^{(t)}) - f(x^*) \leq \epsilon$ is now $\mathcal{O}(\log(1/\epsilon))$. You can use part (iii) in the previous remark to see that gradient descent for

strongly convex functions converges at a rate

$$\mathcal{O}(\kappa \log(1/\epsilon));$$

we have simply pulled out the condition number κ to understand how it affects the rate.

9.2 Acceleration of gradient descent

We defined descent methods to look like any iteration of the form

$$x^{(t+1)} = x^{(t)} + \alpha_t d^{(t)}$$

where the descent direction $d^{(t)}$ is really our choice. For explicit first-order methods, we can pick $d^{(t)}$ to be anything such that $x^{(t+1)}$ is in the set

$$x^{(0)} + \text{span}\left\{\nabla f(x^{(0)}), \nabla f(x^{(1)}), \ldots, \nabla f(x^{(t-1)})\right\}.$$

Is the sub-linear convergence rate of gradient descent for convex functions optimal? The answer to this question comes from a famous theorem.

Theorem 71 (Nemirovski & Yudin '83)**.** *For every integer $t \leq (d-1)/2$ and every x^0 there exist convex functions that are differentiable with L-Lipschitz gradients have a minimizer x^* that is finite and lies within the domain, such that*

$$\min_{1 \leq s \leq t} f(x^{(s)}) - f(x^*) \geq \frac{3}{32} \frac{L\|x^{(t)} - x^*\|^2}{(t+1)^2}.$$

This is a lower bound on the convergence rate. It shows that the convergence rate of gradient descent for (non-strongly) convex functions $\mathcal{O}(1/\epsilon)$ that we showed is not optimal. The optimal rate should be $\mathcal{O}(1/\sqrt{\epsilon})$ instead. This is also true for the strongly-convex case, the convergence rate we obtained was $\mathcal{O}(\kappa \log(1/\epsilon))$ while a similar theorem by Nemirovski & Yudin gives the lower bound to be $\mathcal{O}(\sqrt{\kappa}\log(1/\epsilon))$. These lower bounds are based on a clever construction of the function $f(x)$.

Consider an example with a quadratic loss

$$f(x) = \frac{1}{2}x^\top A x$$

and see that the condition number $\kappa = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$. Problems with large condition numbers converge slower. Intuitively, these are problems that are highly curved in some dimensions and quite flat in others but since gradient descent does not scale the dimensions individually, it is forced to pick a step-size that works for all dimensions.

(Picture of this phenomenon)

Lecture 10

Nesterov's acceleration, Lyapunov functions and gradient flows

Reading

- "A differential equation for modeling Nesterov's accelerated gradient method: Theory and insights" by Su et al. (2014); read the shorter version at https://statweb.stanford.edu/ candes/papers/NIPS2014.pdf.

10.1 Acceleration methods

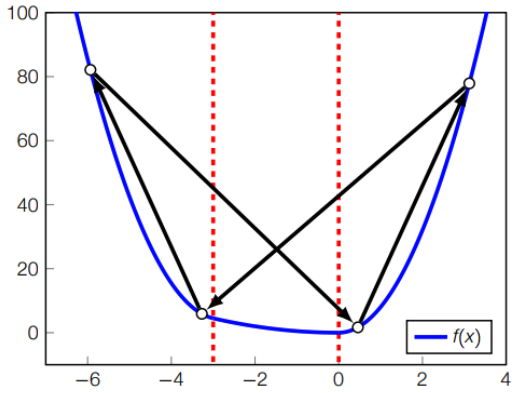
Remark 72 (Polyak's Heavy Ball method)**.** This is a simple modification of gradient descent and looks like

$$x^{(t+1)} = x^{(t)} - \alpha \nabla f(x^{(t)}) + \rho \left(x^{(t)} - x^{(t-1)} \right).$$

The term $(x^{(t)} - x^{(t-1)})$ is referred to as momentum. The intuition is that if the current gradient is in the same direction as the previous gradient we move a bit further in this direction. If the two gradients are misaligned, we move less far. We can rewrite this update into two updates

$$\begin{aligned} x^{(t+1)} &= y^{(t)} - \alpha \nabla f(x^{(t)}) \\ y^{(t)} &= (1+\rho)\, x^{(t)} - \rho x^{(t-1)} \end{aligned} \tag{10.1}$$

Polyak's method can fail to converge for certain loss functions, the iterates of this algorithm can go into a limit cycle depending on the initial conditions, e.g.,



However, it is a very simple technique to accelerate gradient descent and does work in practice.

10.1.1 Nesterov's acceleration for smooth, strongly-convex case

This algorithm maintains two copies of the variable $x^{(t)}$ and updates them alternately as follows:

$$\begin{aligned} x^{(t+1)} &= y^{(t)} - \alpha \nabla f(y^{(t)}) \\ y^{(t)} &= (1+\rho)\, x^{(t)} - \rho x^{(t-1)} \end{aligned} \tag{10.2}$$

where

$$\rho = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$$

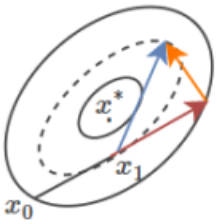
and $\alpha \leq \frac{1}{L}$ as usual. The idea is that we take Polyak's Heavy Ball method and first make the move using the momentum and then update using the gradient at the new location.

Remark 73. The above iterates of Nesterov's method are for strongly-convex functions. For smooth functions, they become

$$\begin{aligned} x^{(t+1)} &= y^{(t)} - \alpha \nabla f(y^{(t)}) \\ y^{(t)} &= x^{(t)} + \frac{t-1}{t+2}\left(x^{(t)} - x^{(t-1)}\right). \end{aligned} \tag{10.3}$$

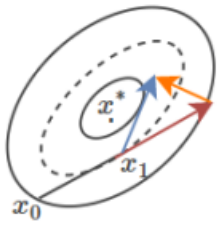
The fraction $\frac{t-1}{t+2} \approx 1 - \frac{3}{t}$ will turn out to be super interesting very soon.

Polyak's Momentum



$$x_{t+1} = x_t - \alpha \nabla f(x_t) + \mu(x_t - x_{t-1})$$

Nesterov Momentum



$$x_{t+1} = x_t + \mu(x_t - x_{t-1}) - \gamma \nabla f(x_t + \mu(x_t - x_{t-1}))$$

Both Nesterov's method and Polyak's Heavy Ball method achieve optimal convergence rates for gradient descent, i.e., this rate matches the lower bounds of Nemirovski & Yudin

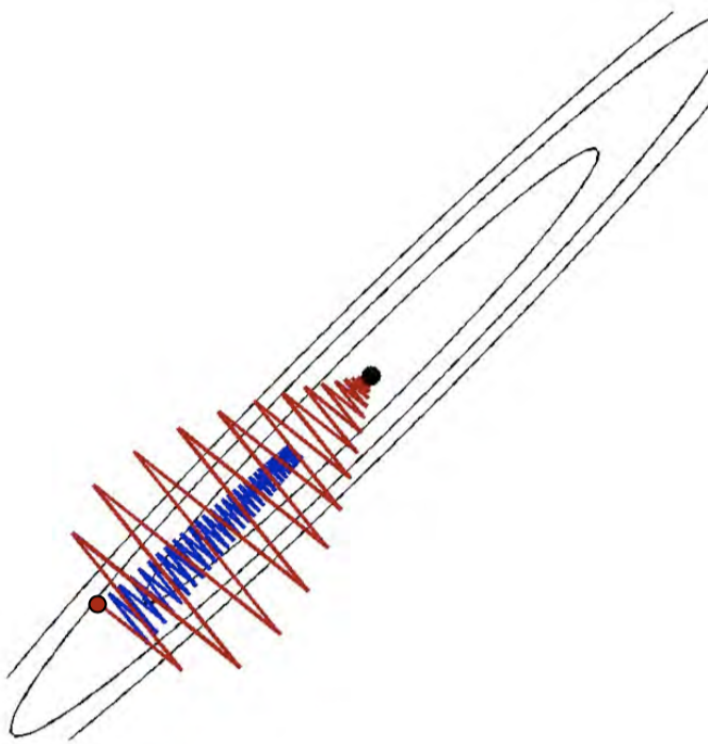
$$\|x^{(t)} - x^*\| \leq \mathcal{O}(\sqrt{\kappa} \log(1/\epsilon)).$$

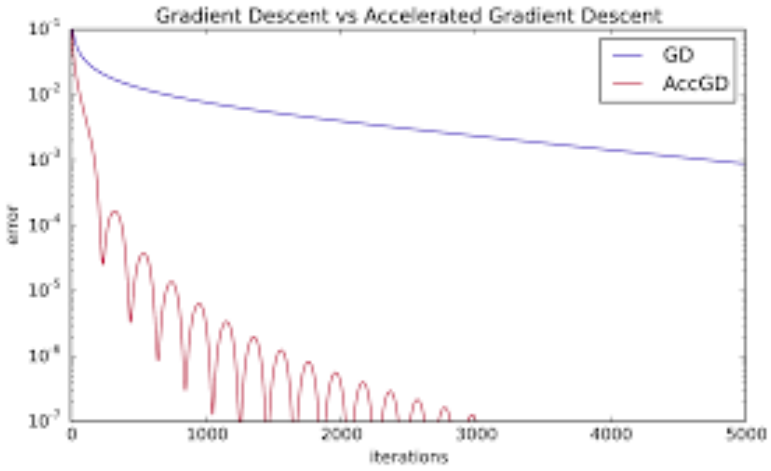
for smooth and strongly-convex functions and

$$\|x^{(t)} - x^*\| \leq \mathcal{O}(1/\sqrt{\epsilon})$$

for smooth convex functions. The proofs are quite tedious and we will not do them in this case. However, the acceleration provided by these techniques is quite important in practice.

Why/how does Nesterov's method work?





Question 74. What can we do to kill these oscillations?

Remark 75 (Summary of all the convergence rates we've seen)**.**

10.2 Gradient flows and gradient descent

We did not go through the proofs for obtaining convergence rates because they are tedious. Gradient descent is only one kind of an optimization algorithm and there are many others, e.g., mirror descent, cubic-regularized Newton's method, proximal gradient descent etc. Acceleration is a general technique that works to speed up all these algorithms but it is cumbersome to prove the convergence rate each and every time.

Let us look at a simple technique that not only gives us convergence rates quickly but also puts numerous methods descent methods. This is a technique from control theory known as Lyapunov functions. It is used to decide whether all trajectories of iterative updates go to the origin.

Remark 76. Interlude on continuous vs. discrete time. Can view algorithms in discrete time as discretizations of flows in continuous time.

Theorem 77 (Lyapunov's global stability theorem in continuous time)**.** *Consider a differential equation*

$$\dot{x} = -\nabla f(x) \quad x(0) = x^{(0)}$$

If there exists a continuous function $V : \mathbb{R}^d \to \mathbb{R}$ (called a Lyapunov function) that satisfies

(i) (positivity) $V(x^*) = 0$, $V(x) > 0$ for all $x \neq x^*$,

(ii) (radial unboundedness) $V(x) \to \infty$ as $\|x\| \to \infty$, and

(iii) (strict decrease) $\frac{dV}{dt} = \dot{V}(x) = \langle \nabla V(x), -\nabla f(x) \rangle < 0$ for all $x \neq 0$, then

for all x^0 we have $x(t) \to x^*$ as $t \to \infty$.

Remark 78. Notice that we have written $x(t) \to x^*$ instead of $x^{(t)} \to x^*$ in the above theorem. The distinction is pedantic but it helps remember that in the continuous-time analysis we are making statements about a continuous trajectory $x(t)$ and not just a discrete sequence of iterates $x^{(t)}$; the two are not the same.

Example 79 (Prove convergence to global minimum)**.** Pick the Lyapunov function to simply be $f(x(t))$ and see that $x(t) \to x^*$. In addition to this, the Lyapunov function always decreases along the trajectories of the dynamics. You can also use the Lyapunov function as $\|x(t) - x^*\|$ along with the co-coercivity condition.

Theorem 80. *For convex f, gradient flow $\dot{x} = -\nabla f(x)$ has the convergence rate*

$$f(x(t)) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|^2}{t} = \mathcal{O}(1/t).$$

Proof. Consider an energy function to be

$$\mathcal{E}(t) = \mathcal{E}(x(t)) := \frac{1}{f(x(t)) - f(x^*)}.$$

Under the gradient flow dynamics $\dot{x} = -\nabla f(x)$, the rate of change of $V(t)$ is

$$\dot{\mathcal{E}} = \frac{\mathrm{d}\mathcal{E}(t)}{\mathrm{d}t} = \left(\frac{\|\nabla f(x(t))\|}{f(x(t)) - f(x^*)}\right)^2 \geq \frac{1}{\|x(t) - x^*\|^2} \geq \frac{1}{\|x^{(0)} - x^*\|^2}.$$

Use the Cauchy-Schwartz inequality and the fact that $\|x(t) - x^*\|$ is a decreasing sequence from the previous example. ■

That was easy! Remember that it took us about one lecture to obtain the same result for the discrete-time case. Let us now show a similar result for the strongly convex case. In fact, we can also use a discrete analogue of this result.

Lemma 81 (Descent lemma for strongly-convex functions)**.** *Strongly-convex functions are those wherein if we take off a quadratic they remain convex, i.e.,*

$$f(x) \geq f(y) + \langle \nabla f(y), x - y \rangle + \frac{m}{2}\|x - y\|^2.$$

Just like we did for the descent lemma, following the same proof, we can sandwich $f(x)$ from the other side

$$f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + \frac{1}{2m}\|\nabla f(x) - \nabla f(y)\|^2.$$

In particular, note that if we substitute $y = x^*$ we have

$$f(x) - f(x^*) \leq \frac{1}{2m}\|\nabla f(x)\|^2.$$

Theorem 82. *For smooth, m-strongly-convex f, the gradient flow $\dot{x} = -\nabla f(x)$ has the convergence rate*

$$f(x(t)) - f(x^*) \leq \left(f(x^0) - f(x^*)\right) e^{-2mt} = \mathcal{O}\left(e^{-2mt}\right).$$

Proof. Consider the energy function to be

$$\mathcal{E}(t) := \mathcal{E}(x(t)) = f(x(t)) - f(x^*)$$

and observe that

$$\dot{\mathcal{E}} = \frac{\mathrm{d}\mathcal{E}(t)}{\mathrm{d}t} = -\|\nabla f(x(t))\|^2 \leq -2m\big(f(x(t)) - f(x^*)\big) = -2m\mathcal{E}$$

We proved in Lecture 9 that for smooth functions $f(x)$

$$f(x) \leq f(y) + \langle \nabla f(y), x - y\rangle + \frac{1}{2L}\|\nabla f(x) - \nabla f(y)\|^2$$

Since we know that $L \geq m$ (the former is an upper bound on the largest eigenvalue of the Hessian while the latter is a lower bound on the smallest eigenvalue of the Hessian), we can also write

$$f(x) \leq f(y) + \langle \nabla f(y), x - y\rangle + \frac{1}{2m}\|\nabla f(x) - \nabla f(y)\|^2.$$

Now set $y = x^*$ and $\nabla f(y) = \nabla f(x^*) = 0$ to get

$$\|\nabla f(x(t))\|^2 \geq 2m\left(f(x(t)) - f(x^*)\right).$$

We have an inequality of the form

$$\frac{\dot{\mathcal{E}}}{\mathcal{E}} \leq -2m$$

We can integrate the left hand side to get

$$\log \mathcal{E}(t) - \log \mathcal{E}(0) \leq -\frac{1}{2L}t \Rightarrow \mathcal{E}(t) \leq \mathcal{E}(0)\, e^{-2mt}.$$

■

Lecture 11

ODE for Nesterov's acceleration, stochastic gradient descent

Reading

- "A differential equation for modeling Nesterov's accelerated gradient method: Theory and insights" by Su et al. (2014); read the shorter version at https://statweb.stanford.edu/ candes/papers/NIPS2014.pdf.

11.1 Understanding acceleration using differential equations

Consider Nesterov's scheme again

$$\begin{aligned} x^{(t+1)} &= y^{(t)} - \alpha \nabla f(y^{(t)}) \\ y^{(t)} &= x^{(t)} + \frac{t-1}{t+2}\left(x^{(t)} - x^{(t-1)}\right). \end{aligned}$$

As we saw Nesterov proved that

$$f(x^{(t)}) - f(x^*) \leq \mathcal{O}\left(\frac{L\|x^0 - x^*\|^2}{t^2}\right).$$

As we saw using Nemirovski's lower bound, the rate of $1/t^2$ is optimal, there cannot exist a first-order explicit gradient-based method that achieves a better convergence rate for all dimensions for all convex functions. The proof of Nesterov's method was a tour-de-force of both creativity. There has been a lot of work replicating this proof for other optimization

algorithms, e.g., proximal gradient descent. The proof fundamentally rests on the momentum coefficient $(t-1)/(t+2)$. We will next see how we can get a proof for Nesterov's method that is super short and elegant. This is the topic of today's reading.

11.1.1 Derivation of Nesterov's ODE

We can combine the two update equations to get

$$\frac{x^{(k+1)} - x^{(k)}}{\sqrt{\alpha}} = \frac{k-1}{k+2} \frac{x^{(k)} - x^{(k-1)}}{\sqrt{\alpha}} - \sqrt{\alpha}\, \nabla f(y^{(k)}). \tag{11.1}$$

Imagine a continuous curve $X(t)$ defined as

$$X(k\sqrt{\alpha}) = x^{(k)}.$$

We have rescaled the discrete-valued time variable k to cook up a continuous-valued time variable t.

$$X(t) \approx x^{(t/\sqrt{\alpha})} = x^{(k)} \quad \text{and} \quad X(t+\sqrt{\alpha}) = x^{((k+\sqrt{\alpha})/\sqrt{\alpha})} = x^{(k+1)}.$$

We can now do a Taylor expansion of Nesterov's updates in (11.1) to get

$$\begin{aligned}
\frac{x^{(k+1)} - x^{(k)}}{\sqrt{\alpha}} &= \dot{X}(t) + \frac{1}{2}\ddot{X}(t)\sqrt{\alpha} + \mathcal{O}(\sqrt{\alpha}) \\
\frac{x^{(k)} - x^{(k-1)}}{\sqrt{\alpha}} &= \dot{X}(t) - \frac{1}{2}\ddot{X}(t)\sqrt{\alpha} + \mathcal{O}(\sqrt{\alpha}) \\
\sqrt{\alpha}\nabla f(y^{(k)}) &= \sqrt{\alpha}\, \nabla f(X(t)) + \mathcal{O}(\sqrt{\alpha})
\end{aligned}$$

(show the derivation of the last approximation)

Thus the updates in (11.1) can now be written as

$$\dot{X}(t) + \frac{1}{2}\ddot{X}(t)\sqrt{\alpha} = \left(1 - \frac{3\sqrt{\alpha}}{t}\right)\left(\dot{X}(t) - \frac{1}{2}\ddot{X}(t)\sqrt{\alpha}\right) - \sqrt{\alpha}\, \nabla f(X(t)) + \mathcal{O}(\sqrt{\alpha}).$$

This is true for any α, so by comparing the coefficient of $\sqrt{\alpha}$ we have

$$\ddot{X} + \frac{3}{t}\dot{X} + \nabla f(X) = 0 \tag{11.2}$$

with the initial condition $X(0) = x^0$ as an ordinary differential equation that approximates the updates of Nesterov.

Question 83. A second-order ODE has two initial conditions. What is the initial condition on $\dot{X}(0)$?

Remark 84. Notice that the ODE is singular at time $t = 0$. Su et al. (2014) show that the ODE is well-posed and a solution still exists.

Theorem 85. *Let $X(t)$ be the solution of Nesterov's ODE with initial conditions $X(0) = x^0$ and $\dot{X}(0) = 0$. For any $t > 0$,*

$$f(X(t)) - f(x^*) \leq \frac{2}{t^2} \|x^0 - x^*\|^2.$$

Proof. Consider the energy functional

$$\mathcal{E}(t) := t^2 \left(f(X(t)) - f(x^*)\right) + 2\|X + \frac{t}{2}\dot{X} - x^*\|^2.$$

The time-derivative of $\mathcal{E}$ is

$$\begin{aligned}
\dot{\mathcal{E}} &= 2t \left(f(X(t)) - f(x^*)\right) + t^2 \left\langle \nabla f(X(t)), \dot{X} \right\rangle + 4 \left\langle X + \frac{t}{2}\dot{X} - x^*, \frac{3}{2}\dot{X} + \frac{t}{2}\ddot{X} \right\rangle \\
&= 2t \left(f(X(t)) - f(x^*)\right) + t^2 \left\langle \nabla f(X(t)), \dot{X} \right\rangle + 4 \left\langle X + \frac{t}{2}\dot{X} - x^*, -t\nabla f(X(t)) \right\rangle \\
&= 2t \left(f(X(t)) - f(x^*)\right) + 4 \left\langle X(t) - x^*, -\frac{t}{2}\nabla f(X(t)) \right\rangle \\
&= 2t \left(f(X(t)) - f(x^*)\right) - 2t \left\langle X(t) - x^*, \nabla f(X(t)) \right\rangle \\
&\leq 0 \quad \text{(from convexity)}.
\end{aligned}$$

We therefore have $\mathcal{E}(t) \leq \mathcal{E}(0)$ which gives

$$f(X(t)) - f(x^*) \leq \frac{2\|x^0 - x^*\|}{t^2}.$$

■

Compare the above convergence rate to the discrete-time convergence rate

$$f(x^{(k)}) - f(x^*) \leq \frac{2L\|x^0 - x^*\|}{k^2}.$$

Picking the maximum allowed step-size $a = 1/L$ gives the same form as that of the previous theorem. This again shows that the sampling rate $t = k\sqrt{\alpha}$ is consistent.

Remark 86 (Time-dilation property of continuous curves)**.** Consider re-parametrizing the

time. We will set $\tilde{t} = ct$ for some $c > 0$ to get an ODE

$$\frac{\mathrm{d}^2 X}{\mathrm{d}\tilde{t}^2} + \frac{3}{\tilde{t}} \frac{\mathrm{d}X}{\mathrm{d}\tilde{t}} + \frac{\nabla f(X)}{c^2} = 0.$$

The solution we get if we minimize $f(x)$ or $f(x)/c^2$ is the same. Thus the model of taking a continuous-time limit of Nesterov's iterations is valid even for some other time t that is different from $k\sqrt{\alpha}$ we considered.

11.1.2 The deal about the constant 3

Remember that the constant 3 in Nesterov's ODE comes from the special momentum parameter

$$\frac{k-1}{k+2} \approx 1 - \frac{3}{k}.$$

The same analysis goes through if we use the ODE

$$\frac{\mathrm{d}^2 X}{\mathrm{d}t^2} + \frac{r}{t} \frac{\mathrm{d}X}{\mathrm{d}t} + \nabla f(X) = 0;$$

with

$$r \geq 3.$$

It has the same convergence rate

$$f(X(t)) - f(x^*) \leq \frac{(r-1)^2 \|x^0 - x^*\|}{2t^2}.$$

However if $r < 3$, this result does not hold and there is a phase transition in the convergence rate. The reason for this is the oscillations we saw towards the end of convergence. The friction term

$$-\frac{r}{t}\dot{X}$$

should be high enough to kill the oscillations. Nesterov's iterations show that friction can go to zero with time but it needs to be large enough. Why does it not matter if friction is too large? This is because the large gradient at the beginning of optimization compensates for the high friction. This also leads to the idea of restarting. Restarting is a simple technique to accelerate convergence for non-stochastic optimization problems: each time you find the function increasing

$$f(x^{(t+1)}) > f(x^{(t)})$$

you simply set $y^{(t+1)} = 0$. You can show that this keeps $\left\langle \nabla f, \dot{X} \right\rangle < 0$. Restarting is also essential in theory. We can easily come up with counter-examples which show

that the discreteization of Nesterov's ODE does not converge linearly for strongly-convex functions without restart. It converges with a rate $\tau^{-\frac{2r}{3}}$. With restart, the we get the optimal convergence rate back, i.e., $e^{-\sqrt{L}t}$.

Once you know the ODE of Nesterov, you can cook up lots of optimization algorithms directly. For instance, the ODE for Polyak's Heavy ball method is

$$\ddot{X} + \rho\dot{X} + \nabla f(X) = 0.$$

It is now easy to see based on our discussion of friction why this method can diverge if you don't pick ρ properly.

Picking the correct coefficient for friction seems difficult. Pick it too high and you slow down the algorithm at the beginning where it really matters. Pick it too low and you might slow down the algorithm towards the end. Let's think about picking friction using the curvature: we can hope to adapt the

How would Newton's iteration look in continuous time? It is easy:

$$\nabla^2 f(X)\dot{X} + \nabla f(X) = 0. \tag{11.3}$$

We can now merge this with Nesterov's ODE to get an update dynamics

$$\ddot{X} + \frac{r\dot{X}}{t} + p\nabla^2 f(X)\dot{X} + \nabla f(X) = 0. \tag{11.4}$$

This is known as Damped Newton's iteration. A few years ago, we were playing with an update equation of the kind

$$\begin{aligned} x^{(t+1)} &= x^{(t+1)} + \alpha\left(y^{(t)} - \nabla f(x^{(t)})\right) \\ y^{(t+1)} &= y^{(t+1)} + \alpha\left(-\beta y^{(t)} - \nabla f(x^{(t)})\right) \end{aligned}$$

as a way of coming up with a restarting scheme to accelerate optimization. Turns out the ODE for these updates is exactly the damped Newton.

The continuous-time point of view is powerful. One can use lots of ideas such a Euler-Lagrange equations, control theory, Hamiltonian systems that preserve energy etc. to understand optimization. The paper by Wibisono et al. (2016) is a great reading about these ideas. If you want something more fun, you can also read https://web.stanford.edu/group/lavxm/publications/book-chapters/LeMaOrWe04b.pdf. We will see these ideas again when we look at Markov chain Monte Carlo (MCMC) algorithms towards the end of the class.

11.2 Stochastic gradient descent

SGD has its roots in stochastic optimization (). A stochastic optimization problem looks like

$$x^* = \operatorname*{argmin}_x \mathop{\mathbb{E}}_{\xi} [f(x;\xi)]$$

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., (**?**). 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 x 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.

In machine learning today, we solve a slightly different but related problem. This is called the finite-sum problem. Given a finite dataset $D = \{(\xi_i, y_i)\}_{i=1,\dots,n}$ we minimize

$$f(x) := \frac{1}{n} \sum_{i=1}^{n} \ell(x; \xi_i, y_i).$$

I have changed the notation for the optimization part of the course. The variable x denotes the parameters of the model and ξ_i denotes the i^{th} datum. As we have discussed before, the number of data n can be very large in modern machine learning (can anyone say why?). It is difficult to do gradient descent in this case because the gradient

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(x; \xi_i, y_i).$$

involves a sum the entire dataset. Note that the convergence rate of optimization algorithms we have look at is independent of the dimension; this is a general principle for gradient-based optimization algorithms (for which algorithms is this not true?).

Stochastic gradient descent for the finite-sum case performs the following iterations

$$x^{(t+1)} = x^{(t)} - \eta \nabla \ell(x^{(t)}; \xi_{i_t}, y_{i_t}) \tag{11.5}$$

The datum (x_{i_t}, y_{i_t}) over which we compute the gradient before updating the weights is picked randomly from the dataset D. So each iteration of SGD is a factor of n times faster than that of gradient descent. Is this the direction of steepest descent though?

Example 87. Play with the step-size at
http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html.

Example 88. Let us take a quadratic loss function with scalar data $\{(a_i, b_i)\}$.

$$f(x) = \frac{1}{2n} \sum_{i=1}^{n} (a_i x - b_i)^2 .$$

Discuss the solution of this problem and the fact that SGD will find solutions that try to fit each datum at each iteration but bounce around.

- Very fast speed outside the region of confusion.
- Once you get close to the optimum, the gradients keep fighting at successive iterations.

Remark 89. Sampling with and without replacement

Remark 90 (Mini-batch version of SGD)**.**

$$x^{(t+1)} = x^{(t)} - \frac{\eta}{\mathscr{b}} \sum_{k=1}^{\mathscr{b}} \nabla \ell(x^{(t)}; \xi_{t_k}, y_{t_k}). \tag{11.6}$$

Samples in the mini-batch $\{(\xi_{t_k}, y_{t_k})\}_{k=1,\ldots,\mathscr{b}}$ are picked randomly from the dataset at each iteration. Note that if $\mathscr{b} = n$, this is exactly gradient descent.

Lecture 12

Stochastic gradient descent, Markov chains

Reading

- "Stochastic gradient descent tricks" by Bottou (2012). Great paper with lots of little tricks of how to use SGD in practice.
- Till 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.
- http://proceedings.mlr.press/v28/sutskever13.pdf
- https://distill.pub/2017/momentum

12.1 Stochastic gradient descent

SGD has its roots in stochastic optimization (). A stochastic optimization problem looks like

$$x^* = \operatorname*{argmin}_x \mathop{\mathbb{E}}_{\xi} [f(x; \xi)]$$

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 x 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.

In machine learning today, we solve a slightly different but related problem. This is called the finite-sum problem. Given a finite dataset $D = \{(\xi_i, y_i)\}_{i=1,\dots,n}$ we minimize

$$f(x) := \frac{1}{n} \sum_{i=1}^{n} \ell(x; \xi_i, y_i).$$

I have changed the notation for the optimization part of the course. The variable x denotes the parameters of the model and ξ_i denotes the i^{th} datum. As we have discussed before, the number of data n can be very large in modern machine learning (can anyone say why?). It is difficult to do gradient descent in this case because the gradient

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(x; \xi_i, y_i).$$

involves a sum the entire dataset. Note that the convergence rate of optimization algorithms we have look at is independent of the dimension; this is a general principle for gradient-based optimization algorithms (for which algorithms is this not true?).

Stochastic gradient descent for the finite-sum case performs the following iterations

$$x^{(t+1)} = x^{(t)} - \alpha \nabla \ell(x^{(t)}; \xi_{\omega_t}, y_{\omega_t}) \tag{12.1}$$

The datum $(x_{\omega_t}, y_{\omega_t})$ over which we compute the gradient before updating the weights is picked randomly from the dataset D. So each iteration of SGD is a factor of n times faster than that of gradient descent. Is this the direction of steepest descent though?

Example 91. Play with the step-size at
http://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html.

Example 92. Let us take a quadratic loss function with scalar data $\{(a_i, b_i)\}$.

$$f(x) = \frac{1}{2n} \sum_{i=1}^{n} (a_i x - b_i)^2 .$$

Discuss the solution of this problem and the fact that SGD will find solutions that try to fit each datum at each iteration but bounce around.

- Very fast speed outside the region of confusion.
- Once you get close to the optimum, the gradients keep fighting at successive iterations.

Remark 93. Sampling with and without replacement.

Remark 94 (Mini-batch version of SGD)**.**

$$x^{(t+1)} = x^{(t)} - \frac{\eta}{\mathscr{b}} \sum_{k=1}^{\mathscr{b}} \nabla \ell(x^{(t)}; \xi_{\omega_t^k}, y_{\omega_t^k}). \tag{12.2}$$

Samples in the mini-batch $\left\{(\xi_{\omega_t^k}, y_{\omega_t^k})\right\}_{k=1,\dots,\mathscr{b}}$ are picked randomly from the dataset at each iteration. Note that if $\mathscr{b} = n$, this is exactly gradient descent.

Let us denote the mini-batch gradient with a batch-size of size $\mathscr{b}$ to be

$$\nabla f_{\mathscr{b}}(x^{(t)}) := \frac{1}{\mathscr{b}} \sum_{k=1}^{\mathscr{b}} \nabla \ell(x^{(t)}; \xi_{\omega_t^k}, y_{\omega_t^k}).$$

This will help us write SGD in short-form as

$$x^{(t+1)} = x^{(t)} - \eta \nabla f_{\mathscr{b}}(x^{(t)}).$$

This looks very similar to gradient descent, except that there is a $\mathscr{b}$ at the subscript. To make our notation in the following a bit clearer, we will also use SGD with a batch-size of $\mathscr{b} = 1$ but denote it as

$$\nabla f_{\omega}(x^{(t)}) := \nabla \ell(x^{(t)}; \xi_{\omega_t}, y_{\omega_t}).$$

In other words ∇f_{ω} is simply the stochastic gradient $\nabla f_{\mathscr{b}}$ with $\mathscr{b} = 1$.

12.2 Convergence rate for SGD

We will only consider strongly convex functions for analyzing SGD. The proofs are much more tedious for the others. Remember that gradient descent requires $\mathcal{O}(n)$ operations each iteration. For strongly convex functions, gradient descent converges at a rate

$$f(x^{(t)}) - f(x^*) \leq \mathcal{O}(c^t).$$

The number of steps required to reach an error ϵ is $\mathcal{O}(\log(1/\epsilon))$. Each step needs $\mathcal{O}(n)$ operations and therefore the total computation used by GD is

$$\mathcal{O}(n \log(1/\epsilon)).$$

The updates of SGD are stochastic so

$$f(x^{(t)}) - f(x^*)$$

is a random variable. It depends on the initial condition and the random data that were chosen in the first t iterations to compute the gradients. How should we understand the convergence of SGD then? SGD clearly does not "converge" in the zone of confusion in the sense of iterates stopping to move. Every time we pick a new example or a new mini-batch, you get some gradient and you move.

Remark 95. There are many notions of convergence for a sequence of random variables $X_1, X_2, \ldots, X_n$:

- convergence almost surely:

$$\mathbb{P}\left(\lim_{n\to\infty} X_n = X^*\right) = 1.$$

- convergence in moments of order r (this is the one we will use)

$$\mathbb{E}\left[|X_n - X^*|^r\right] \to 0.$$

- convergence in probability: for all $\epsilon > 0$,

$$\mathbb{P}\left(|X_n - X^*| \geq \epsilon\right) \to 0.$$

- weak convergence (convergence in distribution)

$$\mathbb{E}\left[h(X_n)\right] \to \mathbb{E}\left(h(X^*)\right)$$

  for all continuous bounded functions h.

As ususal let's use L-smooth and m-strongly convex function $f(x)$ (we do not need strong convexit for the following two lemmas but will use it later). Note this is $f(x)$, not $\ell(x, \xi, y)$. We can get a descent-lemma style result now.

Lemma 96 (Descent lemma for stochastic updates)**.** *The next update for SGD satisfies*

$$\mathop{\mathbb{E}}_{\omega_t}\left[f(x^{(t+1)})\right] - f(x^{(t)}) \leq -\alpha \nabla f(x^{(t)})^\top \mathop{\mathbb{E}}_{\omega_t}\left[\nabla f_{\omega_t}(x^{(t)})\right] + \frac{L\alpha^2}{2} \mathop{\mathbb{E}}_{\omega_t}\left[\|\nabla f_{\omega_t}(x^{(t)})\|^2\right].$$

Compare this with the descent lemma that we saw for gradient descent

$$f(x^{(t+1)}) \leq f(x^{(t)}) - \alpha \nabla f(x^{(t)})^\top \nabla f(x^{(t)}) + \frac{L\alpha^2}{2} \|\nabla f(x^{(t)})\|^2.$$

Proof. Use the descent lemma from Lecture 9, substitute the iterates of SGD and take an expectation on both sides over the index of the datum ω_t. ■

Typical assumptions in SGD analysis. Based on the previous lemma, we can construct a set of assumptions that help prove convergence. Remember that if we can bound the right hand side of the above inequality using some deterministic quantity, we should be good to prove the convergence of SGD and obtain a rate. This is similar to what we did for gradient descent.

- Assume that the stochastic gradient is unbiased

  $$\nabla f(x) = \mathop{\mathbb{E}}_{\omega} [\nabla f_\omega(x)]$$

  for all x 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 you will change it only if you are doing tricks with the sampling distribution, e.g., boosting etc.

- The second condition is important. There exist scalars σ_0 and σ such that

  $$\mathop{\mathbb{E}}_{\omega_t} \left[\|\nabla f_\omega(x)\|^2 \right] \leq \sigma_0 + \sigma \|\nabla f(x)\|^2.$$

  This condition assumes something about the second term in the descent lemma for SGD. It assumes that the stochastic gradient is not too bad: near a critical point (locations where $\nabla f(x) = 0$), it is allowed to grow in a similar fashion as the true gradient except with a scaling factor $\sigma > 0$ and a constant σ_0.

Lemma 97. *Under the above assumptions on the loss function in SGD, we have*

$$\begin{aligned} \mathop{\mathbb{E}}_{\omega_t} \left[f(x^{(t+1)}) \right] - f(x^{(t)}) &\leq -\alpha \|\nabla f(x^{(t)})\|^2 + \frac{L\alpha^2}{2} \mathop{\mathbb{E}}_{\omega_t} \left[\|\nabla f_{\omega_t}\|^2 \right] \\ &\leq -\left(1 - \frac{L\alpha\sigma}{2}\right) \alpha \|\nabla f(x^{(t)})\|^2 + \frac{\alpha^2 L \sigma_0}{2}. \end{aligned}$$

To prove this simply substitute the assumptions into Lemma 96. Compare this to the

corresponding result we derived for gradient descent in Lecture 9

$$f(x^{(t+1)}) - f(x^{(t)}) \leq -\frac{\alpha}{2}\|\nabla f(x^{(t)})\|^2.$$

For strongly convex functions, you can pick a larger step-size in gradient descent $\alpha < 2/L$ and obtain a similar expression as Lemma 97.

Notice that the full objective at the next step depends only on the index of the datum we picked at this iteration and the step-size α. It does not depend on any of the past iterates. We have essentially achieved our goal. We upper bounded the left hand side using a deterministic quantity. Notice that for small α, the first term is strictly negative. However the second term might be quite large if σ_0 is large. Picking the step-size α in such a way that it balances of these two terms in SGD is critical to get good performance in practice.

Theorem 98 (Convergence rate of SGD for smooth, strongly-convex functions)**.** *If we pick a step-size*

$$\alpha \leq \frac{1}{L\sigma}$$

then the expected optimality gap satisfies

$$\mathop{\mathbb{E}}_{\omega_1,\omega_2,\ldots,\omega_t}\left[f(x^{(t+1)})\right] - f(x^*) \leq \frac{\alpha L \sigma_0}{2m} + (1-\alpha m)^t\left(f(x^0) - f(x^*) - \frac{\alpha L \sigma_0}{2m}\right).$$

Proof. The proof follows by a direct application of the descent lemma Lemma 97, see Bottou et al. (2018) Theorem 4.6 for the proof. ■

This theorem beautiful 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 $f(x^{(t+1)})$ converges at a linear rate $(1-\alpha m)^t$. Some points to notice

- When gradient computation 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.
- If we pick a small α, we get closer to the global minimum but go there quite slowly. On the other hand, we can pick a large α and get to a neighborhood of the global minimum quickly but we will then have a large error leftover at the end.

This was well-known even to Robbins and Monro (1951). We should pick the step-size to decay with time. But we do not want to decay too quickly, which might slow down the

progress. A good schedule to pick is such that

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

Theorem 99 (SGD convergence with decaying step-size)**.** *If we pick a step-size schedule*

$$\alpha_t = \frac{\beta}{t + t_0} \quad \textit{where} \quad \beta > \frac{1}{m}, \textit{ and } t_0 \textit{ is such that } \alpha_1 < \frac{1}{L\sigma}.$$

then the expected optimality gap satisfies

$$\mathop{\mathbb{E}}_{\omega_1,\ldots,\omega_t} \left[f(x^{(t+1)}) - f(x^*) \right] \leq \mathcal{O}\left(\frac{1}{t + t_0} \right).$$

We will not do the proof. If you are interested, see Theorem 4.7 in Bottou et al. (2018). Notice that by decaying the step-size we converge only at a sub-linear rate even for strongly convex loss functions.

Remark 100 (Mini-batching in SGD)**.** Does mini-batching provide an improvement in the number of iterations? If we use the gradient

$$\nabla f_{\mathscr{b}}$$

instead of ∇f_{ω_t} to make updates, the variance of the stochastic gradients decreases by a factor of $\mathscr{b}$. The constants σ and σ_0 therefore decrease by a factor of $\mathscr{b}$ as well (remember that we want as tight an inequality as we can get when we want to analyze the converge rate of an algorithm). This modifies the theorem for expected optimality gap to

$$\mathop{\mathbb{E}}_{\omega_1,\omega_2,\ldots,\omega_t} \left[f(x^{(t+1)}) \right] - f(x^*) \leq \frac{\alpha L \sigma_0}{2m\mathscr{b}} + (1 - \alpha m)^t \left(f(x^0) - f(x^*) - \frac{\alpha L \sigma_0}{2m\mathscr{b}} \right).$$

Compare to the convergence rate for single-sample SGD update and notice that if we use a step-size $\alpha/\mathscr{b}$ in single-sample SGD, we obtain a similar expression expect that the contraction rate $(1 - \alpha m)^t$ becomes

$$1 - \frac{\alpha}{m\mathscr{b}}.$$

This roughly indicates that if we were to use single-sample SGD, we need to run $O(\mathscr{b})$ more iterations to achieve the same optimality gap as mini-batch SGD with a batch-size of $\mathscr{b}$. Each iteration of mini-batch SGD is $\mathscr{b}$ times more expensive each iteration of single-sample SGD. We do not gain anything by using mini-batch SGD. Then why do we use it in practice?

Remark 101. First show using the arithmetic-mean greater than or equal to geometric-mean inequality that $\sigma \geq 1$. It is reasonable to imagine that if we use mini-batch SGD, we could use b times larger step-size than the single-sample SGD. Is this correct? Notice that the condition in Theorem 103 indicates that the largest initial step-size we are allowed is $\frac{1}{L}$. Effectively, we could imagine increasing the step-size with time if we are using mini-batch SGD, we just need to careful not to use too large a step-size at the beginning.

Remark 102. Compare the number of computations of GD with SGD. If the average accuracy you desire on every sample in the dataset is less than $\mathcal{O}(1/n)$ then you should use stochastic gradient descent. If you want better accuracy, gradient descent is faster. This explains in a very simple way why stochastic gradient descent is so powerful for machine learning. We do not care about getting a very low error on the training set.

12.3 Acceleration of stochastic optimization

The convergence rate of SGD is quite bad, it is sub-linear. Let us look at a simple technique to improve the convergence rate

12.3.1 Polyak-Ruppert averaging

For sub-optimal stochastic optimization algorithms, we can typically gain acceleration simply by averaging the iterates. Consider the SGD updates

$$\begin{aligned} x^{(t+1)} &= x^{(t)} - \alpha_t \nabla f_b(x^{(t)}) \\ y^{(t)} &= \frac{x^{(1)} + x^{(2)} + \ldots + x^{(t)}}{t}. \end{aligned} \tag{12.3}$$

This is known as Polyak-Ruppert averaging (Polyak, 1990; Polyak and Juditsky, 1992; Ruppert, 1988). We maintain the average of all the past steps and every time someone asks for a model we output $y^{(t)}$ instead of $x^{(t)}$. What is shown in the above papers is that the convergence rate of this averaged iterate $y^{(t)}$ is as fast a second-order SGD method

$$x^{(t+1)} = x^{(t)} - \left(\nabla^2 f(x)\right)^{-1} \nabla f_b(x^{(t)}).$$

The convergence rate of averaged SGD is still sub-linear.

12.3.2 Momentum and SGD

Does momentum help for stochastic gradient descent? Let us look at an animation from https://distill.pub/2017/momentum.

Lecture 13

Acceleration of SGD, Markov chains

Reading

- "A Stochastic Gradient Method with an Exponential Convergence Rate for Finite Training Sets" by Le Roux et al. (2012)
- "A variational analysis of stochastic gradient algorithms" by Mandt et al. (2016)

Theorem 103 (Convergence rate of SGD for smooth, strongly-convex functions)**.** *If we pick a step-size*

$$\alpha \leq \frac{1}{L\sigma}$$

then the expected optimality gap satisfies

$$\mathop{\mathbb{E}}_{\omega_1,\omega_2,\ldots,\omega_t}\left[f(x^{(t+1)})\right] - f(x^*) \leq \frac{\alpha L \sigma_0}{2m} + (1-\alpha m)^t \left(f(x^0) - f(x^*) - \frac{\alpha L \sigma_0}{2m}\right).$$

Proof. The proof follows by a direct application of the descent lemma for SGD in the previous lecture, see Bottou et al. (2018) Theorem 4.6 for the proof. ■

This theorem beautifully demonstrates the interplay between the step-size and and the variance of SGD gradients. Note that gradient descent converges at a linear rate c^t. Some points to notice

- When gradient computation 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.

- If we pick a small α, we get closer to the global minimum but go there quite slowly. On the other hand, we can pick a large α and get to a neighborhood of the global minimum quickly but we will then have a large leftover error at the end.

This was well-known even to Robbins and Monro (1951). To perform this trade-off between large α and small α, we should pick the step-size to decay with time. But we do not want to decay too quickly and slow down the progress. A good schedule to pick is such that

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

Theorem 104 (SGD convergence with decaying step-size)**.** *If we pick a step-size schedule*

$$\alpha_t < \frac{1}{m(t+t_0)} \quad \textit{where} \quad t_0 \textit{ is such that } \alpha_1 < \frac{1}{L\sigma}.$$

then the expected optimality gap satisfies

$$\mathop{\mathbb{E}}_{\omega_1,\ldots,\omega_t}\left[f(x^{(t+1)}) - f(x^*)\right] \leq \mathcal{O}\left(\frac{1}{t+t_0}\right).$$

We will not do the proof. If you are interested, see Theorem 4.7 in Bottou et al. (2018). Notice that by decaying the step-size we converge only at a sub-linear rate even for strongly convex loss functions.

Remark 105 (Mini-batching in SGD)**.** Does mini-batching provide an improvement in the number of iterations? If we use the gradient

$$\nabla f_{\cancel{b}}$$

instead of ∇f_{ω_t} to make updates, the variance of the stochastic gradients decreases by a factor of $\cancel{b}$. The constants σ and σ_0 therefore decrease by a factor of $\cancel{b}$ as well (remember that we want as tight an inequality as we can get when we want to analyze the converge rate of an algorithm). This modifies the theorem for expected optimality gap to

$$\mathop{\mathbb{E}}_{\omega_1,\omega_2,\ldots,\omega_t}\left[f(x^{(t+1)})\right] - f(x^*) \leq \frac{\alpha L\sigma_0}{2m\cancel{b}} + (1-\alpha m)^t\left(f(x^0) - f(x^*) - \frac{\alpha L\sigma_0}{2m\cancel{b}}\right).$$

Compare to the convergence rate for single-sample SGD update and notice that if we use a step-size $\alpha/\cancel{b}$ in single-sample SGD, we obtain a similar expression expect that the

contraction rate $(1-\alpha m)^t$ becomes

$$1-\frac{\alpha}{m\mathscr{b}}.$$

This roughly indicates that if we were to use single-sample SGD, we need to run $O(\mathscr{b})$ more iterations to achieve the same optimality gap as mini-batch SGD with a batch-size of $\mathscr{b}$. Each iteration of mini-batch SGD is $\mathscr{b}$ times more expensive each iteration of single-sample SGD. We do not gain anything by using mini-batch SGD. Then why do we use it in practice?

Remark 106. First show using the arithmetic-mean greater than or equal to geometric-mean inequality that $\sigma \geq 1$. It is reasonable to imagine that if we use mini-batch SGD, we could use $\mathscr{b}$ times larger step-size than the single-sample SGD. Is this correct? Notice that the condition in Theorem 103 indicates that the largest initial step-size we are allowed with $\sigma > 1$ is $\frac{1}{L}$. Effectively, we could imagine increasing the step-size with time if we are using mini-batch SGD, we just need to careful not to use too large a step-size at the beginning. We will see this again in a couple of classes as "warmup".

Remark 107. Compare the number of computations of GD with SGD. If the average accuracy you desire on every sample in the dataset is less than $\mathcal{O}(1/n)$ then you should use stochastic gradient descent. If you want better accuracy, gradient descent is faster. This explains in a very simple way why stochastic gradient descent is so powerful for machine learning. We do not care about getting a very low error on the training set.

Remark 108 (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 only reducing the size of the ball in which SGD bounces after it has converged.

13.1 Acceleration of stochastic optimization

The convergence rate of SGD is quite bad, it is sub-linear. Let us look at a simple technique to improve the convergence rate

13.1.1 Polyak-Ruppert averaging

For sub-optimal stochastic optimization algorithms, we can typically gain acceleration simply by averaging the iterates. Consider the SGD updates

$$\begin{aligned} x^{(t+1)} &= x^{(t)} - \alpha_t \nabla f_{\mathfrak{b}}(x^{(t)}) \\ y^{(t)} &= \frac{x^{(1)} + x^{(2)} + \ldots + x^{(t)}}{t}. \end{aligned} \tag{13.1}$$

This is known as Polyak-Ruppert averaging (Polyak, 1990; Polyak and Juditsky, 1992; Ruppert, 1988). We maintain the average of all the past steps and every time someone asks for a model we output $y^{(t)}$ instead of $x^{(t)}$. What is shown in the above papers is that the convergence rate of this averaged iterate $y^{(t)}$ is as fast a second-order SGD method

$$x^{(t+1)} = x^{(t)} - \left(\nabla^2 f(x)\right)^{-1} \nabla f_{\mathfrak{b}}(x^{(t)}).$$

The convergence rate of averaged SGD is still sub-linear. It is $\mathcal{O}(1/t)$ but with a better constant in the $\mathcal{O}(\cdot)$ than that of SGD.

13.1.2 Momentum does not help accelerate SGD

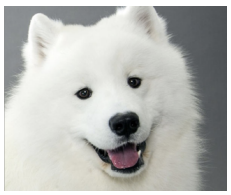
The power of momentum lies in making faster progress at the beginning of training when the gradient is large. The stochastic gradient is not a good proxy for the true gradient however, so if we compute a momentum that is the difference $x^{(t+1)} - x^{(t)}$ for two iterates of SGD, it may not always point in the direction of $\nabla f(x^{(t)})$. Here is a surprising and deep fact that you should remember (Kidambi et al., 2018; Liu and Belkin, 2018).

Nesterov's momentum does not accelerate convergence for SGD

To be more precise, the authors in the above papers come up with a simple counterexample where they show that Nesterov's updates when used with stochastic gradients *do not* improve the $e^{-t/\kappa}$ rate for the second term of SGD to something like $e^{-t/\sqrt{\kappa}}$, Nesterov's updates only lead to an improvement of the form $e^{-ct/\kappa}$. There are advanced variants of SGD that lead to acceleration. One example is the algorithm of Liu and Belkin (2018)

$$\begin{aligned} x^{(t+1)} &= x^{(t)} - \alpha \nabla f_{\omega_t}(y^{(t)}) \\ y^{(t)} &= x^{(t)} + \rho(x^{(t)} - x^{(t-1)}) + \beta \nabla f_{\omega_{t-1}}(y^{(t)}). \end{aligned}$$

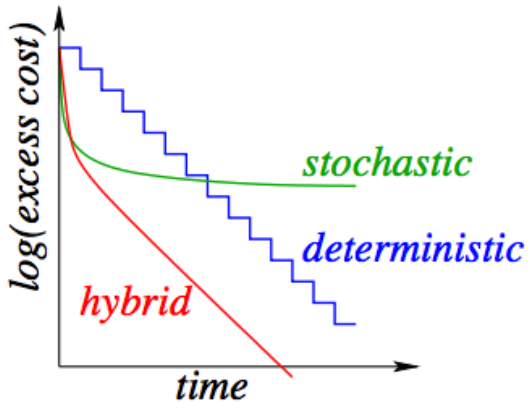
Question 109. So why do we use Nesterov's acceleration while training a neural network? Did you try training a network with and without acceleration? The former is definitely faster, can you guess why?



The datasets we use in computer vision are quite clean, they were curated very carefully and contain images that are quite similar to each other. The gradient of the same neural network on two similar images is quite similar and consequently, even with few samples in the mini-batch in practice, we get a very good proxy for the true gradient on the entire dataset.

13.1.3 Stochastic averaged gradient (SAG)

Let's draw a picture of all the convergence rates we have seen so far.



The key observation is that non-stochastic gradient descent is so fast because it evaluates all the n gradients at each step. We did not use the fact that the dataset is finite in our analysis of SGD yet. A natural question to ask then is whether we can get improved convergence for SGD if we consider a finite dataset.

There has been about 50 years of work in stochastic optimization algorithms. The paper by Le Roux et al. (2012) which is today's assigned reading is the first method that was shown

to achieve linear convergence rate for SGD. It is also quite simple to understand. Consider the gradient descent update which we can rewrite as

$$\begin{aligned} x^{(t+1)} &= x^{(t)} - \frac{\alpha}{n} \sum_{i=1}^{n} \nabla f(x^{(t)}, (\xi_i, y_i)) \\ &=: x^{(t)} - \frac{\alpha}{n} \sum_{i=1}^{n} v_i^t. \end{aligned} \tag{13.2}$$

We can think of the quantity v_i^t as a buffer which maintains the gradient of the weights $x^{(t)}$ on each of the data i in the dataset. The length of this buffer is n, one for each datum. The update of gradient descent is a sum of the buffer. The update of stochastic gradient descent takes one random sample from this buffer and throws the buffer away before moving on to the next iterate $x^{(t+1)}$. The update of stochastic averaged gradient (SAG) does not throw the buffer away. The SAG algorithm samples an ω_t at each time-step t and only updates that particular entry of the buffer

$$v_i^t = \begin{cases} \nabla f_{\omega_t}(x^{(t)}) & \text{if } \omega_t = i \\ v_i^{t-1} & \text{else.} \end{cases} \tag{13.3}$$

This simple modification matches the linear convergence rate of non-stochastic gradient descent even with stochastic gradient updates.

Remark 110. Polyak's heavy ball method performs geometric averaging of the previous gradients. We can rewrite Heavy Ball momentum with stochastic gradient updates as

$$x^{(t+1)} = x^{(t)} - \alpha \sum_{j=1}^{t} \rho^{t-j} \nabla f_{\omega_j}(x^{(j)}).$$

Note that the momentum parameter is smaller than 1, so a gradient $f_{\omega_j}(x^{(j)})$ that is far away in history gets smaller weight in the sum. SAG on the other hand performs a selection and averaging

$$x^{(t+1)} = x^{(t)} - \alpha \sum_{j=1}^{t} \chi_j \nabla f_{\omega_j}(x^{(j)}).$$

where χ_j is $1/n$ if the ω_j^{th} element of the buffer was not refreshed after the j^{th} iteration; it is zero otherwise.

Question 111 (What is the catch with the SAG algorithm?)**.** We cannot execute it for a very large dataset. The amount of memory required is $\mathcal{O}(nd)$, the factor of n is the number of data and the d is the dimensionality of each gradient.

Question 112. Clearly SAG uses stale gradients, it will take lots of time-steps until every element of the buffer is updated, even more so if the dataset is quite large. Why does this not hurt? Remember when we saw the scheme for a general descent algorithm

$$x^{(t+1)} = x^{(t)} + \alpha\, d^{(t)}$$

the $d^{(t)}$ did not have to depend only on $x^{(t)}$, it could also depend on the entire history $\left\{x^{(1)}, x^{(2)}, \ldots, x^{(t)}\right\}$. So we will not ruin speed so long as the SAG gradient is highly correlated with the true gradient $\nabla f(x^{(t)})$. This is an instance of manipulating the bias-variance tradeoff in our favor. The gradient of SGD is

$$\nabla f_{\omega_t}(x^{(t)})$$

which is unbiased with respect to the true gradient $\nabla f(x^{(t)})$ but may have a large variance because we are using only one datum. The gradient of SAG on the other hand is

$$\frac{1}{n}\sum_{i=1}^{n} v_i^t = \underbrace{\underbrace{v_{\omega_t}^t - v_{\omega_t}^{(t-1)}}_{\text{update to buffer}} + \underbrace{\frac{1}{n}\sum_{i=1}^{n} v_i^{(t-1)}}_{\text{old average of buffer}}}_{\text{new average of buffer}} \tag{13.4}$$

which is biased with respect to the true gradient $\nabla f(x^{(t)})$ because it depends on the past iterates $\left\{x^{(1)}, x^{(2)}, \ldots, x^{(t)}\right\}$ but it may have a smaller variance because we are summing up n terms. You can also reduce the bias by refreshing the entire buffer every once in a while, say every n iterations so that the computational complexity of refreshing the buffer is amortized.

Let us look at another neat trick which removes the need for maintaing this large buffer.

13.1.4 Control variate for variance reduction

This is a general concept from the literature on Monte-Carlo integration. It is typically introduced as follows. Say we have a random variable X and we would like to guess its expected value $\mu = \mathbb{E}[X]$. Note that X is an unbiased estimator of μ but it may have a large variance. If we have another random variable Y with known expected value $\mathbb{E}[Y]$, then

$$\mathrm{H}X = X + c(Y - \mathbb{E}[Y]) \tag{13.5}$$

is also an unbiased estimator for μ for any value of c. The variance of $\mathrm{H}X$ is

$$\mathrm{Var}(\mathrm{H}X) = \mathrm{Var}(X) + c^2\,\mathrm{Var}(Y) + 2c\;\mathrm{Cov}(X, Y).$$

which is minimized for

$$c^* = -\frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(Y)}$$

to

$$\begin{aligned}\mathrm{Var}(\mathrm{H}X) &= \mathrm{Var}(X) - c^{*2}\;\mathrm{Var}(Y) \\ &= \left(1 - \left(\frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(Y)}\right)^2\right)\mathrm{Var}(X).\end{aligned}$$

By subtracting $Y - \mathbb{E}[Y]$ from our observed random variable X, we have reduced the variance of X if the correlation between X and Y is non-zero. Most importantly, note that no matter what Y we plug into the above expression, we can never increase the variance of X; the worst that can happen is that we pick a Y that is completely uncorrelated with X and end up achieving nothing.

Consider (13.4) which has a similar form as that of (13.5). Our current random variable $v_{\omega_t}^t$ (akin to X in (13.5)) first loses a term $v_{\omega_t}^{(t-1)}$ (like Y) and then gains a term $\frac{1}{n}\sum_{i=1}^n v_i^{(t-1)}$. Note that we like to think of $v_{\omega_t}^t$ as a proxy for the stochastic gradient $\nabla f_{\omega_t}(x^{(t)})$. So we are really reducing the variance of our stochastic gradient by subtracting a Y and then adding back $\mathbb{E}[Y]$.

Stochastic Variance Reduced Gradient (SVRG by Johnson and Zhang (2013)) uses this observation to write the update of SGD as

$$x^{(t+1)} = x^{(t)} - \alpha\left(\nabla f_{\omega_t}(x^{(t)}) - \nabla f_{\omega_t}(x^{\text{ckpt}}) + \frac{1}{n}\sum_{i=1}^n \nabla f_i(x^{\text{ckpt}})\right). \tag{13.6}$$

This algorithm maintains a copy of the weights x^{ckpt} and the full gradient evaluated at this checkpoint $\frac{1}{n}\sum_{i=1}^n \nabla f_i(x^{\text{ckpt}})$. At every iteration it evaluates two stochastic gradients, one

on $x^{(t)}$ and one on x^{ckpt} using the same randomly sampled datum ω_t.

The buffer for SVRG is therefore only of size $\mathcal{O}(d)$ but the computational complexity has increased by $2\times$, we are evaluating two gradients at each iteration as compared to SGD or SAG.

Question 113 (How to update the checkpoint x^{ckpt}?)**.** Updating it too frequently (say faster than every $\mathcal{O}(n)$ updates) ruins our computational complexity. If we update it too slowly, the correlation $\text{Cov}(x^{(t)}, x^{\text{ckpt}})$ might be essentially zero and we do not get any variance reduction; the convergence rate would be similar to that of SGD in this case. We can check when this correlation is too small and update the checkpoint accordingly.

Lecture 14

Markov chains, Gibbs distribution

Reading

- "A variational analysis of stochastic gradient algorithms" by Mandt et al. (2016)

14.1 Continuous-time analog of SGD

We saw that the continuous-time point of view for gradient descent gives very quick and clean results as compared to using the discrete-time update equations. The analysis of stochastic gradient descent is quite new (most advancements like SAG, SVRG, proximal terms to control the covariance etc. are within the last 7 years). The fashion of using continuous-time analysis for stochastic algorithms is much newer, within the past 3 years, but it gives an equally powerful understanding of these algorithms.

The big problem however is that continuous-time stochastic processes are difficult to handle mathematically and involve different kinds of calculi (some of you may have heard of names like Ito's calculus or Stratonovich calculus). Further, mathematically stochastic differential equations do not allow us to talk about one trajectory of optimization (like we always did for gradient flows). So this style of analysis does not give a convergence rate but it is useful to obtain different kind of results.

We will not go deep into this kind of analysis. Our objective in this section is to be as heuristic as we can while understanding the general principles behind stochastic optimization for neural networks.

14.1.1 How do our convergence results change if the function is not convex?

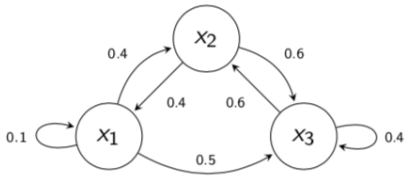
(picture of a convex loss function, sgd bounces around. What happens if we never decrease the step-size?)

14.1.2 Markov chains

Consider the Whack-The-Mole game: a mole has borrowed a network of three holes x^1, x^2, x^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 x^j given that it resurfaced at hole x^i the last time.

P^t is the t-step transition matrix. For a discrete state, discrete time Markov chain with $N < \infty$ states where all states $x^i \in X$ for $i \leq N$,

$$P_{ij}^t = \mathbb{P}(x^{(t)} = x^j \,|\, x^{(0)} = x^i).$$

If there exist times t, t' such the both the probabilities

$$\mathbb{P}(x^{(t)} = x^j \,|\, x^{(0)} = x^i) \quad \mathbb{P}(x^{(t')} = x^i \,|\, x^{(0)} = x^j)$$

are non-zero the two states x^i and x^j are said to "communicate"

$$x^i \leftrightarrow x^j$$

The set of states that all communicate with each other are an equivalence class. You can visit from from any state in this equivalence class to any other state with non-zero probability, you might have to wait for a long time. 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 x 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.

Remark 114 (Invariant measure of a Markov chain)**.** The probability of being in a state x^i at time $t+1$ can be written as

$$\mathbb{P}(x^{(t+1)} = x^i) = \sum_{j=1}^{N} \mathbb{P}(x^{(t+1)} = x^i \,|\, x^{(t)} = x^j)\ \mathbb{P}(x^{(t)} = x^j).$$

This equation governs how the probabilities $\mathbb{P}(x^{(t)} = x^i)$ change with time t. Let's do the calculations for the Whack-the-mole example. Say the mole was at hole x^1 at the beginning. So the probability distribution of its presence

$$\pi^{(t)} = \begin{bmatrix} \mathbb{P}(x^{(t)} = x^1) \\ \mathbb{P}(x^{(t)} = x^2) \\ \mathbb{P}(x^{(t)} = x^3) \end{bmatrix}$$

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 a steady state distribution. The numbers $\mathbb{P}(x^{(t+1)} = x^i)$ stop changing with time. We can

compute this steady state distribution by writing

$$\pi = P^\top \pi.$$

Does such a π exist? If a Markov chain is irreducible and recurrent, then a π always exists and it is also unique. Because of the above equation, we can also compute the π given a transition matrix P: the steady-state distribution is the (right-)eigenvector of the matrix $P^\top$ corresponding to the eigenvalue 1.

Example 115. 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 ensures 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. We are therefore guaranteed that a steady state distribution exists. In this case it is

$$\pi^1 = 0.5\pi^1 + 0.4\pi^2$$
$$\pi^2 = 0.5\pi^1 + 0.6\pi^2.$$

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

$$\pi^1 = 4/9 \quad \pi^2 = 5/9.$$

Remark 116 (Time spent at a given state)**.** http://setosa.io/ev/markov-chains

14.1.3 Markov process for SGD

(picture of a discrete-state, discrete-time version of SGD)

The updates of SGD are given by

$$x^{(t+1)} = x^{(t)} - \alpha \nabla f_{\omega_t}(x^{(t)}).$$

The iterate $x^{(t+1)}$ is independent of the previous iterate $x^{(t-1)}$ given the current iterate $x^{(t)}$.

Let us imagine a Markov chain with a trajectory $X^{(0)}, X^{(1)}, \ldots, X^{(t)}$. It proceeds as follows:

$$\begin{aligned} X^{(0)} &= X_0 \\ X^{(t+1)} &\sim \mathbb{P}(X^{(t+1)}|X^{(t)}). \end{aligned}$$

This trajectory depends both on the initial condition X_0 and the sample from the transition kernel $\mathbb{P}(\cdot \mid X^{(t)})$. What is the transition kernel

$$\mathbb{P}\left(x^{(t+1)}|x^{(t)}\right)$$

for SGD? If you take the expectation with respect to ω_t conditioned on $x^{(t)}$ in the update equation we have

$$\mathop{\mathbb{E}}_{\omega_t}\left[x^{(t+1)} \mid x^{(t)}\right] = x^{(t)} - \alpha \nabla f(x^{(t)}),$$

as expected the update of SGD is the update of gradient descent in expectation. What is the second moment?

$$\begin{aligned} \mathop{\mathrm{Var}}_{\omega_t}\left(x^{(t+1)} \mid x^{(t)}\right) &= \mathop{\mathrm{Var}}_{\omega_t}\left(x^{(t+1)} - x^{(t)} \mid x^{(t)}\right) \\ &= \mathop{\mathrm{Var}}_{\omega_t}\left(-\alpha \nabla f_\omega(x^{(t)})\right) \\ &= \alpha^2 \mathop{\mathbb{E}}_{\omega}\left[\left(\nabla f_\omega(x^{(t)}) - \nabla f(x^{(t)})\right)\left(\nabla f_\omega(x^{(t)}) - \nabla f(x^{(t)})\right)^\top\right]. \end{aligned}$$

Again as expected the variance of the stochastic update is proportional to α^2. The standard deviation is this proportional to α and we have seen this before: there is a factor of α in the first term for the expression of the convergence rate of SGD.

Assumption 117. The function $f(x)$ is smooth.

Assumption 118. We are sampling with replacement, so the two gradients ∇f_ω and $\nabla f_{\omega'}$ are independent, i.e.,

$$\mathop{\mathbb{E}}_{\omega,\omega'}\left[\left(\nabla f_\omega(x^{(t)}) - \nabla f(x^{(t)})\right)\left(\nabla f_{\omega'}(x^{(t)}) - \nabla f(x^{(t)})\right)^\top\right] = 0.$$

Using the second assumption above we can obtain the variance for mini-batch SGD as

$$\begin{aligned}
&\text{Var}\left(x^{(t+1)} - x^{(t)} \mid x^{(t)}\right) \\
&= \alpha^2 \underset{\omega^1,\ldots,\omega^{\mathscr{b}}}{\text{Var}}\left[\frac{1}{\mathscr{b}}\sum_{i=1}^{\mathscr{b}} \nabla f_{\omega^i}(x^{(t)})\right] \\
&= \frac{\alpha^2}{\mathscr{b}^2}\sum_{i=1}^{\mathscr{b}} \text{Var}(\nabla f_{\omega^i}(x^{(t)})) \\
&= \frac{\alpha^2}{\mathscr{b}} \text{Var}(\nabla f_{\omega}(x^{(t)})).
\end{aligned}$$

The last equality follows because we are sampling with replacement, each of the ω^i are identically distributed.

Next we will assume something quite brutal. We will assume that the matrix $\text{Var}(\nabla f_\omega(x^{(t)}))$ does not depend on $x^{(t)}$, and moreover, that it is simply identity.

Assumption 119. The variance of the SGD update at time t is such that

$$\text{Var}\left(x^{(t+1)} - x^{(t)} \mid x^{(t)}\right) = \frac{\alpha^2}{\mathscr{b}} I_{d\times d}.$$

We are now ready to *approximate* the updates of SGD.

$$\begin{aligned}
X^{(t+1)} &= \underbrace{X^{(t)} - \alpha\nabla f(X^{(t)})}_{\text{mean}} + \underbrace{\text{noise}}_{\text{standard deviation}} \\
&= X^{(t)} - \alpha\nabla f(X^{(t)}) + \sqrt{\frac{\alpha}{\mathscr{b}}}\,\zeta^{(t)}.
\end{aligned} \tag{14.1}$$

where $\zeta^{(t)} \sim N(0, \alpha I_{d\times d})$ is a Gaussian random variable which is zero mean and variance α.

Remark 120. The above equation is a discrete-time equation but the state of the Markov chain $X^{(t)}$ is no longer finite. Simulating this equation is easy. Notice that if we have a gradient flow

$$\dot{X} = -\nabla f(X)$$

we simulate it by discretizing time as

$$x^{(t+1)} = x^{(t)} - \alpha\nabla f(x^{(t)}).$$

We can think of the step-size α in our standard gradient descent update as the discretization interval of the time variable. We can also simulate this stochastic update equation by taking

the full-gradient $\nabla f(X^{(t)})$ and adding Gaussian noise to it of variance α^2/ℓ. This is not exactly SGD but only a model for it; the difference is primarily due to Assumption 119.

Lecture 15

Gibbs distribution

Reading

- "Investigations on the theory of the Brownian movement" by Einstein (1956) at http://www.maths.usyd.edu.au/u/UG/SM/MATH3075/r/Einstein_1905.pdf

---

15.1 Gibbs distribution

We now have a model for SGD as a discrete-time Markov chain. What is the steady-state distribution of this Markov chain? The answer is given by what is known as the Gibbs distribution. If we have a Markov chain where each successive update is given by

$$X^{(t+1)} = X^{(t)} - \alpha \nabla f(X^{(t)}) + \sqrt{\frac{2}{\beta}}\, \zeta^{(t)} \tag{15.1}$$

and $\zeta^{(t)}$ is a Gaussian random variable with mean zero and variance αI, the Gibbs distribution is

$$\rho^{\infty}(x) := \lim_{t \to \infty} \mathbb{P}(X^{(t+1)} = x) = \frac{1}{Z(\beta)} e^{-\beta f(x)}.$$

The normalization constant $Z(\beta)$ ensures that the Gibbs distribution is a legitimate probability distribution, i.e., $\int \rho^{\infty}(x)\, \mathrm{d}x = 1$. We therefore have

$$Z(\beta) = \int e^{-\beta f(x)}\, \mathrm{d}x.$$

This distribution exists uniquely under certain technical conditions on $f(x)$ which we will

implicitly assume. None of these conditions however require that $f(x)$ be convex. For our model of SGD notice that we have

$$\beta^{-1} = \frac{\alpha}{2\cancel{b}}.$$

(picture of Gibbs distribution)

Remark 121. Let us list a few properties of the Gibbs distribution that are apparent simply by looking at the formula.

- The probability that the iterates of SGD are found at a location x is proportional to $e^{-\beta f(x)}$. If the training loss $f(x)$ 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, i.e., the limit $t \to \infty$ is achieved, 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 $f(x)$ is convex. However this statement does require the assumption that the steps-size α of SGD does not go to zero.

- The term β^{-1} is quite common in physics where it is called the "temperature". The temperature $\beta^{-1} = \frac{\alpha}{\cancel{b}}$ governs fundamentally how the Gibbs distribution looks. Higher the temperature, more the noise in the iterates and vice-versa. As you can imagine, if there is lots of noise in the SGD updates, if the learning rate α is large or the batch-size $\cancel{b}$ is small, it is easy for SGD to jump over hills. This is the reason why the Gibbs distribution will be spread around the entire energy landscape at high temperatures. The Gibbs distribution is essentially uniform over the entire state-space and does not care what the loss $f(x)$ at a location is in this case. On the other hand, if the temperature is very small, the Gibbs distribution cares a lot about the training loss and the probability of finding SGD at other places diminishes. In particular, if $\beta \to \infty$, the Gibbs distribution only puts non-zero probability on the global minima of the loss function $f(x)$.

- Written in another way, if we want the Gibbs distribution to remain the same we should ensure that

  $$\beta^{-1} = \frac{\alpha}{2\cancel{b}} \text{is a constant.}$$

  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.

- We have achieved something remarkable by looking at the Gibbs distribution. We have an algorithm to find the global minimum of a non-convex loss function.

  - Start from some initial condition x^0

- Take lots of steps of SGD with some fixed step-size α until SGD equilibriates
- Reduce the step-size α, and take lots of steps of SGD again until it equilibriates;
- Repeat the previous step

This is a formal algorithm but it will converge to the global minimum of a non-convex function $f(x)$. The catch of course is that at each step we have to wait until SGD equilibriates. Remember that the Gibbs distribution is defined as the limit as $t \to \infty$, it may take an inordinately long amount of time to SGD to equilibriate. How much time does it take to equilibriate for a convex loss function?

15.1.1 A peek into stochastic differential equations

Consider the model for SGD that we wrote in (15.1) as

$$X^{(t+1)} = X^{(t)} - \alpha \nabla f(X^{(t)}) + \sqrt{2\beta^{-1}}\, \zeta^{(t)}$$

where $\zeta^{(t)} \sim N(0, \alpha I)$. The fact that the variance of ζ scales linearly with the step-size α is special. Consider a sequence of the form

$$\zeta^{(0)}, \zeta^{(0)} + \zeta^{(1)}, \ldots, \sum_{i=0}^{t} \zeta^{(k)}, \ldots$$

where each $\zeta^{(k)} \sim N(0, I\, \Delta t)$ and is i.i.d. As $\Delta t \to 0$, this sequence reaches a continuous-time limit which is called Brownian motion denoted by $B(t)$. The discrete-time increments, i.e., our $\zeta^{(t)}$ are then effectively the "derivative" of this continuous curve, this derivative is denoted as $\mathrm{d}B(t)$. This gives rise to the stochastic counterpart of the continuous-time flow which we saw earlier. The dynamics of SGD can be written as

$$\mathrm{d}X(t) = -\nabla f(X(t))\, \mathrm{d}t + \sqrt{2\beta^{-1}}\, \mathrm{d}B(t). \tag{15.2}$$

The above equation is called an Ito stochastic differential equation. It is written in the funny way above because the derivative $\frac{\mathrm{d}B(t)}{\mathrm{d}t}$ does not actually exist, Brownian motion $B(t)$ is a continuous function of time t but is not differentiable at any point. You can watch it here https://en.wikipedia.org/wiki/Wiener_process#/media/File Wiener_process_animated.gif.

Remark 122. Einstein's original paper where he mathematically predicted Brownian motion is today's assigned reading. You are invited to read the paper if you are interested in historial trivia, you do not need to understand/follow it. The first section is a prime example of how clever Einstein was as a politician and a scientist: "If what I predict in this paper can be actually observed under a standard microscope (and of course I saw it myself before writing

this paper!) then current understanding of thermodynamics has deep flaws. If my predictions are wrong then well we have deep problems with the kinetic theory of heat". The year 1905 was still a time when the existance of atoms was heavily debated upon. By being a staunch scientist and reporting his findings with as precise wording as he could, Einstein as a young researcher (he was 26 years old) did not have to pick sides in this very contentious debate. This paper also gives an estimate of Avogadro's number, which is the number of molecules in one mole of a gas.

In short, just as the continuous flow $\dot{X}(t) = -\nabla f(X(t))$ converges to x^* for deterministic optimization, if one takes the iterate $X(t)$ of the stochastic differential equation

$$\mathrm{d}X(t) = -\nabla f(X(t))\,\mathrm{d}t + \sqrt{2\beta^{-1}}\,\mathrm{d}B(t)$$

with the condition $X(0) = x_0$ for many different trajectories $X^1(t), X^2(t), \ldots, X^N(t)$, the distribution of all these iterates converges to the Gibbs distribution, e.g.,

$$\lim_{t\to\infty} \frac{1}{N}\sum_{k=1}^{N} X^k(t) \approx \mathbb{E}\left[X(t)\right] = \frac{1}{Z(\beta)}\int x e^{-\beta f(x)}\,\mathrm{d}x.$$

The most important fact to remember is that this happens even if the loss function $f(x)$ is not convex. The Gibbs distribution allows us to understand an infinite number of trajectories of a stochastic system without worrying about what happens to each one of them. You can now appreciate why it has roots in physics where one would like to understand the properties of lots of molecules of a gas in this room without worrying about the trajectories of each one of them.

15.2 A Lyapunov functional for SGD

15.2.1 Convergence of a Markov chain to the steady state

Definition 123 (Kullback-Leibler divergence between two probability distributions)**.** For two distributions $p(x)$ and $q(x)$ supported on a discrete set $\mathcal{X}$, the Kullback-Leibler divergence between them is given by

$$\mathrm{KL}(p \,||\, q) = \sum_{x\in\mathcal{X}} p(x)\log\frac{p(x)}{q(x)}. \tag{15.3}$$

This formula is well-defined only if for all x where $q(x) = 0$, we also have $p(x) = 0$. The KL divergence is a measure of the distance between two distributions. Notice that it is not

symmetric

$$\mathrm{KL}(q \,||\, p) = \sum_{x \in \mathcal{X}} q(x) \log \frac{q(x)}{p(x)} \quad \neq \mathrm{KL}(p \,||\, q).$$

Therefore the KL divergence is not a metric. It is always positive however which you can show using an application of Jensen's inequality.

(draw a picture of the two cases)

For distributions with continuous support, we integrate over the entire space $\mathcal{X}$ and define KL divergence as

$$\mathrm{KL}(p \,||\, q) = \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x.$$

We will next show a very powerful fact: the KL divergence is a Lyapunov functional for a Markov chain. Consider our Markov chain on a discrete set $\mathcal{X}$ with transition matrix P. If the Markov chain starts at some state π^0 let the distribution of the states at time t be π^t. We will assume that a steady-state distribution π^* exists. Let us also assume that a reverse transition matrix P^{rev} exists. For any distribution $\pi(\cdot)$ and states x, y this transition matrix satisfies the definition of conditional probability

$$\mathbb{P}(x^{(t+1)} = y | x^{(t)} = x) \, \mathbb{P}(x^{(t)} = x) \quad = \quad \mathbb{P}(x^{(t)} = x | x^{(t+1)} = y) \, \mathbb{P}(x^{(t+1)} = y).$$

In our notation, we have

$$P^{\text{rev}}_{xy} = \frac{P_{yx}\pi(y)}{\pi(x)} = \frac{P_{yx}\pi(y)}{\sum_y P_{yx}\pi(y)}.$$

Now consider

$$\begin{aligned}
\mathrm{KL}(\pi^* \,||\, \pi^{t+1}) &= \sum_x \pi^*(x)\ \log \frac{\pi^*(x)}{\pi^{t+1}(x)} \\
&= \sum_x \pi^*(x)\ \log \frac{\pi^*(x)}{\sum_y P_{yx}\ \pi^t(y)} \\
&= -\sum_x \pi^*(x)\ \log \frac{\sum_y P_{yx}\ \pi^t(y)}{\pi^*(x)} \\
&= -\sum_x \pi^*(x)\ \log \left(\sum_y P^{\text{rev}}_{xy}\ \frac{\pi^t(y)}{\pi^*(y)} \right) && \text{(substitute definition of } P^{\text{rev}} \text{ for distribution } \pi^*) \\
&\leq -\sum_x \pi^*(x)\ \sum_y P^{\text{rev}}_{xy}\ \log \frac{\pi^t(y)}{\pi^*(y)} && \text{(Jensen's inequality)} \\
&= \sum_y \sum_x P^{\text{rev}}_{xy}\ \pi^*(x)\ \log \frac{\pi^*(y)}{\pi^t(y)} && \text{(flip the negative sign, exchange sum)} \\
&= \sum_y \pi^*(y)\ \log \frac{\pi^*(y)}{\pi^t(y)} \\
&= \mathrm{KL}(\pi^* \,||\, \pi^t).
\end{aligned}$$

The distance to the steady-state distribution π^* decreases at each step of the Markov chain. The only condition required for this to happen is that we have a steady-state distribution for the Markov chain under consideration (barring a few technical conditions that state that $\pi^{t+1}(x)$ is not zero at any state x). This distance also decreases monotonically, Jensen's inequality is an equality for the function $\log$ if and only if all the entires in the summation are equal which can happen only if we are already at steady state (see the very first step of the derivation). We have found an equivalent of the Lyapunov function for Markov chains, it is simply the Kullback-Leibler divergence. A similar computation is true for the reverse KL divergence as well:

$$\mathrm{KL}(\pi^{t+1} \,||\, \pi^*) \leq \mathrm{KL}(\pi^t \,||\, \pi^*).$$

Definition 124. The KL divergence with respect to a uniform distribution is the negative entropy up to a constant

$$\begin{aligned}
\mathrm{KL}(p \,||\, \text{uniform}) &= \sum_x p(x)\ \log p(x) - \log |\mathcal{X}| \\
&=: -\mathrm{H}(p) + \text{constant}.
\end{aligned}$$

In continuous spaces, we write

$$\begin{aligned}\mathrm{KL}(p \,||\, \text{uniform}) &= \int_{\mathcal{X}} p(x) \log p(x) \, \mathrm{d}x + \text{constant that depends on } \mathcal{X}. \\ &=: -\mathrm{H}(p) + \text{constant}.\end{aligned}$$

To reiterate, the constant above does not depend on p.

15.2.2 Splitting the KL divergence

Let us now imagine a Gibbs distribution which takes values on a discrete state-spaces (*this* is actually called the Boltzmann distribution). This would be the steady state distribution of a Markov chain wherein the distribution

$$X^{(t+1)} = x^j \mid X^{(t)} = x^i = \text{Normal}\left(-\left(f(x^i) - f(x^j)\right),\, 2\beta^{-1} I\right)$$

Notice that expression $-\left(f(x^i) - f(x^j)\right)$ is similar to the gradient $\nabla f(x^i)$. If the loss $f(x^j)$ is smaller than the loss $f(x^j)$ the Markov chain makes that transition with large probability. Similar to our earlier exposition on the Gibbs distribution, the steady-state distribution of this Markov chain is given by

$$\pi^*(x) = \frac{1}{Z(\beta)} \, e^{-\beta f(x)}.$$

We now have the following very interesting theorem.

Theorem 125 (Convergence of a discrete-time discrete-state Markov chain to its steady state)**.** *For an irreducible, positively recurrent Markov chain, the functional*

$$\begin{aligned}\mathrm{KL}(\pi^t \,||\, \pi^*) &= \sum_{x \in \mathcal{X}} \pi^t(x) \, f(x) - \beta^{-1} H(\pi^t) \\ &= \mathop{\mathbb{E}}_{x \sim \pi^t}[f(x)] - \beta^{-1} \mathrm{H}(\pi^t).\end{aligned} \tag{15.4}$$

decreases monotonically, i.e.,

$$\mathrm{KL}(\pi^{t+1} \,||\, \pi^*) \leq \mathrm{KL}(\pi^t \,||\, \pi^*).$$

Proof. We can expand as

$$\begin{aligned}\mathrm{KL}(\pi^t \,||\, \pi^*) &= -\mathrm{H}(\pi^t) - \sum_x \pi^t(x)\ \log \pi^*(x) \\ &= -\mathrm{H}(\pi^t) + \sum_x \beta\ \pi^t(x)\ f(x) + \log Z(\beta).\end{aligned}$$

We know the monotonic decrease from the calculation in the previous section. It is customary to write the factor of β on the entropy with an inverse instead of the energy term $\mathbb{E}_{x\sim\pi^t}[f(x)]$.

■

There is a continuous-time counterpart of this theorem which is among the most celebrated mathematical results of the 20th century with ties to mathematics of as far back as the French revolution (https://en.wikipedia.org/wiki/Gaspard_Monge) as well as ties to physics through Boltzmann's work on H-theroems (https://en.wikipedia.org/wiki/H-theorem).

Theorem 126 (Jordan et al. (1998))**.** *For a continuous-time stochastic differential equation (15.2), i.e., our model of SGD, if the distribution of the iterates at time t is denoted by $\mathbb{P}(X^{(t)} = x) = \rho^t$, the Jordan-Kinderlehrer-Otto (JKO) functional*

$$KL(\rho^t \,||\, \rho^\infty) = \mathop{\mathbb{E}}_{x\sim\rho^t}[f(x)] - \beta^{-1}\, H(\rho^t) \tag{15.5}$$

decreases monotonically. The steady-state distribution is given by

$$\rho^\infty(x) = \frac{1}{Z(\beta)} e^{-\beta f(x)}$$

and satisfies

$$\rho^\infty = \mathrm{argmin}_{\rho\in\mathcal{P}(\mathcal{X})} \left\{ \mathop{\mathbb{E}}_{x\sim\rho}[f(x)] - \beta^{-1}\, H(\rho) \right\}. \tag{15.6}$$

where $\mathcal{P}(\mathcal{X})$ is the set of all probability distributions supported on the set $\mathcal{X}$.

Remark 127. We next make a few remarks about this result. We will compare and constrast this theorem with what we have learnt for non-stochastic optimization.

- Functionals like (15.4) where we have the expected value of some energy $f(x)$ along with the entropy of a distribution $H(\rho^t)$ are called "free energies" in physics. This theorem says that under a very general set of conditions (really only the fact that the steady state distribution exists) the distribution of the iterates of SGD converges to the steady-state distribution as time goes to infinity.
- Notice that SGD is playing a balancing game between energy and entropy. The steady state distribution ρ^∞, i.e., the distribution of the iterates at time equal to infinity is

such that it minimizes a combination of the expected value of the loss $f(x)$ and likes to be as entropic as possible (roughly speaking, it spreads its probability mass all over the state-space).

- We saw the implicit regularization of gradient descent for over-parametrized least squares in Lecture 6. Gradient descent converges to solutions that have minimum ℓ_2 norm. The above theroem should be understood as the variational version of the same result. In the case of SGD, we do not (cannot) talk of one trajectory of SGD and where it converges, we always have to talk about the steady-state distribution. The steady-state distribution, i.e., "the solution", of SGD has high entropy just like gradient descent favors solutions that have low norm.

- The ratio
$$\beta^{-1} = \frac{\alpha}{2\mathscr{b}}$$
is the coefficient of the entropy term. If we are to preserve the properties of SGD, e.g., generalization performance, convergence speed etc. in practice, but the steady-state distribution more precisely, we should ensure that this constant is invariant. The JKO functional under our model only depends on the loss $f(x)$, the learning rate α and the batch-size $\mathscr{b}$. The theorem also gives us our first insight into the role of optimization in generalization.

(draw a picture of sharp minimum and a wide minimum, Gibbs distribution puts more mass on the wide)

- Think back to Problem Set 2 where you devised a technique to find a suitable learning rate given a generic image classification problem. If you have a bigger GPU and wanted to double the batch-size, the expression for β^{-1} suggests that you should also double the learning rate by the same factor.

- We know fairly well how gradient descent and stochastic gradient descent converge for convex functions $f(x)$. Today's result holds for non-convex functions also, we see the same style of result wherein monotonic improvements are made to certain quantities. In gradient descent these quantities were $\|x^{(t+1)} - x^*\|$ or $f(x^{(t+1)}) - f(x^*)$, for stochastic gradient descent showed that
$$\mathop{\mathbb{E}}_{\omega^{(1)},\ldots,\omega^{(t)}}[f(x^{(t+1)}) - f(x^{(t)})]$$

decreases monotonically. This theorem is a similar result we use the distribution

$$\rho^t(x) = \mathbb{P}(X^{(t)} = x)$$

instead of the individual iterate $X^{(t)}$ and use the functional

$$\mathrm{KL}(\rho^t \,||\, \rho^\infty).$$

(draw a picture of $\mathcal{P}(\mathcal{X})$ with ρ^t and ρ^∞)

- We commented that momentum does not seem to help one single trajectory of SGD, it does not seem to accelerate the convergence. However, one can show using our model for SGD that momentum does accelerate convergence of the entire distribution ρ^t to ρ^∞.

Lecture 16

Linear neural networks, stable manifold theorem, linear residual networks

Reading

- Goodfellow Chapter 13
- "Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima" by Baldi and Hornik (1989)
- "Identity matters in deep learning" by Hardt and Ma (2016)

16.1 Introduction

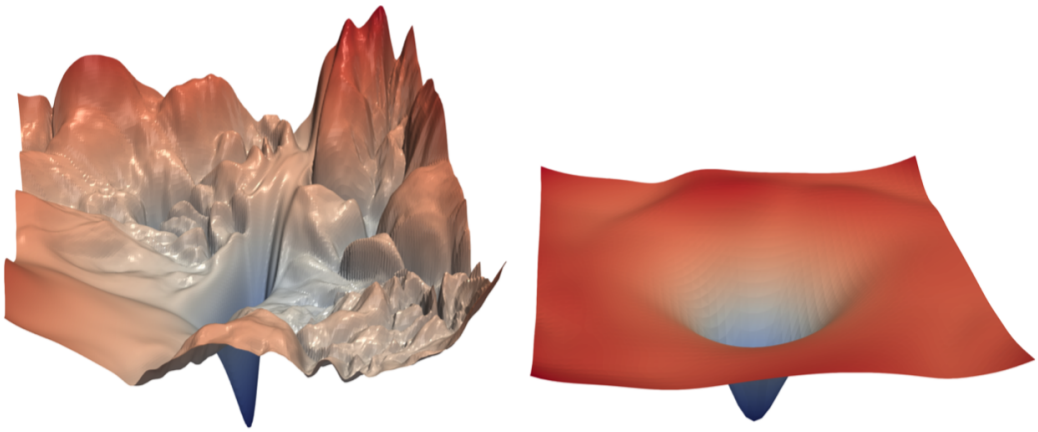
We have finished two main modules in the class so far. The first one introduced the mechanics of using deep neural networks, what are the architectures, what are standard regularizers, which algorithms we use to train them etc. The second module focused on optimization algorithms, namely, gradient descent, stochastic gradient descent (SGD), their accelerated variants and convergence properties of these algorithms. The second module had two flavors: (i) finite-time convergence-rate results for convex functions, and (ii) asymptotic results about the distribution of the iterates of SGD for both convex and non-convex functions.

This is the beginning of the third module where we will try to understand the shape of the loss function of neural networks. We know that the loss function is not convex, but how

non-convex is it? The key questions we want to answer here are:

- how many global minima exist?
- how many local minima and saddle points exist?
- 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?
- what is the local geometry of the loss function?
- what is the global topology of the loss function?

Along the way we will study why SGD seems to train so efficiently on neural networks, get an insight into how generalization is related to the shape of the loss function, why architectures like residual networks are so good for training etc. 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

16.2 Introduction

There are a few key players in a non-convex optimization problem:

- Global minima are all points in the set

$$\{x : f(x) \leq f(y) \text{ for all } y \in \mathcal{X}\}.$$

Note that there may exist many different locations all with the same loss $f(x)$, they would all be global minima in this case. What is the Gibbs distribution for such a situation? (picture)

- Local minima are all points in the set

$$\left\{x : \nabla f(x) = 0, \nabla^2 f(x) \succeq 0 \text{ i.e., the Hessian is positive semi-definite}\right\}.$$

Note that the two conditions (i) first-order stationarity $\nabla f(x) = 0$ and (ii) positive semi-definiteness of the Hessian $\nabla^2 f(x) \succeq 0$ also have to be satisfied for all global minima.

- Critical points are all locations which satisfy only first order stationarity

$$\{x : \nabla f(x) = 0\}.$$

- Saddle points are critical points but which are neither local minima not local maxima

$$\left\{x : \nabla f(x) = 0, \nabla^2 f(x) \text{ is neither positive nor negative semi-definite}\right\}.$$

There are also things called "strict saddle points" where the Hessian is such that it either has all positive eigenvalues or at least one strictly negative eigenvalue.

Non-convex functions can have all the above players which affect the process of training a neural network. Our objective in this module is to paint a mental picture of what the loss function looks like on the basis of a number of theoretical and practical results.

16.3 Deep linear network have no bad local minima

Let us consider the simplest case of linear neural networks first. We will consider a two-layer neural network which takes in inputs x_i and predicts outputs y_i. For simplicity in our calculations we will consider the case when both

$$x_i, y_i \in \mathbb{R}^d.$$

This network predicts vectors in $\mathbb{R}^d$ as the output given vectors as the input. We will also use a simpler loss, the ℓ_2 regression loss.

$$\ell = \frac{1}{n}\sum_{i=1}^{n} \|y_i - AB\, x_i\|_2^2 \tag{16.1}$$

The matrices A, B are the weights of the neural network. 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. We are interested in finding A and B.

(develop Baldi & Hornik's paper)

Remark 128. 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.

16.3.1 Gradient descent does not converge to strict saddle points

16.3.2 Deep linear residual networks have no bad critical points

This is Theorem 2.1 of Hardt and Ma (2016). Write down the model for identity connections and some linear ground-truth data generator $Rx + \xi$. Can show that if

$$\gamma := \max\left\{\log \sigma_{\max}(R), \log \sigma_{\min}(R)\right\}$$

then there exists a global minimum solution A^* for the population loss with norm

$$|||A^*||| \leq \frac{2(\sqrt{\pi} + \sqrt{3\gamma})^2}{l}.$$

In other words, as the depth increases, low norm solutions for the population loss exist. One can can restrict the attention to the set of critical points near the global minimum

$$\mathcal{B}_\tau = \left\{A : |||A||| \leq \tau\right\}.$$

Then for any $\tau < 1$ there are no critical points of the population loss in the set $\mathcal{B}_\tau$.

Lecture 17

Linear neural networks, stable manifold theorem, linear residual networks

Reading

- Goodfellow Chapter 13
- Bishop Section 12.1, 12.4.2
- "Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima" by Baldi and Hornik (1989)
- "Identity matters in deep learning" by Hardt and Ma (2016)

17.1 Deep linear network have no bad local minima

Let us consider the simplest case of linear neural networks first. We will consider a two-layer neural network which takes in inputs x_i and predicts outputs y_i. For simplicity in our calculations we will consider the case when both

$$x_i, y_i \in \mathbb{R}^d.$$

This network predicts vectors in $\mathbb{R}^d$ as the output given vectors as the input. We will also use a simpler loss, the ℓ_2 regression loss.

$$\ell(A, B) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - AB\, x_i\|_2^2 \tag{17.1}$$

The matrices A, B are the weights of the neural network. 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. We are interested in finding A and B.

We next develop Baldi & Hornik's paper.

- Least squares solution for
$$\min_L \frac{1}{2n} \|Y - LX\|^2$$
is
$$L = \Sigma_{yx}\, \Sigma_{xx}^{-1}$$
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 "covaraiance matrices". These show up all the time in the study of neural networks. It will be useful to also keep track of the case when both the labels and the data are the same, i.e., we are learning a network to regerate the image x_i itself. This is called the "auto-associative" cases or, these days, the auto-encoder case. In this case, Σ_{yx} is the same as Σ_{xx}.

- There is no unique solution to this optimization problem.

- The loss is not convex. But if we fix either A or B, it is convex in the other.

- The rank of $W = AB$ is at most p.

- 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 project is $\hat{v} \in \text{span}\,\{m_1, \dots, m_n\}$, we know that it has to satisfy
$$(v - \hat{v}) \perp m_k \text{ for all } k \le n \quad \Rightarrow \quad m_k^\top (v - \hat{v}) = 0.$$

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.

$$\hat{v} = \underbrace{M(M^\top M)^{-1} M^\top}_{\text{projection matrix}} v =: P_M v.$$

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: $P^\top = P$. To see this, consider two vectors v, w and the dot products

$$(P_M v) \cdot w, \quad \text{and} \quad v \cdot (P_M w).$$

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

$$(Pv) \cdot w = v \cdot (Pw) = (Pv) \cdot (Pw).$$

We can now rewrite the first equality to obtain

$$P = P^\top.$$

Question 129. What are Σ_{xx} and Σ_{yx} for MNIST?

Fact 130. For any A, the function $\ell(A, B)$ is convex in B and has a minimum at

$$A^\top A \hat{B}(A) \Sigma_{xx} = A^\top \Sigma_{yx}.$$

If Σ_{xx} is invertible and A is full-rank, then we can write

$$\hat{B}(A) = (A^\top A)^{-1} A^\top \Sigma_{yx} \Sigma_{xx}^{-1}.$$

For the auto-encoder we have

$$\hat{B}(A) = (A^\top A)^{-1} A^\top.$$

Fact 131. 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, we have a unique minimum

$$\hat{A}(B) = \Sigma_{yx}\, B^\top \left(B\Sigma_{xx}B^\top\right)^{-1}.$$

Fact 132. We have a critical point, i.e., the gradient of the loss function in both A and B is identically zero when

$$W = AB = P_A\, \Sigma_{yx}\Sigma_{xx}^{-1}.$$

You can show that the matrix A for the critical point satisfies

$$P_A\Sigma = \Sigma P_A = P_A\Sigma P_A.$$

where

$$\Sigma = \Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy}.$$

Fact 133. This is an important fact and understanding it is the key to getting an insight into almost all results on the shape of the loss function. 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

$$\begin{aligned} A &= U_{\mathcal{I}}\, C \\ B &= C^{-1}U_{\mathcal{I}}^\top\, \Sigma_{yx}\Sigma_{xx}^{-1}. \end{aligned} \tag{17.2}$$

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 Fact 4. This is a special form. The solution $W = AB$ in a two-layerd linear network is given by our original ordinary least squares regression matrix projected followed by an orthogonal projection onto the subspace spanned by p eigenvectors

of Σ.

Fact 134. The loss function $\ell(A, B)$ at a critical point is

$$\text{trace}(\Sigma_{yy}) - \sum_{i_k \in \mathcal{I}} \lambda_{i_k}. \tag{17.3}$$

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.

Fact 135. (Saddle points) 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 hiddle 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 136. (No local minima in a deep linear network, or 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".

Remark 137 (The global minimum is not unique)**.** This is perhaps the most important point of this lecture. The loss at the global minimum is unique, not the global minimum. 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.

Remark 138 (All the previous results are true for deep 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.

Question 139. Let the gradient of the loss function with respect to A be $\nabla_A \ell$. The dot product of the gradient along the eigenvectors of the matrix C is zero. This is quite intuitive

because elements of C do not play any role in creating the loss function. If you imagine the yourself at the global minimum for a particular C, no matter what you do, you cannot move around. There is simply no gradient, neither the full-gradient, not even the stochastic gradient. The loss function of a deep network is quite pathological this way. It looks conceptually like the the basin of the Colorado river

The connected region are the symmetries C. The basin is the entire "flat" global minimum. In fact, you can show that for linear networks there is only such river basin in the entire energy landscape, all global minima are connected to each other (if only you could travel across them, you could simply list them all.) More precisely, if we define a sub-level set of energy ϵ, i.e., all parameters such that

$$\{A, B : \ell(A, B) \leq \epsilon\}$$

it is always a connected set for any value of ϵ.

Question 140. We know that Nesterov's momentum does not lead to acceleration in SGD. And yet, you always see a noticeable speedup in training if you switch on Nesterov's momentum. Does the picture of the connected level set suggest a possible answer to this conundrum?

The fact that the sub-epsilon level set is connected is not true for non-linear networks, my conjecture for how the energy landscape might look based on some ideas in physics is as follows.

We know mathematically that non-linearities break the river in multiple places. So the energy landscape near the global minimum energy level should look more like a bunch of lakes than a contiguous river.

Remark 141. Let us draw a cartoon picture of the loss function to solidify all these new results. It helps to think of the shape of the loss function after quotienting out all these symmetries.

Question 142. How would the Gibbs distribution look for this network?

Question 143. What does weight-decay or ℓ_2 regularization of the weights do to the energy landscape of deep linear networks?

17.2 Gradient descent does not converge to strict saddle points

Let us a consider a quadratic loss function $f(x) = \frac{1}{2}x^\top Hx$ where the Hessian matrix is diagonal $H = \text{diag}(\lambda_1, \ldots, \lambda_N)$. Let's say the first k eigenvalues are positive and the rest are negative. This is therefore not a convex loss function but $x^* = 0$ is the unique critical point. If we initialize gradient descent at some x_0 we get

$$x^{(t+1)} = (I - \alpha H)\, x^{(t)} = (I - \alpha H)^t\, x_0$$

The eigenvectors of H are the canonical eigenvectors $e_1, e_2, \ldots, e_N$ because it is diagonal. We can therefore write the above equation as

$$x^{(t+1)} = \sum_{i=1}^{N} (1 - \alpha\lambda_i)^t \, e_i^\top x_0 \, e_i.$$

We know that we need a step-size $\alpha < 1/L$ for gradient descent to converge and the L for this example is simply the largest eigenvalue of the Hessian. We thus have

$$(1 - \alpha\lambda_i) < 1 \text{ for all } i \leq k$$

and

$$(1 - \alpha\lambda_i) > 1 \text{ for all } i > k.$$

We therefore get a result that if

$$x_0 \in E_s := \text{span}\,\{e_1, \ldots, e_k\}$$

gradient descent converges to the saddle point $x^* = 0$ since $(1 - \alpha\lambda_i)^t \to 0$. However, if x_0 has a component outside this span, then gradient descent diverges to $-\infty$. The set E_s is called the global attractive set of the critical point x^*. Now imagine if we picked the initial state x_0 uniformly randomly in the entire $\mathbb{R}^N$ dimensional space. The probability of this initial condition being inside the set E_s is zero, there will always be components aligned with other $N - k$ canonical eigenvectors. Gradient descent will therefore never converge to this saddle point with probability one.

The above situation is an example of a powerful theorem in the study of dynamical systems called the stable manifold theorem (see https://www.cds.caltech.edu/ murray/wiki/images/b/ba/Cds140a-wi11-Week4Notes.pdf if you want to read more). The updates of gradient descent are an iterative map

$$x^{(t+1)} = \phi(x^{(t)}).$$

Note that $\phi : \mathbb{R}^N \to \mathbb{R}^N$ and $\phi = (I - \alpha H)$ for our previous example. Given such a mapping ϕ we define an unstable fixed point as

$$A_\phi = \left\{ x : \phi(x) = x; \; \max_i |\lambda(D\phi(x))| > 1 \right\}$$

where the notation $D\phi(x)$ is the linearization of ϕ at x. For a general function $f(x)$ we have $D\phi(x) = I - \alpha\nabla^2 f(x)$ for gradient descent. The above set therefore consist of all fixed points of the map $x = \phi(x)$ with eigenvalues that are larger than 1 in magnitude.

Theorem 144 (Stable Manifold Theorem)**.** *For a continuous map ϕ such that* $\det(D\phi(x)) \neq$

0 *for any x, the set of initial conditions that converge to an unstable fixed point has measure zero:*

$$\mu\left(\left\{x_0 : \lim_{t\to\infty} \phi^t(x_0) \in A_\phi\right\}\right) = 0.$$

This is a fancy way of saying that even gradient descent will never converge to saddle points. Stochastic gradient descent is a noisy version of gradient descent, so even if we have picked an initial condition in span$(e_1, \ldots, e_k)$ in our example, we are guaranteed that SGD will not converge to any of the exponentially many saddle points in the energy landscape.

Remark 145. Gradient descent does not converge to saddle points with probability 1 but it can slow down arbitrarily badly near saddle points. Imagine if one of the eigenvalues λ_i for the Hessian in our previous example was very close to zero. The successive iterations for that eigenvalue $(1 - \alpha\lambda_i)^t$ would essentially be the identity map and we would never make progress in this direction.

17.3 Deep linear residual networks have no bad critical points

We saw that gradient descent does not converge to saddle points. But it can slow down near them, and this slowdown can be quite drastic. Here is another way to get rid of the critical points in energy landscape. This is Theorem 2.1 of Hardt and Ma (2016).

Write down the model for identity connections and some linear ground-truth data generated using the model $y = Rx + \xi$ where ξ is noise. Can show that if

$$\gamma := \max\left\{\log \sigma_{\max}(R), \log \sigma_{\min}(R)\right\}$$

then there exists a global minimum solution A^* for the population loss with norm

$$|||A^*||| \leq \frac{2(\sqrt{\pi} + \sqrt{3\gamma})^2}{l}.$$

In other words, as the depth increases, low norm solutions for the population loss exist. One can can restrict the attention to the set of critical points near the global minimum

$$\mathcal{B}_\tau = \left\{A : |||A||| \leq \tau\right\}.$$

Then for any $\tau < 1$ there are no critical points of the population loss in the set $\mathcal{B}_\tau$.

(draw a picture)

Lecture 18

Stable manifold theorem, linear residual networks, shape of local minima

Reading

- "Identity matters in deep learning" by Hardt and Ma (2016)

- "Entropy-SGD: Biasing gradient descent into wide valleys" by Chaudhari et al. (2016)

The loss function of a deep network looks conceptually like the the basin of the Colorado river

The connected region are the symmetries C. The basin is the entire "flat" global minimum. In fact, you can show that for linear networks there is only such river basin in the entire energy landscape, all global minima are connected to each other (if only you could travel across them, you could simply list them all them.) More precisely, if we define a sub-level set of energy ϵ, i.e., all parameters such that

$$\{A, B : \ell(A, B) \leq \epsilon\}$$

it is always a connected set for any value of ϵ.

Question 146. We know that Nesterov's momentum does not lead to acceleration in SGD. And yet, you always see a noticeable speedup in training if you switch on Nesterov's momentum. Does the picture of the connected level set suggest a possible answer to this conundrum?

The fact that the sub-epsilon level set is connected is not true for non-linear networks, my conjecture for how the energy landscape might look based on some ideas in physics is as follows.

 .

We know mathematically that non-linearities break the river in multiple places. So the energy landscape near the global minimum energy level should look more like a bunch of lakes than a contiguous river.

Remark 147. Let us draw a cartoon picture of the loss function to solidify all these new results. It helps to think of the shape of the loss function after quotienting out all these symmetries.

Question 148. How would the Gibbs distribution look for this network?

Question 149. What does weight-decay or ℓ_2 regularization of the weights do to the energy landscape of deep linear networks?

18.1 Gradient descent does not converge to strict saddle points

Let us a consider a quadratic loss function $f(x) = \frac{1}{2}x^\top Hx$ where the Hessian matrix is diagonal $H = \text{diag}(\lambda_1, \ldots, \lambda_N)$. Let's say the first k eigenvalues are positive and the rest are negative. This is therefore not a convex loss function but $x^* = 0$ is the unique critical point. If we initialize gradient descent at some x_0 we get

$$x^{(t+1)} = (I - \alpha H)\, x^{(t)} = (I - \alpha H)^t\, x_0$$

The eigenvectors of H are the canonical eigenvectors $e_1, e_2, \ldots, e_N$ because it is diagonal. We can therefore write the above equation as

$$x^{(t+1)} = \sum_{i=1}^{N} (1 - \alpha\lambda_i)^t \, e_i^\top x_0 \, e_i.$$

We know that we need a step-size $\alpha < 1/L$ for gradient descent to converge and the L for this example is simply the largest eigenvalue of the Hessian. We thus have

$$(1 - \alpha\lambda_i) < 1 \text{ for all } i \leq k$$

and

$$(1 - \alpha\lambda_i) > 1 \text{ for all } i > k.$$

We therefore get a result that if

$$x_0 \in E_s := \text{span}\,\{e_1, \ldots, e_k\}$$

gradient descent converges to the saddle point $x^* = 0$ since $(1 - \alpha\lambda_i)^t \to 0$. However, if x_0 has a component outside this span, then gradient descent diverges to $-\infty$. The set E_s is called the global attractive set of the critical point x^*. Now imagine if we picked the initial state x_0 uniformly randomly in the entire $\mathbb{R}^N$ dimensional space. The probability of this initial condition being inside the set E_s is zero, there will always be components aligned with other $N - k$ canonical eigenvectors. Gradient descent will therefore never converge to this saddle point with probability one.

The above situation is an example of a powerful theorem in the study of dynamical systems called the stable manifold theorem (see https://www.cds.caltech.edu/ murray/wiki/images/b/ba/Cds140a-wi11-Week4Notes.pdf if you want to read more). The updates of gradient descent are an iterative map

$$x^{(t+1)} = \phi(x^{(t)}).$$

Note that $\phi : \mathbb{R}^N \to \mathbb{R}^N$ and $\phi = (I - \alpha H)$ for our previous example. Given such a mapping ϕ we define an unstable fixed point as

$$A_\phi = \left\{ x : \phi(x) = x; \; \max_i |\lambda(D\phi(x))| > 1 \right\}$$

where the notation $D\phi(x)$ is the linearization of ϕ at x. For a general function $f(x)$ we have $D\phi(x) = I - \alpha\nabla^2 f(x)$ for gradient descent. The above set therefore consist of all fixed points of the map $x = \phi(x)$ with eigenvalues that are larger than 1 in magnitude.

Theorem 150 (Stable Manifold Theorem)**.** *For a continuous map ϕ such that* $\det(D\phi(x)) \neq$

0 *for any x, the set of initial conditions that converge to an unstable fixed point has measure zero:*

$$\mu\left(\left\{x_0 : \lim_{t\to\infty} \phi^t(x_0) \in A_\phi\right\}\right) = 0.$$

This is a fancy way of saying that even gradient descent will never converge to saddle points. Stochastic gradient descent is a noisy version of gradient descent, so even if we have picked an initial condition in span$(e_1, \ldots, e_k)$ in our example, we are guaranteed that SGD will not converge to any of the exponentially many saddle points in the energy landscape.

Remark 151. Gradient descent does not converge to saddle points with probability 1 but it can slow down arbitrarily badly near saddle points. Imagine if one of the eigenvalues λ_i for the Hessian in our previous example was very close to zero. The successive iterations for that eigenvalue $(1 - \alpha\lambda_i)^t$ would essentially be the identity map and we would never make progress in this direction.

Adding extra noise to gradient descent is a popular technique to accelerating the escape of GD out of the saddle points. If the norm of the gradient is smaller than some threshold, we can add extra noise:

$$x^{(t+1)} = x^{(t)} - \alpha\nabla f(x^{(t)}) + \xi$$

where $\xi \sim N(0, \sigma^2 I)$ for some chosen value $\sigma^2 > 0$. If we make sure that the noise being added is uniformly distributed in all the directions, i.e., the covariance of the noise is identity times some constant, we are guarnateed that even if SGD is approaching the saddle point along the slow dimension, it will be perturbed by such noise.

Remark 152 (How to find saddle points?)**.** Sometimes, we may wish to study saddle points in the energy landscape. How does one find such points? As we saw, it is hard to find saddle points using gradient descent. We have to use other algorithms to find saddle points. A prime example is Newton's method:

$$x^{(t+1)} = x^{(t)} - \left[\nabla^2 f(x^{(t)})\right]^{-1} \nabla f(x^{(t)}).$$

If the Hessian matrix is symmetric, we can do an eigendecomposition of the Hessian to write

$$\nabla^2 f(x^{(t)}) = PDP^\top.$$

for some orthonormal matrix P and a diagonal matrix D consisting of the eigenvalues of the Hessian. This gives

$$x^{(t+1)} = x^{(t)} - PD^{-1}P^\top \nabla f(x^{(t)}).$$

(draw a picture)

The scaling by the inverse of the eigenvalues D^{-1} causes Newton's method to move towards

the saddle point along the direction of the negative eigenvalue's eigenvector. Therefore we can use Newton's method to converge to a saddle point efficiently. How do we go about discovering all saddle points?

Remark 153 (A simple example of the "apparent" non-convexity of neural networks)**.** Consider the optimization problem

$$\min_{w} \left(w^2 - 1\right)^2$$

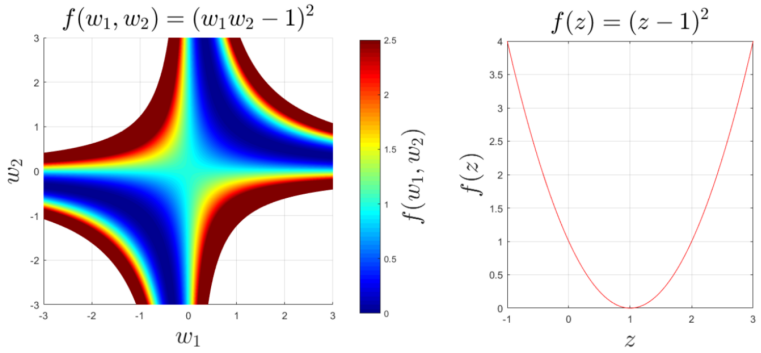
and the auxiliary problem

$$\min_{z \geq 0} (z-1)^2.$$

There are really just the same problems, the latter is convex and its global minimum ($z = 1$) corresponds to two global minima of the first problem, namely $z = \pm 1$. There is also a saddle point in between the two global minima in the first problem that connects them. This simple observation is at the heart of a lot of analyses in understanding the non-convexity of neural networks. Consider a related problem

$$\min_{w_1, w_2} \left(w_1 w_2 - 1\right)^2.$$

How do the global minima of this function look?



(a) Original Problem (b) Auxiliary Problem

18.2 Deep linear residual networks have no bad critical points

We saw that gradient descent does not converge to saddle points. But it can slow down near them, and this slowdown can be quite drastic. Here is another way to get rid of the critical points in energy landscape. This is Theorem 2.1 of Hardt and Ma (2016).

Write down the model for identity connections and some linear ground-truth data generated using the model $y = Rx + \xi$ where ξ is noise. Can show that if

$$\gamma := \max\left\{\log \sigma_{\max}(R), \log \sigma_{\min}(R)\right\}$$

then there exists a global minimum solution A^* for the population loss with norm

$$|||A^*||| \leq \frac{2(\sqrt{\pi} + \sqrt{3\gamma})^2}{l}.$$

In other words, as the depth increases, low norm solutions for the population loss exist. One can can restrict the attention to the set of critical points near the global minimum

$$\mathcal{B}_\tau = \left\{A : |||A||| \leq \tau\right\}.$$

Then for any $\tau < 1$ there are no critical points of the population loss in the set $\mathcal{B}_\tau$.

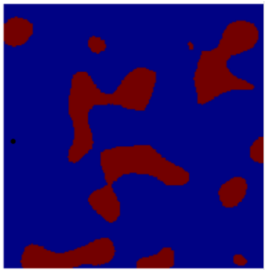
(draw a picture)

18.3 Some remarks about the connectedness of valleys

Given a function $f(x)$, a sub-epsilon level set is defined as the set

$$L_f(\epsilon) = \left\{x : f(x) \leq \epsilon\right\}.$$

Here's a picture:



A level set is connected if given any two points $x^1, x^2 \in L_f(\epsilon)$ we can find a continuous curve that connects x^1 and x^2 such that all points on the curve lie in $L_f(\epsilon)$. Connectedness of the level set is a topological property of the energy landscape. Linear neural networks have connected global minima, i.e., $L_f(0)$ is connected. They also have $L_f(\epsilon)$ connected for any ϵ. Linear residual networks have a similar structure. How pervasive is this property of connectedness?

- We know that regularization in general breaks these connected regions. For two-layer neural networks, one can show that adding weight decay does not destroy this connectedness property. This is however quite a special property of the ℓ_2 norm.

- Nonlinear models are generally disconnected. In the case of ReLU non-linearities with one hidden layer, we can show that if the dimensionality of the hidden layer is large enough, i.e., if the neural network is fat enough, the level set becomes more and more connected. More precisely, a path inside $L_f(\epsilon + \tau)$ exists between any two points $x^1, x^2 \in L_f(\tau)$ if

$$\epsilon \approx p^{-1/d}.$$

  In other words, one has to climb a tiny hill (the height of the hill becomes tinier if p is large) to go from any point to any other point in the level set. This has not been shown for deeper networks although it is expected that it is true Freeman and Bruna (2016); Venturi et al. (2018).

- Neural networks with polynomial activation functions lie somewhere in between linear neural networks and ones with ReLU non-linearities. We essentially use the kernel trick to understand this. If the hidden layers of the network are wide enough, we again have a connected global minimum. Roughly, the bounds indicate that the number of hidden neurons should be twice the dimensionality of the data.

$$p \geq 2d.$$

  Does this hold in practice? The answer is more or less yes for well-performing networks. The All-CNN network you used in your problem set has some layers that are wider than $d = 3 \times 32 \times 32 = 3072$.

- Roughly speaking, the way to understand the energy landscape consists of convexifying the problem and mapping the parameters of the neural network to some other space where we can use techniques from convexity, e.g., connectedness of the level sets at all energy levels (remember the over-parametrized one-dimensional example).

- This is what theory tells us. In practice, it is essentially impossible to detect local minima, it is hard to find a network which is "stuck at a local minimum". So be suspicious of any sentence in any paper (peer-reviewed or not) that says "we have an algorithm for jumping of local minima/we believe the competitor's algorithm does not work because it is stuck in local minima and ours is not, etc.".

- Among the many conjectures in deep learning is that these level sets are quite regular if the energy/loss is high. They are expected to become more and more irregular as we approach the global minimum. So effectively, if one were to draw a picture of the

energy landscape of a nonlinear deep neural network, it looks like lakes descending all the way down. The ones at the top are the largest, the ones at the bottom are the most disconnected.

- It is very important to remember that such a study of over-parametrization does not tell us anything about the generalization performance of the network. We are simply making statements about the training error.

Question 154. Now can you answer the question of why we use a staircase schedule for the learning rate during training? What happens if you drop the learning rate too quickly? What happens if you don't reduce the learning rate at all?

18.4 Geometry of the minima

We have a reasonable idea of the topology of the energy landscape now, i.e., which parts of the parameter space are connected to which parts, how does this connectivity change with respect to the loss at that location, etc. In this section, we will consider some models for how the minimum looks like locally, essentially we want to understand the shape of the lake. This will give an insight into generalization and we will also be able to exploit the understanding to come up with a better algorithm for training a neural network as compared to SGD.

18.4.1 Binary perceptron

We will not do any math in this sub-section, we only want to develop an intuitive understanding of the material here.

Let us consider a simple model: that of a binary perceptron. We are given data $x_i \in \{-1, 1\}^d$, labels $y_i \in \{-1, 1\}$ and would like to fit a one layer neural network with sign non-linearities $\text{sign}(x) = 1$ if $x > 0$ and $\text{sign}(x) = -1$ if $x < 0$. We want to fit discrete valued weights $w \in \{-1, 1\}^d$ that minimize the errors of the perceptron:

$$f(w) = \sum_{i=1}^{n} \mathbf{1}_{\hat{y}_i = y_i}$$

where

$$\hat{y}_i = \text{sign}(w^\top x_i).$$

Everything is discrete in this problem. Each weight is ± 1, each input neuron is ± 1 and so are the labels $y_i = \pm 1$. We cannot take the gradient of the loss function $f(w)$ with respect

to w because $f(\cdot)$ is not a continuous function of w. Such problems are called discrete optimization problems and are typically much harder than continuous optimization problems which we have seen till now.

Question 155. Can you suggest an algorithm to train a binary perceptron?

Question 156. Are there symmetries in the binary perceptron like those in deep linear networks?

There are two kinds of questions we are interested in asking for the binary perceptron:

- Given a dataset $\{x_i, y_i\}_{i=1,\ldots,n}$ how many different $w \in \{-1, 1\}^d$ can fit this dataset. This is the discrete analog of the size of the lake in the continuous-valued case.
- How different is each of the solution obtained above? Do some solutions generalize better than others?

In order to understand generalization, we need to first construct a model for the probability distribution that generates the dataset $\{x_i, y_i\}_{i=1,\ldots,n}$. Here's a powerful way to think of generalization. It is called the teacher-student model. We assume that nature, which generates the data, is also a neural network. Let us say that the neural network that nature uses has some weights w^*. Nature draws n vectors $x_i \in \{-1, 1\}^d$ uniformly randomly from this discrete space and feeds them through its network to get the true labels

$$y_i = \text{sign}({w^*}^\top x_i).$$

The data is random but the dataset is not random, the labels y_i are a function of the specific w^* that was chosen by nature

$$D = \left\{x_i, \text{sign}({w^*}^\top x_i)\right\}_{i=1,\ldots,n}.$$

We are the student, we would like to to fit a binary perceptron w on this dataset. Note that if we managed to fit $w = w^*$, we will get perfect generalization, any sample $x \in \{-1, 1\}^d$ will be labeled in exactly the same way by the teacher and the student.

Lecture 19

Binary Perceptron and Entropy-SGD

Reading

- "Entropy-SGD: Biasing gradient descent into wide valleys" by Chaudhari et al. (2016)

19.1 Geometry of the minima: Binary Perceptron

We have a reasonable idea of the topology of the energy landscape now, i.e., which parts of the parameter space are connected to which parts, how does this connectivity change with respect to the loss at that location, etc. In this section, we will consider some models for how the minimum looks like locally, essentially we want to understand the shape of the lake. This will give an insight into generalization and we will also be able to exploit the understanding to come up with a better algorithm for training a neural network as compared to SGD.

We only want to develop an intuitive understanding of the material in this sub-section.

Let us consider a simple model: that of a binary perceptron. We are given data $x_i \in \{-1, 1\}^d$, labels $y_i \in \{-1, 1\}$ and would like to fit a one layer neural network with sign non-linearities $\text{sign}(x) = 1$ if $x > 0$ and $\text{sign}(x) = -1$ if $x < 0$. We want to fit discrete valued weights $w \in \{-1, 1\}^d$ that minimize the errors of the perceptron:

$$f(w) = \sum_{i=1}^{n} \mathbf{1}_{\hat{y}_i \neq y_i}$$

where

$$\hat{y}_i = \text{sign}(w^\top x_i).$$

Everything is discrete in this problem. Each weight is ± 1, each input neuron is ± 1 and so are the labels $y_i = \pm 1$. We cannot take the gradient of the loss function $f(w)$ with respect to w because $f(\cdot)$ is not a continuous function of w. Such problems are called discrete optimization problems and are typically much harder than continuous optimization problems which we have seen till now.

Question 157. Can you suggest an algorithm to train a binary perceptron?

Question 158. Are there symmetries in the binary perceptron like those in deep linear networks?

There are two kinds of questions we are interested in asking for the binary perceptron:

- Given a dataset $\{x_i, y_i\}_{i=1,\dots,n}$ how many different $w \in \{-1, 1\}^d$ can fit this dataset. This is the discrete analog of the size of the lake in the continuous-valued case.
- How different is each of the solution obtained above? Do some solutions generalize better than others?

In order to understand generalization, we need to first construct a model for the probability distribution that generates the dataset $\{x_i, y_i\}_{i=1,\dots,n}$. Here's a powerful way to think of generalization. It is called the teacher-student model. We assume that nature, which generates the data, is also a neural network. Let us say that the neural network that nature uses has some weights w^*. Nature draws n vectors $x_i \in \{-1, 1\}^d$ uniformly randomly from this discrete space and feeds them through its network to get the true labels

$$y_i = \text{sign}({w^*}^\top x_i).$$

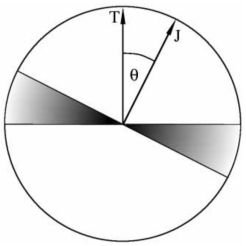
The data is random but the dataset is not random, the labels y_i are a function of the specific w^* that was chosen by nature

$$D = \left\{x_i, \text{sign}({w^*}^\top x_i)\right\}_{i=1,\dots,n}.$$

We are the student, we would like to to fit a binary perceptron w on this dataset. Note that if we managed to fit $w = w^*$, we will get perfect generalization, any sample $x \in \{-1, 1\}^d$ will be labeled in exactly the same way by the teacher and the student.

Remark 159 (Generalization error)**.** For this teacher-student setting and this simplistic model, we can obtain the generalization error analytically. The input-output relation of a perceptron has a simple interpretation: consider the hyper-plane $w^\top x = 0$ and let's study it in the input space. All inputs x that lie on the same side as w, i.e., their dot product is

positive are mapped to +1, all the other inputs are mapped to -1. A mis-classification by the student w occurs only if x is between the two planes w and w^*. Since we are drawing inputs uniformly randomly from the entire space, the generalization error is thus proportional to the angle between w and w^*.



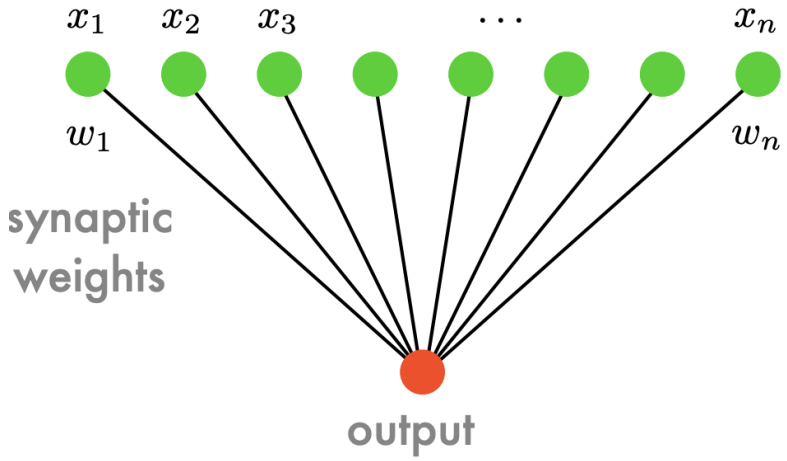
We have

$$\epsilon(w; w^*) = \frac{1}{\pi} \cos^{-1}\left(\frac{w^\top w^*}{d}\right).$$

Characterizing the energy landscape of the perceptron, say the global minima of the loss function amounts to counting all possible students which fit the training data created by the teacher exactly, e.g.,

$$\sum_{w \in \{-1,1\}^d} \prod_{i=1}^{n} \mathbf{1}_{\text{sign}(w^\top x_i) = \text{sign}(w^{*\top} x_i)}.$$

Notice that this is a complex expression: it is a sum over all possible weights, there are exactly 2^d possible students. For each of the students, we get a boolean variable of whether it classifies the entire training dataset correctly or not. There are techniques in statistical physics, known as replica theory, to approrimately perform such kind of computations. These techniques paint the following picture of the energy landscape.



This picture is drawn for a typical dataset of n samples generated by the teacher. The circle denotes all possible 2^d students. Note that this is a discrete Hamming space.

Each red dot in the circle denotes a student perceptron which fits the training dataset exactly, it has zero training error. There are many many such global minima. In constrast to the picture we saw for linear neural networks or even deep nonlinear two-layer ReLU neural networks, the global minima are not connected to each other. Turns out there are exponentially many global minima in the number of weights

$$\text{number of global minima} = \text{constant}^d.$$

All these global minima are far away from each other, i.e., their are isolated from each other by hills.

(draw a partial picture of the energy landscape) How would our training algorithm look for this energy landscape? (draw a picture of the training curve)

Turns out there are also some solutions to this problem that are not isolated. These solutions are such that if a particular w^a is a solution, almost all other perceptrons in its immediate neighborhood

$$\mathcal{N}(w^a;\ \gamma) = \{w :\ \text{dist}(w, w^a) < \gamma\}$$

for some $\gamma > 0$ also have zero training error on the same dataset. The distance $\text{dist}(w, w^a)$ is called the Hamming distance and it is a metric for discrete spaces. It measures the number of elements in which the two vectors w and w^a differ from each other.

$$\text{dist}(w, w^a) = \sum_{i=1}^{d} \mathbf{1}_{w_i \neq w_i^a}.$$

Such solutions are shown as blue clusters in the above picture. Each cluster is super tiny, its diameter ϵ is independent of the number of weights d, so if we make the perceptron bigger and bigger, the blue cluster of solutions becomes comparatively smaller and smaller. How many solutions does a cluster contain however? It is

$$\binom{d}{\epsilon} \approx d^\epsilon.$$

In other words, the number of solutions in the blue cluster is a tiny fraction of the total number of solutions for the perceptron; the former is a polynominal in d, the latter is exponential in d

$$\lim_{d\to\infty} \frac{d^\gamma}{\text{constant}^d} = 0.$$

How does the Gibbs distribution of this energy landscape look now? Since solutions in the

dense clusters are so small in number, they do not get any probability mass in the Gibbs distribution. The solutions in the dense clusters are sub-dominant, the exponentially many isolated solutions dominate the Gibbs distribution.

Remark 160 (Generalization error of dense clusters is better)**.** One can show (again using replica theory) that solutions in dense clusters have better generalization error. Dense clusters are isolated solutions perform the same on the training set and both have zero training error but the former have better error on new data created by the same teacher perceptron. While this is an intricate calculation, it is easy to appreciate that it could be true.

Observe that generalization is about understanding what happens to the loss when the input data is changed. Because of the special structure of the perceptron, if the input x in the training set changes slightly, the student perceptron can still classify this input correctly by changing the weights slightly due to the linear coupling $w^\top x$. Effectively, understanding the loss when the input changes is equivalent to understanding how the loss changes upon changes of the weights. If the loss of the perceptron does not change upon changing the weights w slightly, it also means that the loss does not change upon slight changes in the input x, i.e., that the perceptron generalizes.

The above verbal argument can be formalized and one can do calculations to show that dense clusters have better generalization properties.

This suggests that we should find dense clusters in the parameter space while training the perceptron. How do we find them? We know that the Gibbs distribution does not put any probability mass on these clusters. So SGD-like algorithms will not be able to find the dense clusters.

Remark 161. Can you think of some trick that finds dense clusters? Does adding noise in our SGD-like algorithm help find dense clusters for training the perceptron?

19.1.1 Changing the loss function for training

We have always imagined the loss function for training a neural network as a given. We use cross-entropy of the softmax outputs of the network to train because it is a good proxy for the classification error. The preceeding discussion of the perceptron suggests that we should invest in better loss functions to train neural networks if we want to achieve good generalization error in addition to simply fitting the training dataset.

Let us change the loss function for the perceptron to pay more attention to the dense clusters. Consider solving

$$\min_{w} \max_{w' \in \mathcal{N}(w,\gamma)} f(w') \tag{19.1}$$

which finds weights w such that any other w' in their γ-neighborhood $\mathcal{N}(w, \epsilon)$ has a small loss $f(w')$. How would we solve this problem iteratively?

Remark 162 (Metropolis-Hasting's algorithm, or our poor-man's version of it)**.**

This problem is similar to the following problem.

$$\begin{aligned}&\min_{w} \min_{w' \in \mathcal{N}(w, \sqrt{\gamma})} e^{-f(w')} \\ &\approx \min_{w} \sum_{w' \in \{-1,1\}^d} e^{-\frac{\text{dist}(w', w)}{2\gamma}} e^{-f(w')}.\end{aligned}$$

Note that we changed γ to $\sqrt{\gamma}$, it does not really matter because we will think of γ as a hyper-parameter. Let's take the logarithm of the loss function to make sure we don't have absurdly large numbers due to the large summation over the Hamming cube $\{-1, 1\}^d$.

$$\min_{w} \log \sum_{w' \in \{-1,1\}^d} e^{-\frac{\text{dist}(w', w)}{2\gamma}} e^{-f(w')}. \tag{19.2}$$

This loss function, namely

$$\log \sum_{w' \in \{-1,1\}^d} e^{-\frac{\text{dist}(w', w)}{2\gamma}} e^{-f(w')}$$

is called "local entropy" in physics because it is very related to the log-partition function of a certain Gibbs distribution.

Question 163. How is changing the loss function different from regularization?

Question 164. How would one train the binary perceptron using this new loss function?

19.1.2 Continuous-version of local entropy

Lecture 20

Langevin dynamics, Markov Chain Monte Carlo

Reading

- "Bayesian Learning via Stochastic Gradient Langevin Dynamics" by Welling and Teh (2011), at https://www.ics.uci.edu/w̃elling/publications/papers/stoclangevin_v6.pdf.
- http://arogozhnikov.github.io/2016/12/19/markov_chain_monte_carlo.html
- Bishop Chapter 11-11.2

Lecture 21

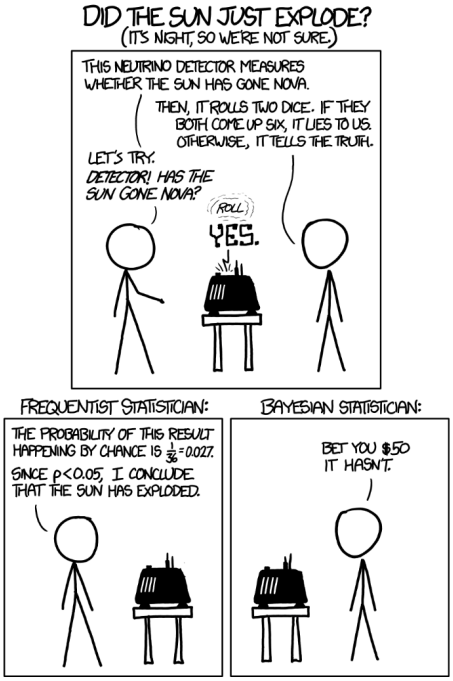
Wrap up of Markov Chain Monte Carlo, Variational Inference

Reading

- Sections 1-2 of "Variational Inference: A Review for Statisticians" by Blei et al. (2017).
- Sections 1-5 of "Auto-Encoding Variational Bayes" by Kingma and Welling (2013)
- Bishop Chapter 11.5-11.6
- Bishop Chapter 10-10.3

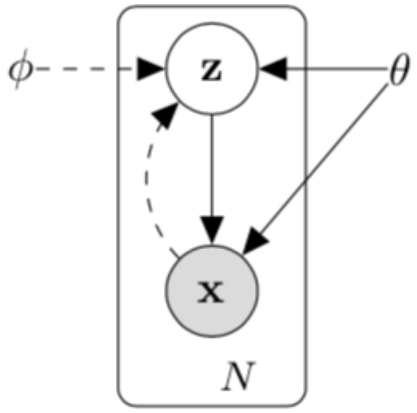
21.1 Review of MCMC, Hamiltonian Monte Carlo

Remark 165 (Why do we care about sampling more generally)**.** The standard notion of probability is defined using an event repeated may times over.... Sometimes we may want to do inference on events that cannot be repeated many times, e.g., if we may know from last year how fast the polar ice cap is melting. Suppose we have new evidence as data from a satellite, how should we update our belief?



Maximum likelihood vs. maximum a posteriori inference

21.2 Basics of Variational Inference



Will talk about about the "latent" variable z which is a function of the data x. E.g., if you

want to learn a network to draw an image of a dog, the variable z represents the dog, the color of its fur, the background (grass, house, park etc.). Given these latent factors, it is easy for you to imagine the picture of a dog in your head. What you are really doing in this process is that you are sampling from the distribution

$$\text{likelihood} = p(x\,|z).$$

This is called the likelihood, i.e., the likelihood of a datum x being generated by these latent factors z. For all the pictures of dogs you have have seen there is some distribution

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

which tells you often a particular latent factor shows up in the image of a dog. **We of course do not know the true distributions $p(x|z)$ and $p(z)$ in both these cases**. But suppose we have a dataset of $\{x_1, \ldots, x_n\}$ and their correponding latent variables $\{z_1, \ldots, z_n\}$, we would like to estimate the posterior distribution

$$p(z|x).$$

for a new datum x.

Remark 166 (Variational Calculus)**.** A function is something takes in a variable as input and returns the value of the function as the output, e.g., $\mathbb{R} \ni f(x) = 5x^2$ for $x \in \mathbb{R}$. A *functional* is something that takes in a function as an input and returns the output of the functional. An example of this is the entropy functional

$$\mathbb{R} \ni \mathrm{H}(p) = \int p(x)\ \log p(x)\ \mathrm{d}x$$

The function $p(x)$ is a probability density on the space $\mathbb{R}^N$. Entropy is therefore a *functional*: it takes in a distribution p as input and, in this case, returns a real number as the output. Similar to the standard calculus we know for functions, there is a corresponding calculus for functionals, e.g., the functional derivative $\frac{\delta H(p)}{\delta p}$ is defined in a funny way as

$$\int \frac{\delta H(p)}{\delta p}\phi(x)\ \mathrm{d}x = \lim_{e\to 0} \frac{\mathrm{H}(p+\epsilon\phi) - \mathrm{H}(p)}{\epsilon}$$

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.

The KL-divergence $\mathrm{KL}(p\ ||\ q)$ is another such functional, it has two arguments p and q.

Variational optimization is different from standard optimization only in the domain of the variable of optimization, e.g.,

$$x^* = \underset{x \in \mathbb{R}^N}{\text{argmin}} \; f(x)$$

is an example of the latter while

$$q^* = \underset{q \in \mathcal{P}(\mathcal{X})}{\text{argmin}} \; \text{KL}(q \,||\, p)$$

is an example of the former. In the second case, the variable of optimization is q and the domain is

$$\mathcal{P}(\mathcal{X})$$

which is the set of probability distributions with support on the set $\mathcal{X}$.

Remark 167 (The case of KL$(q||p)$ versus KL$(p||q)$)**.**

21.3 Auto-encoders

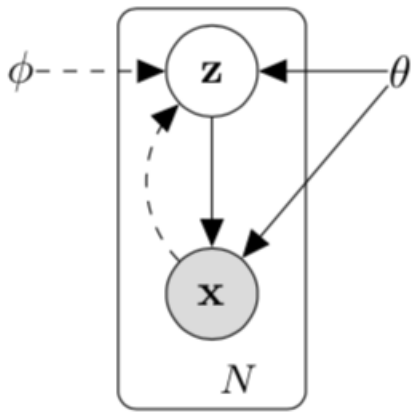
Lecture 22

Variational Inference, Auto-Encoders

Reading

- Sections 1-2 of "Variational Inference: A Review for Statisticians" by Blei et al. (2017).
- Sections 1-5 of "Auto-Encoding Variational Bayes" by Kingma and Welling (2013)
- Chapter 2 of Durk Kingma's thesis: https://pure.uva.nl/ws/files/17891313/Thesis.pdf.
- Bishop Chapter 11.5-11.6
- Bishop Chapter 10-10.3
- Lots of great intuition at http://ruishu.io/2018/03/14/vae/

22.1 Basics of Variational Inference



Will talk about about the "latent" variable z which is a function of the data x. E.g., if you want to learn a network to draw an image of a dog, the variable z represents the dog, the color of its fur, the background (grass, house, park etc.). Given these latent factors, it is easy for you to imagine the picture of a dog in your head. What you are really doing in this process is that you are sampling from the distribution

$$\text{likelihood} = p(x\,|z).$$

This is called the likelihood, i.e., the likelihood of a datum x being generated by these latent factors z. For all the pictures of dogs you have have seen there is some distribution

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

which tells you how often a particular latent factor shows up in the image of a dog. **We do not know the true (Nature's) distributions** $p(x|z)$ **and** $p(z)$. But suppose we have a dataset of $\{x_1, \ldots, x_n\}$ and their correponding latent variables $\{z_1, \ldots, z_n\}$, we would like to estimate the posterior distribution

$$p(z|x).$$

for a new datum x.

Remark 168 (Variational Calculus)**.** A function is something takes in a variable as input and returns the value of the function as the output, e.g., $\mathbb{R} \ni f(x) = 5x^2$ for $x \in \mathbb{R}$. A *functional* is something that takes in a function as an input and returns the output of the

functional. An example of this is the entropy functional

$$\mathbb{R} \ni \mathrm{H}(p) = \int p(x) \log p(x) \, \mathrm{d}x$$

The function $p(x)$ is a probability density on the space $\mathbb{R}^N$. Entropy is therefore a *functional*: it takes in a distribution p as input and, in this case, returns a real number as the output. Similar to the standard calculus we know for functions, there is a corresponding calculus for functionals, e.g., the functional derivative $\frac{\delta H(p)}{\delta p}(x)$ is defined in a funny way as

$$\int \frac{\delta H(p)}{\delta p}(x) \, \varphi(x) \, \mathrm{d}x = \lim_{\epsilon \to 0} \frac{\mathrm{H}(p + \epsilon\varphi) - \mathrm{H}(p)}{\epsilon}$$

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.

The KL-divergence $\mathrm{KL}(p \mid\mid q)$ is another such functional, it has two arguments p and q. Other than the fact that we are minimizing a functional, variational optimization is different from standard optimization only in the domain of the variable of optimization, e.g.,

$$x^* = \operatorname*{argmin}_{x \in \mathbb{R}^N} \; f(x)$$

is an example of the latter while

$$q^* = \operatorname*{argmin}_{q \, \in \, \mathcal{Q} \, \subset \mathcal{P}(\mathcal{X})} \mathrm{KL}(q \mid\mid p) \tag{22.1}$$

is an example of the former. In the second case, the variable of optimization is q and the domain is

$$\mathcal{Q} \subset \mathcal{P}(\mathcal{X})$$

which is a subset of the set of probability distributions with support on the set $\mathcal{X}$.

Remark 169 (Domain 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 intended 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.

(draw an example)

All the research on variational inference boils down to picking a good family of distributions

$\mathcal{Q}$ that makes solving (22.1) easy.

Remark 170 (Picking the objective)**.** 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.

Remark 171 (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 $\varphi(x)$

$$\mathop{\mathbb{E}}_{x\sim e^{-nf(x)}}[\varphi(x)] = \int e^{-nf(x)}\,\varphi(x)\,\mathrm{d}x.$$

The above integral takes many values, some have small $f(x)$ and some have large $f(x)$. The values of x where $f(x)$ is small are the ones that have the highest $e^{-nf(x)}$, especially as $n\to\infty$, and therefore the ones that we should pay most attention while approximating the integral. For large n, Laplace aprpoximation replaces the above integral by simply

$$\begin{aligned}\int e^{-nf(x)}\,\varphi(x)\,\mathrm{d}x &\approx \int \varphi(x)\,e^{-n\left(f(x^*)+\frac{1}{2}(x-x^*)^\top\nabla^2 f(x^*)(x-x^*)\right)}\,\mathrm{d}x\\ &= e^{-nf(x^*)}\int \varphi(x)\,e^{-\frac{n}{2}(x-x^*)^\top\nabla^2 f(x^*)(x-x^*)}\,\mathrm{d}x\end{aligned}$$

where $x^* = \operatorname{argmin} f(x)$ is the global minimum of $f(x)$. The integral is now with respect to a Gaussian distribution and can be done more easily, e.g., by Monte Carlo, or by Markov Chain Monte Carlo algorithms.

How does a variational approximation differ from the Laplace approximation? Let us look at an example.

Remark 172 (The case of $\mathrm{KL}(q||p)$ and $\mathrm{KL}(p||q)$)**.** (an example of a Gaussian mixture model)

Remark 173 (How to pick a variational family $\mathcal{Q}$)**.**

22.2 The Evidence Lower Bound (ELBO)

We will now go back to the generative model described before. What is the distribution we should try to approximate? We would like to estimate the posterior on the latent factors, $p(z|x)$ after observing some data $\{x_1, \ldots, x_n\}$ and their corresponding factors $\{z_1, \ldots, z_n\}$ which we we write together as $D = \{(x_i, z_i)\}_{i=1,\ldots,n}$. You can think of this in two ways: (i) if z are the classes, you are predicting the posterior on classes $p(z|x)$ after looking at the labeled dataset (this is the same setting as before); (ii) if z are the latent factors of your generative model, e.g., the color of the fur of a cat, you are predicting a posterior so that you may compress/understand many images of cats into their salient factors.

Therefore consider a variational approximation $q(z)$ to the distribution $p(z|x)$ we would like to estimate

$$\begin{aligned}
\mathrm{KL}(q(z) \,||\, p(z|x)) &= \mathop{\mathbb{E}}_{z\sim q(z)}\left[\log \frac{q(z)}{p(z|x)}\right] \\
&= -\mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z|x)\big] + \mathop{\mathbb{E}}_{z\sim q(z)}\big[q(z)\big] \\
&= -\mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z,x) - \log p(x)\big] + \mathop{\mathbb{E}}_{z\sim q(z)}\big[q(z)\big] \\
&= \log p(x) - \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z,x)\big] + \mathop{\mathbb{E}}_{z\sim q(z)}\big[q(z)\big].
\end{aligned}$$

We know that $\mathrm{KL}(q(z) \,||\, p(z|x)) \geq 0$. We therefore have

$$\begin{aligned}
\log p(x) &\geq \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z,x)\big] - \mathop{\mathbb{E}}_{z\sim q(z)}\big[q(z)\big] \\
&= \mathrm{ELBO}(q).
\end{aligned} \tag{22.2}$$

The left-hand side of this inequality is called the evidence, it simply the log-probability of the datum x. The right-hand side is a lower bound on the evidence and therefore called the "Evidence Lower Bound (ELBO)". Notice that the distribution $p(z, x)$ is given by Nature, the only way we know about it is through the samples in our dataset $D = \{(x_i, z_i)\}_{i=1,\ldots,n}$. The second term on the right-hand side is simply the entropy $H(q)$. The first term is the expectation of the joint distribution $p(z, x)$ over our variational approximation q.

Minimizing $\mathrm{KL}(q(z) \,||\, p(z|x)$ is equivalent to

$$q^* = \mathop{\mathrm{argmax}}_{q\in\mathcal{Q}} \mathrm{ELBO}(q) := \left\{ \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z,x)\big] + \mathrm{H}(q) \right\}. \tag{22.3}$$

because $p(x)$ does not depend on q. We have made the dependence of $\mathrm{ELBO}(q)$ on the variable of optimization q explicit.

Remark 174. To understand the problem in (22.3) further let us rewrite it in another way

$$\begin{aligned}\text{ELBO}(q) &= \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(z,x)\big] - \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log q(z)\big] \\ &= \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(x|z) + \log p(z)\big] - \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log q(z)\big] \\ &= \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(x|z)\big] - \text{KL}(q(z)\,||\,p(z)).\end{aligned}$$

This form

$$\text{ELBO}(q) = \mathop{\mathbb{E}}_{z\sim q(z)}\big[\log p(x|z)\big] - \text{KL}(q(z)\,||\,p(z)) \tag{22.4}$$

is very interesting. The first term is the likelihood of data given the latent factor z. The second term is the distance between our variational approximation $q(z)$ and the true marginal distribution $p(z)$. The distribution $p(z)$ is Nature's prior over the latent factors, so the second term simply measures how far $q(z)$ is from Nature's prior. The above expression is conceptually very similar to what we have doing in standard optimization, e.g.,

$$\mathop{\text{argmin}}_{w}\left\{f(w) + \frac{\lambda}{2}\|w\|_2^2\right\}.$$

The first term, say the cross-entropy loss, is akin to the likelihood of data $p(x|z)$. The second term is a regularizer which enforces our prior knowledge that the parameters w should be close to the original; this is akin to Nature picking $p(z)$ to be a Dirac-delta distribution around the origin. ELBO balances between the prior over the latent factors and the likelihood over the data.

22.3 Auto-encoders

Lecture 23

Auto-Encoders, Information Bottleneck

Reading

- Sections 1-3 of "Deep Variational Information Bottleneck " by Alemi et al. (2016)
- "Weight Uncertainty in Neural Networks" by Blundell et al. (2015)
- "Variational Dropout and the Local Reparameterization Trick" by Kingma et al. (2015)
- "The information bottleneck method" by Tishby et al. (2000)
- Some extra reading: Sections 1-3 of "Sticking the Landing: Simple, Lower-Variance Gradient Estimators for Variational Inference" by Roeder et al. 2017 https://papers.nips.cc/paper/7268-sticking-the-landing-simple-lower-variance-gradient-estimators-for-variational-inference.pdf

---

23.1 Gradient of ELBO

23.1.1 Likelihood ratio trick

Remark 175 (Policy gradient in Reinforcement Learning)**.**

23.1.2 Reparametrization trick

23.2 Some tricks and caveats about ELBO

Remark 176 (Bayesian neural networks and stochastic gradient descent)**.**

Remark 177.

23.3 Information Bottleneck

Remark 178. β-variational autoencoders (β-VAE) and disentangled representations.

Lecture 24

Weight uncertainty in neural networks

Reading

- "Weight Uncertainty in Neural Networks" by Blundell et al. (2015)
- Sections 1-6 of notes on generalization bounds by David McAllester at https://ttic.uchicago.edu/~dmcallester/ttic101-07/lectures/generalization/generalization.pdf

---

24.1 Titbits related to variational inference

24.1.1 Beta-variational autoencoders and disentangled representations

Recall that ELBO is given by

$$\text{ELBO}(q) = \mathop{\mathbb{E}}_{z\sim q(z|x)} \left[\log p(x|z)\right] - \text{KL}\left(q(z|x) \,||\, p(z)\right).$$

Instead of deriving it by minimization of $\text{KL}(q(z|x) \,||\, p(z\,|x))$, we can think of solving an optimization problem

$$\begin{aligned} &\text{maximize} \mathop{\mathbb{E}}_{z\sim q(z|x)} \left[\log p(x|z)\right] \\ &\text{such that } \text{KL}\left(q(z|x) \,||\, p(z)\right) \leq \epsilon. \end{aligned} \tag{24.1}$$

This problem seeks to find a $q(z|x)$ such that it maximizes the likelihood $p(x \mid z)$ while remaining close to the prior $p(z)$. Let us introduce a Lagrange multiplier β^{-1} for the

constraint to get

$$\mathcal{L}(q, \beta^{-1}) = \mathop{\mathbb{E}}_{z\sim q(z|x)} \left[\log p(x|z)\right] - \beta^{-1} \left(\mathrm{KL}\left(q(z|x) \,||\, p(z)\right) - \epsilon\right).$$

We have written β^{-1} instead of the standard β for a special reason which will become clear soon. It is typically difficult to optimize both the weights of $q(z|x)$ and β^{-1} in deep learning because one would need to take two backprop updates. It is therefore fashionable to treat β^{-1} as a fixed hyper-parameter and think of imposing the constraint "softly". The corresponding objective

$$\mathcal{L}_\beta(q) := \mathop{\mathbb{E}}_{z\sim q(z|x)} \left[\log p(x|z)\right] - \beta^{-1}\mathrm{KL}\left(q(z|x) \,||\, p(z)\right). \tag{24.2}$$

is called the β-VAE. It affords more control over imposing the prior through the parameter β. Larger the β smaller the KL penalty and more the focus on simply maximizing the likelihood of data. Smaller the β, larger the KL penalty and the lesser the focus on maximizing the likelihood at the cost of moving away from the prior $p(z)$. Note that $\mathcal{L}_\beta(q)$ is no longer a lower bound on $\log p(x)$.

Remark 179 (Disentanglement of the factors)**.** We now know how to create a variational approximation $q(z|x)$ to the true posterior $p(z|x)$. If we are going to use these factors $z \sim q(z|x)$ for doing some downstream deduction, e.g., did the neural network classify a car by detecting the wheels, or did it classify an orange as an orange using both its shape and color? Answering such questions is hard but critical in practice. For example, if we use a neural network to predict a strain of cancer using some features we would like to know which feature was the most important predictor for a particular test datum. The central concept in such questions is known as disentanglement. While an exact definition is unclear, one may imagine disentanglement as measuring how independent the latent factors are from each other, e.g.,

$$\mathrm{KL}\left(q(z|x) \,||\, \prod_{i=1}^{d} q(z_i \mid x)\right) := \mathrm{I}(z \mid x)$$

which is also called the multi-variate mutual information of the random variable $z|x$.

Given a prior $p(z)$, we can think of the hyper-parameter β^{-1} as controlling the complexity of the distribution $q(z \mid x)$ relative to $p(z)$. Intuitively, more independent the latent factors z, more simple the distribution is. This has given rise to a line of literature (Bengio et al., 2013; Mathieu et al., 2018) that uses the hyper-parameter $\beta^{-1} \gg 1$ to learn strong factors, potentially at the cost of the reconstruction error.

24.1.2 Information Bottleneck

We will skip this material. Read work (Achille and Soatto, 2018; Shwartz-Ziv and Tishby, 2017; Tishby et al., 2000) if you are interested in learning about it.

24.2 Bayesian Neural Networks

Maximum Likelihood Estimator (MLE) computes the most likely set of weights that fit the dataset $D = \{(x_i, y_i)\}_{i=1,\dots,n}$:

$$\begin{aligned} w_{\text{MLE}} &= \underset{w \in \mathbb{R}^N}{\text{argmax}} \ \log p(D \mid w) \\ &= \underset{w \in \mathbb{R}^N}{\text{argmax}} \ \sum_{i=1}^{n} \log p(y_i \mid x_i;\ w). \end{aligned} \tag{24.3}$$

The loss $\log p(y \mid x;\ x)$ above can be the cross-entropy and we can optimize the objective using SGD. Regularization involves putting a prior on the weights w and finding the maximum a posteriori (MAP) estimator

$$\begin{aligned} w_{\text{MAP}} &= \underset{w \in \mathbb{R}^N}{\text{argmax}} \ \log p(D \mid w) \\ &= \underset{w \in \mathbb{R}^N}{\text{argmax}} \ \sum_{i=1}^{n} \log p(y_i \mid x_i;\ w) + \log p(w). \end{aligned} \tag{24.4}$$

For example, if $p(w)$ were Gaussian, this would amount to ℓ_2 regularization. If $p(w)$ is a Laplace distribution $\propto e^{-|w|}$, this amounts to ℓ_1 regularization.

Until now, we have always been using variational inference to approximate the distributions on the latent factors of the data. We can also use variational inference to approximate the posterior on the weights themselves. The process is as follows.

1. You pick a prior $p(w)$ based on your domain knowledge, e.g., if you are doing magnetic resonance imaging (MRI) where the true classifier is expected to have sparse weights (lots of zero weights), you would pick $p(w)$ to be a Laplace distribution.

2. Training involves the following: For each weight $w \sim p(w)$ you check the likelihood of data under the drawn weight $p(D \mid w)$ and compute the posterior distribution

   $$p(w \mid D) \propto p(D \mid w)\ p(w)$$

   on the weights w.

3. At inference time, you draw weights from $w \sim p(w \mid D)$ and compute the likelihood of the label y given the datum x under all such draws

$$p(y \mid x;\ D) = \int p(y \mid x; w)\ p(w \mid D)\ \mathrm{d}w.$$

Notice that having access to the distribution $p(w \mid D)$ as opposed to just a point estimate w_{MLE} or w_{MAP} is powerful. Instead of simply computing

$$p(y \mid x;\ w_{\text{MLE}})$$

we can also estimate which weights are critical to the predictions and which are not, e.g., if

$$p(w_i \mid D)$$

has a large variance, the dataset D is not sufficient to learn w_i accurately. If the model has a small training error, this implies that the weight w_i is not important to predict the data. This insight can be used in many different ways: (i) to compress the model for faster inference if computational resources are limited (mobile phones), (ii) to estimate which features are important to predict the lables in medical diagnosis etc. We can also compute the variance of $y \sim p(y \mid x;\ D)$ itself to get an estimate of the uncertainty of the predictions made by the network.

We know a way to approximate distributions: we are simply going to learn a variational approximation $q_\theta(w \mid D)$ to the true posterior $p(w \mid D)$

$$\begin{aligned}
\theta^* &= \underset{\theta}{\text{argmin}}\ \mathrm{KL}\left(q_\theta(w \mid D) \,||\, p(w \mid D)\right) \\
&= \underset{\theta}{\text{argmin}} \int q_\theta(w \mid D)\ \log \frac{q_\theta(w \mid D)}{p(w \mid D)}\ \mathrm{d}w \\
&= \underset{\theta}{\text{argmin}}\, \mathrm{KL}\big(q_\theta(w \mid D) \,||\, p(w)\big) - \underset{w \sim q_\theta(w \mid D)}{\mathbb{E}} \left[\log p(D \mid w)\right].
\end{aligned} \tag{24.5}$$

Note that the parameters θ predict the weights w themselves

$$w \sim q_\theta(w \mid D).$$

The objective in the above problem is again a combination of two terms, the first term is the distance from the prior $p(w)$ and the second term measures how well the distribution $q_\theta(w|D)$ performs *on an average* on the dataset D.

Some remarks are in order:

- While it is not unreasonable to assume that Nature's prior $p(z) = N(0, I)$ in a variational auto-encoder, positing a prior $p(w)$ in (24.5) requires more care. It is unlikely that the prior weights of the model are independent from each other and identically distributed with zero mean. Bayesian Neural Networks therefore consider things like "scale and slab priors" which is a (carefully chosen) mixture of Gaussians

$$p(w) = \prod_{i=1}^{N} p(w_i); \quad \text{where } p(w_i) = \pi N(w_i; 0, \sigma_1^2) + (1 - \pi) N(w_i; 0, \sigma_2^2).$$

- The variational family now is in charge of predicting weights of a neural network. It is therefore difficult to parameterize $q_\theta(w)$ using a neural network itself because although a neural network predicting another neural network that predicts the labels is possible the result will be difficult to train. So we will pick a simpler family: the weights θ are split into two parts $\theta = (\mu, \sigma)$ where

$$w = \mu + \log(1 + \exp(\sigma)) \circ \epsilon$$

  with $\epsilon \sim N(0, I)$. Think carefully about what has happened here, choosing the variational family is a complete design decision, Bayesian neural networks demonstrate how one exploits this concept.

- Because we assumed an intricate prior $p(w)$, we cannot compute the KL-term

$$\mathrm{KL}\left(q_\theta(w) \,||\, p(w)\right)$$

  in closed form anymore like we did for variational auto-encoders. But this does not really hurt. We can minimize the objective in (24.5) using the reparametrization trick

$$\nabla_\theta \mathop{\mathbb{E}}_{q_\theta(w)} [f(w, \theta)] = \mathop{\mathbb{E}}_{q(\epsilon)} \left[\frac{\partial f(w, \theta)}{\partial w} \frac{\partial w}{\partial \theta} + \frac{\partial f(w, \theta)}{\partial \theta} \right].$$

  where $q(\epsilon)$ is noise that we introduce in

$$w = g(\theta) + \epsilon$$

  and

$$f(w, \theta) = \log q_\theta(w) - \log p(w) - \log p(D \mid w).$$

- After training the network, we make predictions on the new datum by estimating the

integral using a few samples

$$p(y \mid x;\ D) = \int p(y \mid x;\ w)\ p(w \mid D)\ \mathrm{d}w$$
$$\approx \frac{1}{m} \sum_{i=1}^{m} p(y \mid x;\ w^i)$$

where $w^i \sim q_\theta(w^i \mid D)$.

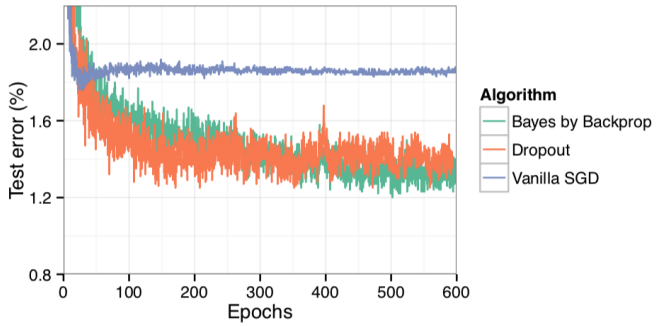


Figure 2. Test error on MNIST as training progresses.

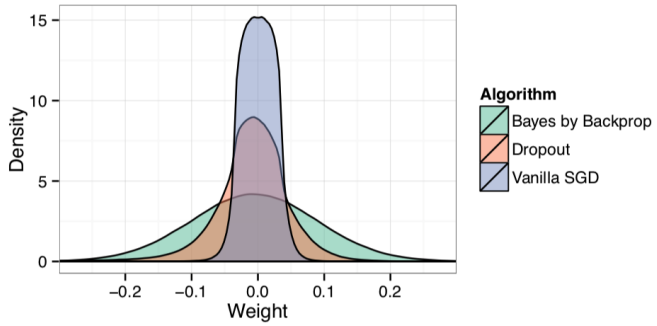


Figure 3. Histogram of the trained weights of the neural network, for Dropout, plain SGD, and samples from Bayes by Backprop.

Remark 180 (Dropout as approximating a Bayesian neural network)**.** Sometimes, instead of sampling the weights from $q_\theta(w)$ we use Dropout at *test time* and average the predictions of the network. This is an approximation of a Bayesian neural network (Kingma et al., 2015).

Remark 181 (SGD already performs Bayesian inference)**.** An almost miraculous interpretation of SGD is that it trains a Bayesian neural network implicitly. Recall that the Gibbs

distribution of the weights of a neural network w as given by

$$\rho^{\infty}(w) = \frac{e^{-\beta f(w)}}{Z(\beta)}.$$

We saw how the distribution of the states in a finite-state Markov chain converges to the steady-state distribution because the quantity

$$\mathrm{KL}(\pi^t \,||\, \pi^*)$$

decreases monotonically. Similar to this, if $\rho^t(w)$ is the distribution of the weights of lots of neural networks trained by SGD at time t on the same dataset, the quantity

$$\mathrm{KL}\left(\rho^t \,||\, \rho^{\infty}\right) = \underset{w \sim \rho^t \in \mathcal{P}}{\mathbb{E}} \left[f(w)\right] - \beta^{-1}\, \mathrm{H}(\rho^t) \tag{24.6}$$

decreases monotonically. The second term is the entropy of the distribution ρ^t. This looks very similar to the objective of a Bayesian neural network

$$\underset{w \sim q_\theta(w \mid D)}{\mathbb{E}} \left[-\log p(D \mid w)\right] + \mathrm{KL}\big(q_\theta(w \mid D) \,||\, p(w)\big).$$

Indeed, if we put $f(w) := -\log p(D \mid w)$ and assume a uniform prior $p(w)$ we get back (24.6) exactly for $\beta^{-1} = 1$. In other words, the stochasticity of mini-batches in SGD automatically creates a regularization effect via the term $\mathrm{KL}\big(q_\theta(w \mid D) \,||\, p(w)\big)$. This regularization is with respect to a uniform prior $p(w)$ on the weights. We will not do this but it turns out that uniform priors are very effective for the neural networks because of the special shape of the energy landscape (lots of connected regions and symmetries in the weight space). This is yet another insight into why SGD is so effective at training deep neural networks.

Lecture 25

Weight uncertainty in neural networks, PAC-Bayes generalization bound

Reading

- Sections 1-6 of notes on generalization bounds by David McAllester at https://ttic.uchicago.edu/~dmcallester/ttic101-07/lectures/generalization/generalization.pdf

---

25.1 Generalization

For a classifier $h \in \mathcal{H}$ where $\mathcal{H}$ is our hypothesis class and $D = \{(x_i, y_i)\}_{i=1,\ldots,n}$ is our dataset, the empirical risk (just our training loss) is

$$\hat{R}(h) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{h(x_i) \neq y_i}.$$

The population risk of h is defined for data drawn from Nature's distribution

$$\begin{aligned} R(h) &= \mathop{\mathbb{E}}_{(x,y)\sim\mathbb{P}} \mathbf{1}_{h(x_i)\neq y_i} \\ &= \mathbb{P}(h(x) \neq y). \end{aligned} \tag{25.1}$$

A generalization bound is of the form

$$|\hat{R}(h) - R(h)| \leq \text{something}.$$

In other words, the empirical risk using n samples is within some value of the population risk. The right hand side of the above inequality should typically go to zero if our learning algorithm is any good. How fast it goes to zero with respect to the number of data and the number of parameters in $\mathcal{H}$ is what we are interested in finding out. This will give us an idea of how much data we should use to fit a model.

25.2 Vapnik-Chervonenkis bounds

We will first review generalization which you might have seen in regards to SVMs. Consider $X_1, \ldots, X_n$ iid zero-one random variables where

$$\mathbb{P}(X_i = 1) = p \quad \mathbb{P}(X_i = 0) = 1 - p.$$

A good estimate of p could be the average of the X_i

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

From the central limit theorem we know that $\hat{p}$ has a normal distribution

$$\sqrt{n}\,(\hat{p} - p) \sim \text{Normal}(0,\ \sigma^2 = p(1-p)).$$

We therefore essentially have

$$\mathbb{P}(\sqrt{n}|\hat{p} - p| > s) \approx 2e^{-\frac{s^2}{2\sigma^2}}.$$

The Chernoff bound is a more precise estimate of a similar nature.

$$\mathbb{P}(|\hat{p} - p| > t) \leq 2e^{-2nt^2}$$

In a more formal form, it looks like

$$\mathbb{P}\left(\left|\mathbb{E}\left[X\right] - \frac{1}{n}\sum_{i=1}^{n} X_i\right| > t\right) \leq 2e^{-2nt^2}. \tag{25.2}$$

Given events A_i, the union bound allows us to compute an upper bound on the probability

$$\mathbb{P}\left(A_1 \cup A_2 \ldots \cup A_n\right) \leq \sum_{i=1}^{n} \mathbb{P}(A_i). \tag{25.3}$$

Note that the empirical risk $\hat{R}(h)$ has the population risk $R(h)$ as the mean. So we can directly write

$$\mathbb{P}(|\hat{R}(h) - R(h)| > t) \leq 2e^{-2nt^2}.$$

Let us imagine that we have a finite number of hypotheses in our hypothesis class

$$\mathcal{H} = \{h_1, h_2, \ldots, h_k\} \quad |\mathcal{H}| = k.$$

For each event A_i which is $|\hat{R}(h_i) - R(h_i)| > t$ we have a corresponding Chernoff bound. We can now combine everything using the union bound to get

$$\begin{aligned}\mathbb{P}(\exists h \in \mathcal{H} \text{ such that } |\hat{R}(h) - R(h)| > t) = \mathbb{P}\left(A_1 \cup \ldots A_k\right) \\ &\leq \sum_{i=1}^{n} \mathbb{P}(A_k) \\ &\leq 2ke^{-2nt^2}.\end{aligned}$$

This is called a uniform convergence bound: we have converted our individual bounds on each of the hypotheses h into a statement about the entire hypothesis class $\mathcal{H}$. We use this in the following way. Suppose we wanted to guarantee that no matter which finite dataset is drawn, with probability at least $1 - \delta$ we should have

$$\mathbb{P}\left(\forall h \in \mathcal{H} \quad |\hat{R}(h) - R(h)| \leq t\right) \geq 1 - \delta$$

we will need

$$n \geq \frac{1}{2t^2} \log \frac{2k}{\delta}$$

amount of data. This function is called the "sample complexity" of the hypothesis class. It is a function of the number of classifiers in $\mathcal{H}$ and our desired probability $1 - \delta$ and fudge room t. You will sometimes see this written as

$$\mathbb{P}\left(\forall h \in \mathcal{H} \quad |\hat{R}(h) - R(h)| \geq \sqrt{\frac{1}{2n} \log \frac{2k}{\delta}}\right) \leq \delta. \tag{25.4}$$

or

$$\hat{R}(h) \leq R(h) + 2\sqrt{\frac{1}{2n} \log \frac{2k}{\delta}}$$

for any $h \in \mathcal{H}$ with probability $1 - \delta$. Now comes a critical step: define the solution of our standard training problem as

$$h_{\text{ERM}} = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{R}(h).$$

The best hypothesis in our hypothesis class for the population risk is

$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} R(h).$$

Since we have for all $h \in \mathcal{H}$

$$|\hat{R}(h) - R(h)| \leq t$$

we can say

$$R(h_{\text{ERM}}) \leq \hat{R}(h_{\text{ERM}}) + t \leq \hat{R}(h^*) + t \leq R(h^*) + 2t. \tag{25.5}$$

This gives the following theorem

$$R(h_{\text{ERM}}) \leq R(h^*) + 2\sqrt{\frac{1}{2n} \log \frac{2|\mathcal{H}|}{\delta}} \tag{25.6}$$

with probability at least $1 - \delta$.

Remark 182 (Infinite hypothesis class)**.** What if $|\mathcal{H}| = \infty$? Even for an SVM with real-valued weights, we have an infinite number of candidate models. To reason about this, we will resort to quantization: say we have d parameters in the SVM and each one of them is stored in finite precision using b bits. We then have

$$|\mathcal{H}| = 2^{bd}$$

which gives

$$R(h_{\text{ERM}}) \leq R(h^*) + 2\sqrt{\frac{1}{2n}\left(bd \log 2 + \log \frac{2}{\delta}\right)} \tag{25.7}$$

In other words, the number of data n has to scale linearly with the number of parameters d if we are to guarantee good generalization performance of an SVM. Note that the right hand-side necessarily depends on the numerical precision used to store the weights, this is a very uncomfortable pattern that we see in all theories for generalization in machine learning.

Remark 183. The Vapnik-Chervonenkis dimension $\text{VC}(\mathcal{H})$ is the size of the largest set that can be shattered by $\mathcal{H}$. See https://en.wikipedia.org/wiki/Vapnik-Chervonenkis_dimension for more information.

(give examples, one for a linear classifier $\text{VC}(\mathcal{H}) = (d + 1)$, for a finite hypothesis class (decision table, $\text{VC}(\mathcal{H}) = \log k$), and a rectangle-based classifier ($\text{VC}(\mathcal{H}) = 4$).

The VC-dimension is a measure of the complexity of the hypothesis class and our generalization bound in (25.6) becomes, if $\text{VC}(\mathcal{H}) = d$

$$R(h_{\text{ERM}}) \leq R(h^*) + 2C\sqrt{\frac{1}{2n}\left(d\log\frac{n}{d} + \log\frac{1}{\delta}\right)} \tag{25.8}$$

where C is some constant which is typically greater than 1. Again the number of data n needs to scale linearly with the VC-dimension of the hypothesis class.

Remark 184 (VC-dimension for neural networks)**.** The generalization bound in (25.8) is very general and applies for neural networks as well. The trouble however starts with computing the VC-dimension of a neural network. This is a very active research area right now and the best estimate of the VC-dimension we have for a neural network is

$$\text{VC}(\mathcal{H}) \approx LN\log N$$

where L is the number of layers and N is the total number of parameters (Bartlett et al., 2019). Let us evaluate this number for a typical application. The number of weights in the neural network you trained on CIFAR-10 is about $N = 10^6$. The number of layers is, say 5. The number of data is $n = 50,000$. The sample complexity in (25.8) is therefore (ignoring the log terms)

$$\approx 2\sqrt{\frac{5\times 10^6}{10^4}} \approx 50.$$

The VC-bound is therefore saying that the gap between the validation *loss* and the training *loss* is less than 50. This is a vacuous statement because we know that the gap between both of them is small: the population risk is simply the probability that the classifier makes an incorrect prediction and the empirical risk is an estimate of this probability, so the gap between them better be less than 1.

This is the reason why generalization performance of deep networks is mysterious. Deep networks have lots of parameters and, by most estimates, a huge VC-dimension. They are trained on relatively little amounts of data and they should overfit heavily. In practice, deep networks also do overfit very heavily. You can essentially take any dataset (even one with completely random labels) and if you use a large enough neural network, you will be able to achieve $\hat{R}(h_{\text{ERM}}) \approx 0$. For most real datasets, despite this overfitting, the generalization gap

$$\text{gap} = |R(h_{\text{ERM}}) - \hat{R}(h_{\text{ERM}})|$$

of deep networks is small, small enough to be deployed in practice for real applications.

25.3 PAC-Bayes bounds

We will now take a look at another theory of generalization. This is quite different from VC-theory and lends itself to slightly tighter generalization bounds even for deep networks. This style of thinking is intimately connected to Bayesian neural networks/ELBO.

Recall that the Bayesian view of model fitting has the benefit of allowing us to enforce a domain-specific prior on the learning algorithm. This prior is simply our understanding of the data, it may or may not be "true". If it is not true, we are unlikely to get good predictions, e.g., enforcing a sparsity prior while doing linear regression where the true weights are not sparse will not have good residuals. Succinctly, the generalization performance of a Bayesian algorithm is dependent on the prior we impose.

The acronym PAC stands for Probably Approximately Correct (Valiant, 2013); see https://jeremykun.com/2014/01/02/probably-approximately-correct-a-formal-theory-of-learning for an intuitive introduction. A PAC inequality states that with an arbitrarily high probability (hence "probably"), the performance (as provided by a loss function) of a learning algorithm is upper-bounded by a term decaying to an optimal value as more data is collected (hence "approximately correct"). The strength of the PAC-approach is that we can obtain a generalization guarantee without assuming that we know the true prior.

Example 185 (Interval estimation by two players: Nature and the Learner)**.** Consider a game between two players: the first is Nature which outputs real numbers. If the number lies inside the interval $[a^*, b^*] \subset \mathbb{R}$ Nature labels it +1 and -1 otherwise. The task of the second player, the Learner, is to guess the interval using the numbers. Note that it does not really matter what prior or algorithm the Learner uses to guess the interval. You can intuitively guess that a good estimate of the interval is simply the minimum and maximum of the numbers labeled +1. Given any estimate $[\hat{a}, \hat{b}]$ we can measure how good the estimate is.

The "probably" part of PAC is upon random draws of the dataset. The "approximately correct" part of PAC theory pertains to the claim of the Learner that $\hat{a} \approx a^*$ and $\hat{b} \approx b^*$. This claim becomes more accurate as we collect more data.

The PAC-Bayesian approach essentially merges the two: the algorithm still uses a domain-specific prior to learn but its generalization guarantee is not dependent on the prior it uses. This is both a feature and a bug, we will focus on the former here.

The rest of the material in this section follows https://ttic.uchicago.edu/d̃mcallester/ttic101-07/lectures/generalization/generalization.pdf and https://ttic.uchicago.edu/d̃mcallester/DeepClass/bounds.pdf

Using essentially the same argument as that in the derivation of VC bound, we can show that

for all $h \in \mathcal{H}$ with probability at least $1 - \delta$

$$\hat{R}(h) \leq R(h) + \sqrt{\frac{|h| \log 2 + \log 1/\delta}{2n}} \tag{25.9}$$

The notation $|h|$ is the number of bits required to store the weights of the hypothesis $h \in \mathcal{H}$. Each hypothesis h has a bit-string representation of the form

$$h = 0001110001 \ldots \text{n}$$

with a null-character "n" terminating it. If we want to enumerate all the hypotheses in $\mathcal{H}$ we simply enumerate all these null-terminated bit-strings.

The above formula is the same as (25.7) except that we can now have a hypothesis class where different hypotheses are represented using different number of bits. This formula is called Occam's bound because the right-hand-side depends on how complex H is. If the function h is complicated and we need lots of bits to store it, we should also expect the generalization gap to be large for that function. This is therefore a model selection mechanism. Instead of simply minimizing the training loss we can add a complexity term as the regularizer and minimize

$$\operatorname*{argmin}_{h \in \mathcal{H}} \hat{R}(h) + \sqrt{\frac{|h| \log 2 + \log 1/\delta}{2n}}.$$

This loss function will pick a simple hypothesis if it fits the data, i.e., if it has a small $\hat{R}(h)$. The theorem guarantees that such simple hypotheses will also generalize well; this is the principle of Occam's Razor. You may have seen similar formulae before while studying Akaike/Bayesian Information Criteria (AIC/BIC) for probabilistic graphical models; see https://en.wikipedia.org/wiki/Bayesian_information_criterion for an introduction.

The expression in (25.9) also has a Bayesian interpretation. Let each rule $h \in \mathcal{H}$ be picked according to some prior probability $p(h)$. Using this prior distribution P we can come up with a more efficient way to represent h, if $p(h)$ is large, we use few bits to encode it and if $p(h)$ is small, we use more bits to encode it. This will make sure that on an average we are not using too many bits. Define the length of this encoding to be

$$|h|_p = -\log p(h).$$

We now get that with probability at least $1 - \delta$,

$$\forall h \in \mathcal{H} \quad \hat{R}(h) \leq R(h) + \sqrt{\frac{|h|_p \log 2 + \log 1/\delta}{2n}}.$$

This bound is Bayesian in the sense that it depends on an arbitrary prior p. But note that the theorem itself holds for any prior p.

The distribution p is about representations of the hypotheses h.

Introduce the Gibbs classifier

25.3.1 Non-vacuous PAC-Bayes bounds

Some of this material is taken from Dziugaite and Roy (2017) but you are not required to read this paper.

Experiment	T-600	T-1200	T-300^2	T-600^2	T-1200^2	T-600^3	R-600
Train error	0.001	0.002	0.000	0.000	0.000	0.000	0.007
Test error	0.018	0.018	0.015	0.016	0.015	0.013	0.508
SNN train error	0.028	0.027	0.027	0.028	0.029	0.027	0.112
SNN test error	0.034	0.035	0.034	0.033	0.035	0.032	0.503
PAC-Bayes bound	0.161	0.179	0.170	0.186	0.223	0.201	1.352
KL divergence	5144	5977	5791	6534	8558	7861	201131
# parameters	471k	943k	326k	832k	2384k	1193k	472k
VC dimension	26m	56m	26m	66m	187m	121m	26m

Table 1: Results for experiments on binary class variant of MNIST. SGD is either trained on (T) true labels or (R) random labels. The network architecture is expressed as N^L, indicating L hidden layers with N nodes each. Errors are classification error. The reported VC dimension is the best known upper bound (in millions) for ReLU networks. The SNN error rates are tight upper bounds (see text for details). The PAC-Bayes bounds upper bound the test error with probability 0.965.

Lecture 26

Generative Adversarial Networks (GANs)

Reading

- Andrew Ng's notes on generative models
  http://cs229.stanford.edu/notes/cs229-notes2.pdf
- The original GAN paper by Goodfellow et al. (2014)
- "The Numerics of GANs" by Mescheder et al. (2017)

26.1 Implicit generative models

We have seen the following two ways to sample from probability distributions.

1. If we only have samples $x_1, \ldots, x_n \sim p(x)$, say a dataset of natural images, and would like to sample more data from the same distribution, we can use an auto-encoder. This is one among the many different ways of creating samples given a bunch of data; other examples would be kernel density estimation, fitting a probabilistic graphical model to this data and sampling from it, etc.

2. It may happen that we do not have samples from the true distribution. Consider a problem where one wants to synthesize proteins with certain properties. Each protein is defined by the position and orientation of its constituent atoms denoted by $q_1, q_2, \ldots, q_N$. It is typically common to know how good a particular configuration in terms of it being useful

of becoming a particular drug, e.g., one can usually talk to a chemist and write down a function

$$H(q_1, \ldots, q_N);$$

if this function is small then the protein is "good", if this function is large then the protein is not useful. Synthesizing a protein would then involve solving

$$q^* = \underset{q_1,\ldots,q_N}{\text{argmin}}\ H(q_1, \ldots, q_N).$$

which will give one particular protein, or sampling many good proteins would involve drawing samples from

$$q \sim \frac{e^{-H(q_1,\ldots,q_N)}}{Z};$$

as we have seen many times now this distribution puts probability mass on configurations where H is small. The trouble is that we know the numerator but do not typically know the denominator Z: computing Z involves a very large sum over all configurations. Rejection sampling, importance sampling, Markov chain Monte Carlo (Langevin dynamics, Hamiltonian Monte Carlo and their variants) are methods that allow sampling from such distributions.

To summarize, both the above methods are explicit. In one case we have a variational approximation of the distribution, say the decoder, whereas in the other case we know the distribution upto the normalization constant. Generative Adversarial Networks (GANs) are an example of *implicit models*. GANs allow drawing samples from a probability distribution so they fall in the same class of generative models as the two methods above. GANs are implicit because they never maintain the probability distribution or an approximation for it. They cannot therefore compute the probability of a particular sample.

The Generator part of a GAN works in the same way as the decoder in a VAE. Given a sample z from some prior, most commonly a standard normal, we train a neural network to generate a sample

$$x = g_\theta(z).$$

The place where GANs differ from explicit models is how they train the generator. We will look at this in the following section.

26.2 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

$$D_1 = \{x_1, \ldots, x_m, : \ x_k \sim p(x)\}$$
$$D_2 = \{x_1, \ldots, x_m, : \ x_k \sim q(x)\}.$$

Given access to D_1 and D_2 we would like to check if the two distributions $p(x)$ and $q(x)$ are the same or not. Let us define the null hypothesis which claims that the two distributions are the same.

$$H_0 : p = q$$

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 typial two-sample test will construct a statistic $\hat{t}$ out of the two datasets, e.g., their individual means, their variances, and will use these statistics to accept or reject the null hypothesis. Let's say that we pick a threshold t, and the test statistic is the difference of the means

$$\hat{t} = \Big|\frac{1}{m}\sum_{x \in D_1} x - \frac{1}{m}\sum_{x \in D_2} x\Big|.$$

A statistician will then say that the null hypothesis is valid with *level* α if

$$\mathop{\mathbb{P}}_{D_1 \sim p,\ D_2 \sim p}\left(\hat{t} > t_\alpha\right) \leq \alpha.$$

In other words, if the null hypothesis were true 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 can be hope that the entire two distributions are the same despite only have a finite dataset to check them. 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}".

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

$$\mathop{\mathbb{P}}_{D_1 \sim p, D_2 \sim q}\left(\hat{t} > t_\alpha\right)$$

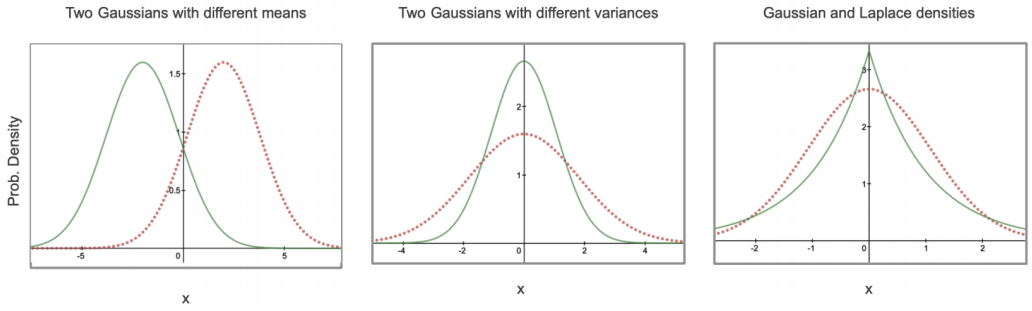
being large.

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 estimate the distance between distributions.

Remark 186. 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., $\alpha = 0.05$ is, roughly speaking, an accepted standard. We would like to make statements about distributions of natural images, what test statistic $\hat{t}$ should we pick? Finding a two-sample test statistic in large dimensions is hard.



The key idea behind GANs is to learn the statistic $\hat{t}$. Given two distributions that we know to be different, namely the one of our natural images and the images generated by the generator, a good statistic is the one that is very large when evaluated across them. We imagine that we learn a binary classifier $d_\varphi : \mathcal{X} \mapsto [0, 1]$. We next create a dataset to train this classifier on: samples $x \sim p(x)$ are labeled $y = 1$ and samples $x \sim q(x)$ are labeled $y = 0$.

$$D = \{(x_i, 0)_{i=1,\ldots,n} : \; x_i \sim p(x)\} \cup \{(x_i, 1)_{i=1,\ldots,n} : \; x_i \sim q(x)\} \, .$$

We will fit d_φ to estimate the probability

$$\mathbb{P}\left(y = 1 | x\right) = d_\varphi(x).$$

The loss function for training this classifier with the binary cross-entropy loss ($-(y \log p +$

$(1-y)\log(1-p)))$ will look like

$$\varphi^* = \underset{\varphi}{\text{argmax}} \underset{x\sim p(x)}{\mathbb{E}} \big[\log d_\varphi(x)\big] + \underset{x\sim q(x)}{\mathbb{E}} \big[\log(1-d_\varphi(x))\big]. \tag{26.1}$$

Remark 187. What is the solution of the above training for the discriminator? It is simply

$$d_{\varphi^*}(x) = \frac{p(x)}{p(x)+q(x)}.$$

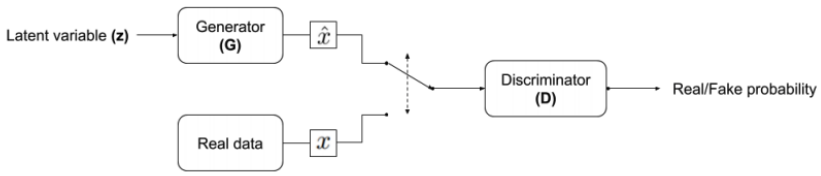
The two-sample test statistic for our setup would be is

$$\hat{t} = \frac{1}{m} \sum_{(x,y)\in \text{validation data}} \mathbf{1}_{d_\varphi(x)>\frac{1}{2}} = y. \tag{26.2}$$

The idea here is that if the data were coming from the same distribution, we would never be able to fit the logistic well enough and its validation accuracy (which is exactly $\hat{t}$ from the above formula) will be near chance level (50%). If the two distributions $p(x)$ and $q(x)$ were different then the validation accuracy would be greater than 50%. This 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).

Remark 188. It can be shown that if the two distributions are not the same, the 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 *validation accuracy* of the discriminator is a good idea.

Remark 189. The second key idea of a GAN is that the generator $g_\theta : \mathcal{Z} \to \mathcal{X}$ that maps some latent space $\mathcal{Z}$ to the image space $\mathcal{X}$ is trained to minimize the power of the two-sample test. The generator g_θ wants to draw samples that look like they came from the true distribution $p(x)$, which will ruin the validation accuracy of the discriminator d_φ.



The GAN loss function is therefore

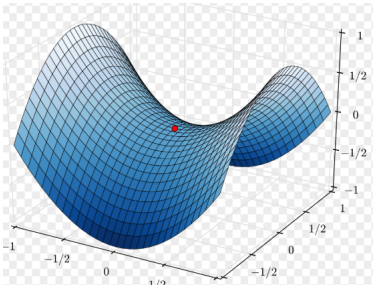
$$\begin{aligned}\ell(\theta, \varphi) &= \min_{\theta} \max_{\varphi} \left\{ E_{x \sim p(x)} \left[\log d_{\varphi}(x) \right] + E_{x \sim q(x)} \left[\log \left(1 - d_{\varphi}(x)\right) \right] \right\} \\ &= \min_{\theta} \max_{\varphi} \left\{ E_{x \sim p(x)} \left[\log d_{\varphi}(x) \right] + E_{z} \left[\log \left(1 - d_{\varphi}(g_{\theta}(z))\right) \right] \right\}.\end{aligned} \tag{26.3}$$

Algorithmically, we sample a mini-batch of training data $\{x_1, \ldots, x_b\}$ and another mini-batch of noise vectors $\{z_1, \ldots, z_b\}$. At each iteration

1. we update the generator using the gradient of the loss with respect to θ.
2. update the discriminator using the gradient of the loss with respect to φ.

A few comments about this objective:

1. 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 decreaes 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 either of the min/max players we risk either 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. This is why the specifics of the hyper-parameters are so tricky to pick while training GANs (this is often called "instability of training").

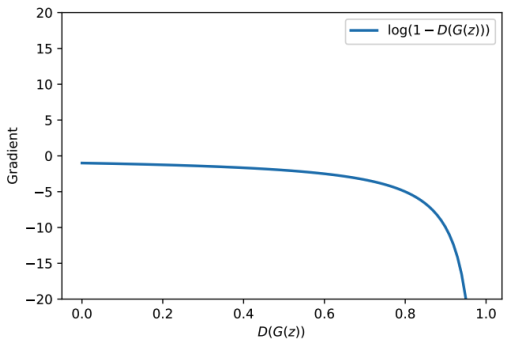
2. The above point is only one reason why GANs are hard to train. They are mathematically sound models but the trouble lies in how we implement them. We decided to learn a two-sample test because it is difficult to find statistics $\hat{t}$ that work well in high dimensions. Why should we expect that a neural network will be able to learn a good statistic?

3. This is the reason why there is a large amount of effort being spent in designing new statistics in GANs. As usual, there is no right or wrong answer to which loss function is better or worse, it depends upon the data at hand.

4. Even the vanilla GAN objective is quite tricky to make work. For example, note that the generator has a much more difficult task than the discriminator. During the 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. Note that the gradient of the second term is

$$\nabla_\theta \log(1 - d_\varphi(g_\theta(z))) = \frac{1}{1 - d_\varphi(g_\theta(z))} \nabla_\theta 1 - d_\varphi(g_\theta(z)).$$

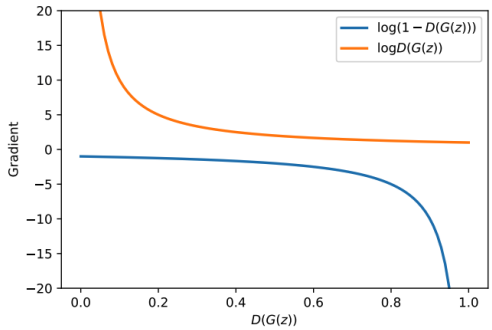
As a function of $d_\varphi(g_\theta(z))$ the second term looks like



In other words, the gradient is essentially zero if the generator g_θ is not working well and the discriminator d_φ easily predicts zero. This does not allow the generator to learn well; this is essentially like your advisor shooting down all your ideas. Most GAN implementations therefore modify the second term to be

$$-\mathop{\mathbb{E}}_{z}\left[\log d_\varphi(g_\theta(z))\right]$$

which does not suffer from the small gradient problem and is still minimized when $d_\varphi(g_\theta(z)) = 1$.



What is the validation procedure for a GAN? How do we check if a GAN is working well?

1. Run the generator a few times to eyeball the quality of images it generates

   But this is a very heuristic and qualitative metric.

2. A number of other metrics exist for evaluating generative models. E.g., 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 that of the FID between two Gaussian distributions p, q is the Wasserstein distance between the two densities, i.e., it measures how easy it is to transport probability mass from one Gaussian to make it look like the other Gaussian. There are others such as the Maximum Mean Discrepency (MMD) score or the explicitly computing the Wasserstein

distance between the two probability distributions. 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 indeed difficult to measure when two distributions are the same. However, the problem is exacerbated by the fact that large deep networks are terrific at over-fitting.

Remark 190. 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.

26.3 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

Achille, A. and Soatto, S. (2018). Emergence of invariance and disentanglement in deep representations. *The Journal of Machine Learning Research*, 19(1):1947–1980.

Alemi, A. A., Fischer, I., Dillon, J. V., and Murphy, K. (2016). Deep variational information bottleneck. *arXiv preprint arXiv:1612.00410*.

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. *Journal of Machine Learning Research*, 20(63):1–17.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828.

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.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*.

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.

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*.

Dziugaite, G. K. and Roy, D. M. (2017). Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*.

Einstein, A. (1956). Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen. *Annalen der Physik*.

Freeman, C. D. and Bruna, J. (2016). Topology and geometry of half-rectified network optimization. *arXiv preprint arXiv:1611.01540*.

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.

Hardt, M. and Ma, T. (2016). Identity matters in deep learning. *arXiv preprint arXiv:1611.04231*.

Johnson, R. and Zhang, T. (2013). Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323.

Jordan, R., Kinderlehrer, D., and Otto, F. (1998). The variational formulation of the fokker–planck equation. *SIAM journal on mathematical analysis*, 29(1):1–17.

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., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

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.

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*.

Mandt, S., Hoffman, M., and Blei, D. (2016). A variational analysis of stochastic gradient algorithms. In *International conference on machine learning*, pages 354–363.

Mathieu, E., Rainforth, T., Siddharth, N., and Teh, Y. W. (2018). Disentangling disentanglement in variational autoencoders.

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.

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.

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.

Ruppert, D. (1988). Efficient estimations from a slowly convergent robbins-monro process. Technical report, Cornell University Operations Research and Industrial Engineering.

Shwartz-Ziv, R. and Tishby, N. (2017). Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv:1412.6806*.

Su, W., Boyd, S., and Candes, E. (2014). A differential equation for modeling nesterov's accelerated gradient method: Theory and insights. In *Advances in Neural Information Processing Systems*, pages 2510–2518.

Tishby, N., Pereira, F. C., and Bialek, W. (2000). The information bottleneck method. *arXiv preprint physics/0004057*.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265.

Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer.

Valiant, L. (2013). *Probably Approximately Correct: NatureÕs Algorithms for Learning and Prospering in a Complex World*. Basic Books (AZ).

Venturi, L., Bandeira, A. S., and Bruna, J. (2018). Spurious valleys in two-layer neural network optimization landscapes. *arXiv preprint arXiv:1802.06384*.

Welling, M. and Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688.

Wibisono, A., Wilson, A. C., and Jordan, M. I. (2016). A variational perspective on accelerated methods in optimization.

Wiener, N. (1965). *Cybernetics or Control and Communication in the Animal and the Machine*, volume 25. MIT press.