Lecture 1: Introduction

Roger Grosse

This series of readings forms the lecture notes for the course CSC421, "Neural Networks and Deep Learning," for undergraduates at the University of Toronto. I'm aiming for it also to function as a stand-alone mini-textbook for self-directed learners and for students at other universities. These notes are aimed at students who have some background in basic calculus, probability theory, and linear algebra, but possibly no prior background in machine learning.

1 Motivation

1.1 Why machine learning?

Think about some of the things we do effortlessly on a day-to-day basis: visually recognize people, places and things, pick up objects, understand spoken language, and so on. How would you program a machine to do these things? Unfortunately, it's hard to give a step-by-step program, since we have very little introspective awareness of the workings of our minds. How do you recognize your best friend? Exactly which facial features do you pick up on? AI researchers tried for decades to come up with computational procedures for these sorts of tasks, and it proved frustratingly difficult.

Machine learning takes a different approach: collect lots of data, and have an algorithm *automatically* figure out a good behavior from the data. If you're trying to write a program to distinguish different categories of objects (tree, dog, etc.), you might first collect a dataset of images of each kind of object, and then use a machine learning algorithm to train a model (such as a neural network) to classify an image as one category or another. Maybe it will learn to see in a way analogous to the human visual system, or maybe it will come up with a different approach altogether. Either way, the whole process can be much easier than specifying everything by hand.

Aside from being easier, there are lots of other reasons we might want to use machine learning to solve a given problem:

- A system might need to adapt to a changing environment. For instance, spammers are constantly trying to figure out ways to trick our e-mail spam classifiers, so the classification algorithms will need to constantly adapt.
- A learning algorithm might be able to perform *better* than its human programmers. Learning algorithms have become world champions at a variety of games, from checkers to chess to Go. This would be impossible if the programs were only doing what they were explicitly told to.

- We may want an algorithm to behave autonomously for privacy or fairness reasons, such as with ranking search results or targeting ads.

Here are just a few important applications where machine learning algorithms are regularly deployed:

- Detecting credit card fraud
- Determining when to apply a C-section
- Transcribing human speech
- Recognizing faces
- Robots learning complex behaviors

1.2 How is machine learning different from statistics?

A lot of the algorithms we cover in this course originally came from statistics: linear regression, principal component analysis (PCA), maximum likelihood estimation, Bayesian parameter estimation, and Expectation-Maximization (EM). (Statisticians got there first because we had data before we had computers.) Much of machine learning, from the most basic techniques to the state-of-the-art algorithms presented at research conferences, is statistical in flavor. It's unsurprising that there should be overlap, since both fields are fundamentally concerned with the question of how to learn things from data.

What, then, is different about machine learning? Opinions will differ on this question, but if I had to offer one rule of thumb, it's this: statistics is motivated by guiding human decision making, while machine learning is motivated by autonomous agents. This means that, even when we talk about the same algorithm, practitioners in the two fields are likely to ask different questions. Statisticians might put more emphasis on being able to interpret the results of an algorithm, or being able to rigorously determine whether a certain observed pattern might have just happened by chance. Machine learning practitioners might put more emphasis on algorithms that can perform well in a variety of situations without human intervention. This overlap in techniques, coupled with the differences in motivation, creates a lot of awkwardness as practitioners in both fields will talk past each other without realizing it.

1.3 Why a course on neural networks?

Neural networks are one particular approach to machine learning, *very* loosely inspired by how the brain processes information. A neural network is composed of a large number of *units*, each of which does very simple computations, but which produce sophisticated behaviors in aggregate. There are lots of other widely used approaches to machine learning, but this class focuses on neural networks for several reasons:

- Neural nets are becoming very widely used in the software industry. They underlie systems for speech recognition, translation, ranking search results, face recognition, sentiment analysis, image search, and many other applications. It's an important tool to know.

- There are powerful software packages like Caffe, Theano, Torch, and TensorFlow, which allow us to quickly implement sophisticated learning algorithms.
- Many of the important algorithms are much simpler to explain, compared with other subfields of machine learning. This makes it possible for undergraduates to quickly get up to speed on state-of-the-art techniques in the field.

This class is very unusual among undergrad classes, in that it covers modern research techniques, i.e. algorithms introduced in the last 5 years. It's pretty amazing that with less than a page of code, we can build learning algorithms more powerful than the best ones researchers had come up with as of 5 years ago.

In fact, these software packages make neural nets *deceptively* easy. One might wonder, if you can implement a neural net in TensorFlow using a handful of lines of code, why do we need a whole class on the subject? The answer is that the algorithms generally won't work perfectly the first time. Diagnosing and fixing the problems requires careful detective work and a sophisticated understanding of what's going on beneath the hood. In this class, we'll work from the bottom up: we'll derive the algorithms mathematically, implement them from scratch, and only then look at the out-of-the-box implementations. This will help us build up the depth of understanding we need to reason about how an algorithm is behaving.

2 Types of machine learning

I said above that in machine learning, we collect lots of data, and then train a model to learn a particular behavior from it. But what kind of data do we collect? The answer will determine what sort of learning algorithm we'll apply to any given problem. Roughly speaking, there are three different types of machine learning:

- In **supervised learning**, we have examples of the desired behavior. For instance, if we're trying to train a neural net to distinguish cars and trucks, we would collect images of cars and trucks, and label each one as a car or a truck.
- In **reinforcement learning**, we don't have examples of the behavior, but we have some method of determining how good a particular behavior was — this is known as a *reward signal*. (By analogy, think of training dogs to perform tricks.) One example would be training an agent to play video games, where the reward signal is the player's score.
- In **unsupervised learning**, we have neither labels nor a reward signal. We just have a bunch of data, and want to look for patterns in the data. For instance, maybe we have lots of examples of patients with autism, and want to identify different subtypes of the condition.

This taxonomy is a vast oversimplification, but it will still help us to organize the algorithms we cover in this course. Now let's look at some examples from each category.

2.1 Supervised learning

The majority of this course will focus on supervised learning. This is the best-understood type of machine learning, because (compared with unsupervised and reinforcement learning) supervised learning problems are much easier to assign a mathematically precise formulation that matches what one is trying to achieve. In general, one defines a **task**, where the algorithm's goal is to train a **model** which takes an **input** (such as an image) and predicts a **target** (such as the object category). One collects a dataset consisting of pairs of inputs and **labels** (i.e. true values of the target). A subset of the data, called the **training set**, is used to train the model, and a separate subset, called the **test set**, is used to measure the algorithm's performance. There are a lot of highly effective and broadly applicable supervised learning algorithms, many of which will be covered in this course.

For several decades, **image classification** has been perhaps *the* prototypical application of neural networks. In the late 1980s, the US Postal Service was interested in automatically reading handwritten zip codes, so they collected 9,298 examples of handwritten digits (0-9), given as 16×16 images, and labeled each one; the task is to predict the digit class from the image. This dataset is now known as the USPS Dataset[1]. In the terminology of supervised learning, we say that the input is the image, and the target is the digit class. By the late 1990s, neural networks were good enough at this task that they became regularly used to sort letters.

In the 1990s, researchers collected a similar but larger handwritten digit dataset called MNIST[2]; for decades, MNIST has served as the "fruit fly" of neural network research. I.e., even though handwritten digit classification is now considered too easy a problem to be of practical interest, MNIST has been used for almost two decades to benchmark neural net learning algorithms. Amazingly, this classic dataset continues to yield algorithmic insights which generalize to challenging problems of more practical interest.

A more challenging task is to classify full-size images into object categories, a task known as **object recognition**. The ImageNet dataset[3] consists of 14 million images of nearly 22,000 distinct object categories. A (still rather large) subset of this dataset, containing 1.2 million images in 1000 object categories, is currently one of the most important benchmarks for computer vision algorithms; this task is known as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Since 2012, all of the best-performing algorithms have been neural networks. Recently, progress on the ILSVRC has been extremely rapid, with the error rate[4] dropping from 25.7% to 5.7% over the span of a few years!

All of the above examples concerned image classification, where the goal is to predict a discrete category for each image. A closely related task is **object detection**, where the task is to identify all of the objects present in their image, as well as their locations. I.e., the input is an image, and the target is a listing of object categories together with their bounding boxes.

[1] http://statweb.stanford.edu/~tibs/ElemStatLearn/data.html

[2] http://yann.lecun.com/exdb/mnist/

[3] http://www.image-net.org/

[4] In particular, the top-5 error rate; the algorithm predicts 5 object categories, and gets it right if any of the 5 is correct.

Other variants include **localization**, where one is given a list of object categories and has to predict their locations, and **semantic segmentation**, where one tries to label each pixel of an image as belonging to an object category. There are a huge variety of different supervised learning problems related to image understanding, depending on exactly what one is hoping to achieve. The variety of tasks can be bewildering, but fortunately we can approach most of them using very similar principles.

Neural nets have been applied in lots of areas other than vision. Another important problem domain is language. Consider, for example, the problem of **machine translation**. The task is to translate a sentence from one language (e.g. French) to another language (e.g. English). One has available a large corpus of French sentences coupled with their English translations; a good example is the proceedings of the Canadian Parliament. Observe that this task is more complex than image classification, in that the target is an entire sentence. Observe also that there generally won't be a unique best translation, so it may be preferable for the algorithm to return a *probability distribution* over possible translations, rather than a single translation. This ambiguity also makes evaluation difficult, since one needs to distinguish almost-correct translations from completely incorrect ones.

The general category of supervised learning problem where the inputs and targets are both sequences is known as **sequence-to-sequence learning**. The sequences need not be of the same type. An important example is **speech recognition**, where one is given a speech waveform and wants to produce a transcription of what was said. Neural networks led to dramatic advances in speech recognition around 2010, and form the basis of all of the modern systems. **Caption generation** is a task which combines vision and language understanding; here the task is to take an image and return a textual description of the image. The most successful approaches are based on neural nets. Caption generation is far from a solved problem, and the systems can be fun to experiment with, not least because of their entertaining errors.[5]

2.2 Reinforcement Learning

The second type of learning problem is reinforcement learning. Here, one doesn't have labels of the correct behavior, but instead has a way of quantitatively evaluating how good a behavior was; this is known as the **reward signal**. Reinforcement learning problems generally involve an **agent** situated in an **environment**. In each time step, the agent has available a set of **actions** which (either deterministically or stochastically) affect the **state** of the agent and the environment. The goal is to learn a **policy**, determining which action to perform depending on the state, in order to achieve has high a reward as possible on average.

Throughout the history of AI, a lot of progress has been driven by game playing. Over the years, AIs have come to defeat human champions in board games of increasing complexity, including backgammon, checkers, chess, and Go. In the case of Go, the success was achieved by a neural network called AlphaGo. Most of these games involve playing against an opponent,

[5] http://deeplearning.cs.toronto.edu/i2t

or **adversary**; this adversarial setting is beyond the scope of this class. However, single-player games can be formulated as reinforcement learning problems. For instance, we will look at the example of training an agent to play classic Atari games. The agent observes the pixels on the screen, has a set of actions corresponding to the controller buttons, and receives rewards corresponding to the score of the game. Neural net algorithms have outperformed humans on many games, in the sense of being able to achieve a high score in a short period of time.

2.3 Unsupervised Learning

The third type of machine learning, where one has neither labels of the correct behavior nor a reward signal, is known as unsupervised learning. Here, one simply has a collection of data and is interested in finding patterns in the data. We will just barely touch upon unsupervised learning in this class, because compared with supervised and reinforcement learning, the principles are less well understood, the algorithms are more mathematically involved, and one must account for a lot more domain-specific structure.

One of the most important types of unsupervised learning is **distribution modeling**, where one has an unlabeled dataset (such as a collection of images or sentences), and the goal is to learn a probability distribution which matches the dataset as closely as possible. In principle, one should be able to **generate from**, or draw samples from, the distribution, and those samples should be indistinguishable from the original data. Sometimes we care about the samples themselves, e.g. if we want to generate images of textures for graphics applications. Another important use of distribution models is to resolve ambiguities; for instance, in speech recognition, "recognize speech" may sound very similar to "wreck a nice beach," but a good distribution model ought to be able to tell us that the former is a more likely explanation than the latter.

Another important use of unsupervised learning is to recover **latent structure**, or high-level explanations that yield insight into the structure underlying the data. One important example is **clustering**, where one is interested in dividing a set of data points into **clusters**, where data points assigned to the same cluster are similar, and data points assigned to different clusters are dissimilar. Much fancier models are possible as well. For instance, a biology lab was running behavior genetics experiments on mice, and wanted to automatically analyze videos of mice to determine whether one genetic variant was more likely to engage in a particular behavior than another variant. If experts had explicitly labeled different behaviors, this would be a supervised learning problem; however, the lab avoided doing this because it would have introduced human biases into the interpretation. Instead, they ran an unsupervised learning algorithm to automatically analyze mouse videos and group them into different categories of behaviors.

3 Neural nets and the brain

The **neuron** is the basic unit of processing in the brain. It has a broad, branching tree of **dendrites**, which receive chemical signals from other neurons at junctions called **synapses**, and convert these into electrical signals.

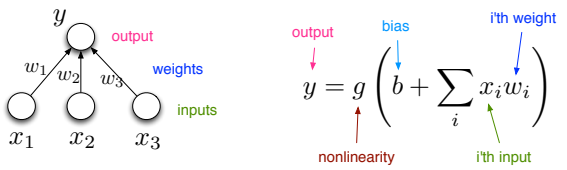


Figure 1: Simplified neuron-like processing unit.

The dendrites integrate these electrical signals in complex, nonlinear ways, and if the combined signal is strong enough, the neuron generates an **action potential**. This is an electrical signal that's propagated down the neuron's **axon**, which eventually causes the neuron to release chemical signals at its synapses with other neurons. Those neurons then integrate their incoming signals, and so on.

In machine learning, we abstract away nearly all of this complexity, and use an extremely simplified model of a neuron shown in Figure [[]]. This neuron has a set of incoming **connections** from other neurons, each with an associated strength, or **weight**. It computes a value, called the **pre-activation**, which is the sum of the incoming signals times their weights:

$$z = \sum_j w_j x_j + b.$$

The scalar value b, called a **bias**, determines the neuron's activation in the absence of inputs. The pre-activation is passed through a **nonlinearity** ϕ (also called an **activation function**) to compute the **activation** $a = \phi(z)$. Examples of nonlinearities include the **logistic sigmoid**

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

and **linear rectification**

$$\phi(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0. \end{cases}$$

In summary, the activation is computed as

$$a = \phi\left(\sum_j w_j x_j + b\right).$$

That's it. That's all that our idealized neurons do. Note that the whole idea of a continuous-valued activation is biologically unrealistic, since a real neuron's action potentials are an all-or-nothing phenomenon: either they happen or they don't, and they do not vary in strength. The continuous-valued activation is sometimes thought of as representing a "firing rate," but mostly we just ignore the whole issue and don't even think about the relationships with biology. From now on, we'll refer to these idealized neurons using the more scientifically neutral term **units**, rather than neurons.

If the relationship with biology seems strained, it gets even worse when we talk about learning, i.e. adapting the weights of the neurons. Most

modern neural networks are trained using a procedure called **backpropagation**, where each neuron propagates error signals backwards through its incoming connections. Nothing analogous has been observed in actual biological neurons. There have been some creative proposals for how biological neurons might implement something like backpropagation, but for the most part we just ignore the issue of whether our neural nets are biologically realistic, and simply try to get the best performance we can out of the tools we have. (There is a separate field called theoretical neuroscience, which builds much more accurate models of neurons, towards the goal of understanding better how the brain works. This field has produced lots of interesting insights, and has achieved accurate quantitative models of some neural systems, but so far there doesn't appear to be much practical benefit to using more realistic neuronal models in machine learning systems.)

However, neural networks do share one important commonality with the brain: they consist of a very large number of computational units, each of which performs a rather simple set of operations, but which in aggregate produce very sophisticated and complex behaviors. Most of the models we'll discuss in this course are simply large collections of units, each of which computes a linear function followed by a nonlinearity.

Another analogy with the brain is worth pointing out: the brain is organized into hierarchies of processing, where different brain regions encode information at different levels of abstraction. Information processing starts at the retina of the eye, where neurons compute simple center-surround functions of their inputs. Signals are passed to the primary visual cortex, where (to vastly oversimplify things) cells detect simple image features such as edges. Information is passed through several additional "layers" of processing, each one taking place in a different brain region, until the information reaches areas of the cortex which encode things at a high level of abstraction. For instance, individual neurons in the infero-temporal cortex have been shown (again, vastly ovsersimplifying) to encode the identities of objects.

In summary, visual information is processed in a series of layers of increasing abstraction. This inspired machine learning researchers to build neural networks which are many layers deep, in hopes that they would learn analogous representations where higher layers represent increasingly abstract features. In the last 5 years or so, very deep networks have indeed been found to achieve startlingly good performance on a wide variety of problems in vision and other application areas; for this reason, the research area of neural networks is often referred to as **deep learning**. There is some circumstantial evidence that deep networks learn hierarchical representations, but this is notoriously difficult to analyze rigorously.

4 Software

There are a lot of software tools that make it easy to build powerful and sophisticated neural nets. In this course, we will use the programming language **Python**, a friendly but powerful high-level language which is widely used both in introductory programming courses and a wide variety of production systems. Because Python is an interpreted language, executing a

line of Python code is very slow, perhaps hundreds of times slower than the C equivalent. Therefore, we never write algorithms directly using for-loops in Python. Instead, we **vectorize** the algorithms by expressing them in terms of operations on matrices and vectors; those operations are implemented in an efficient low-level language such as C or Fortran. This allows a large number of computational operations to be performed with minimal interpreter overhead. In this course, we will use the **NumPy** library, which provides an efficient and easy-to-use array abstraction in Python.

Ten years ago, most neural networks were implemented directly on top of a linear algebra framework like NumPy, or perhaps a lower level programming language when efficiency was especially critical. More recently, a variety of powerful neural net frameworks have been developed, including **Torch**, **Caffe**, **Theano**, **TensorFlow**, and **PyTorch**. These frameworks make it easy to quickly implement a sophisticated neural net model. Here are some of the features provided by some or all of these frameworks (we'll use TensorFlow as an example):

- **Automatic differentiation.** If one implements a neural net directly on top of NumPy, much of the implementational work involves writing procedures to compute derivatives. TensorFlow automatically constructs routines for computing derivatives which are generally at least as efficient as the ones we would have written by hand.

- **Compiling computation graphs.** If we implement a network in NumPy, a lot of time is wasted allocating and deallocating memory for matrices. TensorFlow takes a different approach: you first build a graph defining the network's computation, and TensorFlow figures out an efficient strategy for performing those computations. It handles memory efficiently and performs some other code optimizations.

- **Libraries of algorithms and network primitives.** Lots of different neural net primitives and training algorithms have been proposed in the research literature, and many of these are made available as black boxes in TensorFlow. This makes it easy to iterate with different choices of network architecture and training algorithm.

- **GPU support.** While NumPy is much faster than raw Python, it's not nearly fast enough for modern neural nets. Because neural nets consist of a large collection of simple processing units, they naturally lend themselves to parallel computation. **Graphics processing units (GPUs)** are a particular parallel architecture which has been especially powerful in training neural nets. It can be a huge pain to write GPU routines at a low level, but TensorFlow provides an easy interface so that the same code can run on either a CPU or a GPU.

For this course, we'll use two neural net frameworks. The first is **Autograd**, a lightweight automatic differentiation library. It is simple enough that you will be able to understand how it is implemented; while it is missing many of the key features of PyTorch or TensorFlow, it provides a useful mental model for reasoning about those frameworks.

For roughly the second half of the course, we will use **PyTorch**, a powerful and widely used neural net framework. It's not quite as popular as

TensorFlow, but we think it is easier to learn. But once you are done with this course, you should find it pretty easy to pick up any of the other frameworks.

Lecture 2, Part 1: Linear regression

Roger Grosse

1 Introduction

Let's jump right in and look at our first machine learning algorithm, **linear regression**. In regression, we are interested in predicting a scalar-valued target, such as the price of a stock. By linear, we mean that the target must be predicted as a linear function of the inputs. This is a kind of supervised learning algorithm; recall that, in supervised learning, we have a collection of training examples labeled with the correct outputs.

Regression is an important problem in its own right. But today's discussion will also highlight a number of themes which will recur throughout the course:

- Formulating a machine learning task mathematically as an optimization problem.
- Thinking about the data points and the model parameters as vectors.
- Solving the optimization problem using two different strategies: deriving a closed-form solution, and applying gradient descent. These two strategies are how we will derive nearly all of the learning algorithms in this course.
- Writing the algorithm in terms of linear algebra, so that we can think about it more easily and implement it efficiently in a high-level programming language.
- Making a linear algorithm more powerful using basis functions, or features.
- Analyzing the generalization performance of an algorithm, and in particular the problems of overfitting and underfitting.

1.1 Learning goals

- Know what objective function is used in linear regression, and how it is motivated.
- Derive both the closed-form solution and the gradient descent updates for linear regression.
- Write both solutions in terms of matrix and vector operations.
- Be able to implement both solution methods in Python.

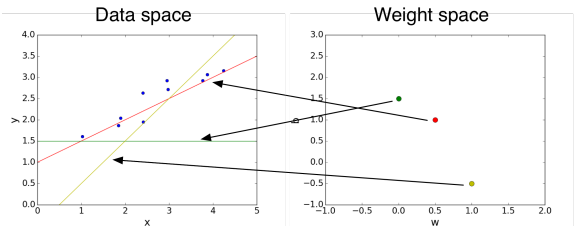


Figure 1: Three possible hypotheses for a linear regression model, shown in data space and weight space.

- Know how linear regression can learn nonlinear functions using feature maps.
- What is meant by generalization, overfitting, and underfitting? How can we measure generalization performance in practice?

2 Problem setup

In order to formulate a learning problem mathematically, we need to define two things: a model and a loss function. The **model**, or **architecture** defines the set of allowable **hypotheses**, or functions that compute predictions from the inputs. In the case of linear regression, the model simply consists of linear functions. Recall that a linear function of D inputs is parameterized in terms of D coefficients, which we'll call the **weights**, and an intercept term, which we'll call the **bias**. Mathematically, this is written as:

$$y = \sum_j w_j x_j + b. \tag{1}$$

Figure 1 shows two ways to visualize linear models. In this case, the data are one-dimensional, so the model reduces to simply $y = wx + b$. On one side, we have the **data space**, or **input space**, where t is plotted as a function of x. Three different possible linear fits are shown. On the other side, we have **weight space**, where the corresponding pairs (w, b) are plotted.

You should study these figures and try to understand how the lines in the left figure map onto the X's on the right figure. Think back to middle school. Hint: w is the slope of the line, and b is the y-intercept.

Clearly, some of these linear fits are better than others. In order to quantify how good the fit is, we define a **loss function**. This is a function $\mathcal{L}(y, t)$ which says how far off the prediction y is from the target t. In linear regression, we use **squared error**, defined as

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2. \tag{2}$$

This is small when y and t are close together, and large when they are far apart. In general, the value $y - t$ is known as the **residual**, and we'd like the residuals to be close to zero.

Why is there the factor of 1/2 in front? It just makes the calculations convenient.

When we combine our model and loss function, we get an optimization problem, where we are trying to minimize a **cost function** with respect to the model parameters (i.e. the weights and bias). The cost function is simply the loss, averaged over all the training examples. When we plug in

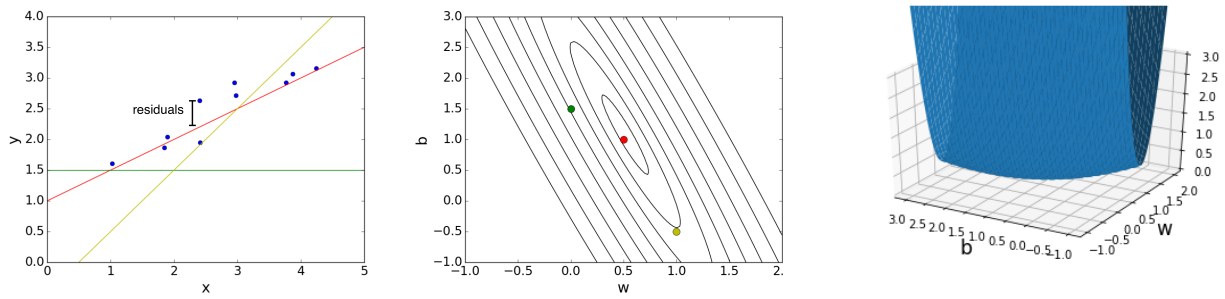


Figure 2: **Left:** three hypotheses for a regression dataset. **Middle:** Contour plot of least-squares cost function for the regression problem. Colors of the points match the hypotheses. **Right:** Surface plot matching the contour plot. Surface plots are usually hard to interpret, so we won't look at them very often.

the model definition (Eqn. 1), we get the following cost function:

$$\mathcal{J}(w_1, \ldots, w_D, b) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)}) \tag{3}$$

$$= \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)}\right)^2 \tag{4}$$

$$= \frac{1}{2N} \sum_{i=1}^{N} \left(\sum_j w_j x_j^{(i)} + b - t^{(i)}\right)^2 \tag{5}$$

Our goal is to choose $w_1, \ldots, w_D$ and b to minimize $\mathcal{J}$. Note the difference between the loss function and the cost function. The loss is a function of the predictions and targets, while the cost is a function of the model parameters. The cost function is visualized in Figure 2.

The distinction between loss functions and cost functions will become clearer in a later lecture, when the cost function is augmented to include more than just the loss — it will also include a term called a regularizer which encourages simpler hypotheses.

3 Solving the optimization problem

In order to solve the optimization problem, we'll need the concept of partial derivatives. If you haven't seen these before, then you should go learn about them, on Khan Academy.[1] Just as a quick recap, suppose f is a function of $x_1, \ldots, x_D$. Then the partial derivative $\partial f / \partial x_i$ says in what way the value of f changes if you increase x_i by a small amount, while holding the rest of the arguments fixed. We can evaluate partial derivatives using the tools of single-variable calculus: to compute $\partial f / \partial x_i$ simply compute the (single-variable) derivative with respect to x_i, treating the rest of the arguments as constants.

Whenever we want to solve an optimization problem, a good place to start is to compute the partial derivatives of the cost function. Let's do that in the case of linear regression. Applying the chain rule for derivatives

[1] https://www.khanacademy.org/math/calculus-home/multivariable-calculus/multivariable-derivatives#partial-derivatives

to Eqn. 5, we get

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N} x_j^{(i)} \left(\sum_{j'} w_{j'} x_{j'}^{(i)} + b - t^{(i)} \right) \tag{6}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} \left(\sum_{j'} w_{j'} x_{j'}^{(i)} + b - t^{(i)} \right). \tag{7}$$

It's possible to simplify this a bit — notice that part of the term in parentheses is simply the prediction. The partial derivatives can be rewritten as:

It's always a good idea to try to simplify equations by finding familiar terms.

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N} x_j^{(i)} (y^{(i)} - t^{(i)}) \tag{8}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} - t^{(i)}. \tag{9}$$

Now, it's good practice to do a sanity check of the derivatives. For instance, suppose we overestimated all of the targets. Then we should be able to improve the predictions by decreasing the bias, while holding all of the weights fixed. Does this work out mathematically? Well, the residuals $y^{(i)} - t^{(i)}$ will be positive, so based on Eqn. 9, $\partial \mathcal{J}/\partial b$ will be positive. This means increasing the bias will increase $\mathcal{J}$, and deceasing the bias will decrease $\mathcal{J}$ — which matches up with our expectation. So Eqn. 9 is plausible. Try to come up with a similar sanity check for $\partial \mathcal{J}/\partial w_j$.

Later in this course, we'll introduce a more powerful way to test partial derivative computations, but you should still get used to doing sanity checks on all your computations!

Now how do we use these partial derivatives? Let's discuss the two methods which we will use throughout the course.

3.1 Direct solution

One way to compute the minimum of a function is to set the partial derivatives to zero. Recall from single variable calculus that (assuming a function is differentiable) the minimum $x^\star$ of a function f has the property that the derivative $\mathrm{d}f/\mathrm{d}x$ is zero at $x = x^\star$. Note that the converse is not true: if $\mathrm{d}f/\mathrm{d}x = 0$, then $x^\star$ might be a maximum or an inflection point, rather than a minimum. But the minimum can only occur at points that have derivative zero.

An analogous result holds in the multivariate case: if f is differentiable, then all of the partial derivatives $\partial f/\partial x_i$ are zero at the minimum. The intuition is simple: if $\partial f/\partial x_i$ is positive, then one can decrease f slightly by decreasing x_i slightly. Conversely, if $\partial f/\partial x_i$ is negative, then one can decrease f slightly by increasing x_i slightly. In either case, this implies we're not at the minimum. Therefore, if the minimum exists (i.e. f doesn't keep growing as x goes to infinity), it occurs at a **critical point**, i.e. a point where the partial derivatives are zero. This gives us a strategy for finding minima: set the partial derivatives to zero, and solve for the parameters. This method is known as **direct solution**.

Let's apply this to linear regression. For simplicity, let's assume the model doesn't have a bias term. (We actually don't lose anything by getting

rid of the bias. Just add a "dummy" input x_0 which always takes the value 1; then the weight w_0 acts as a bias.) We simplify Eqn. 6 to remove the bias, and set the partial derivatives to zero:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} \left(\sum_{j'=1}^{D} w_{j'} x_{j'}^{(i)} - t^{(i)} \right) = 0 \tag{10}$$

Since we're trying to solve for the weights, let's pull these out:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{j'=1}^{D} \left(\sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)} \right) w_{j'} - \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)} = 0 \tag{11}$$

The details of this equation aren't important; what's important is that we've wound up with a system of D linear equations in D variables. In other words, we have the system of linear equations

$$\sum_{j'=1}^{D} A_{jj'} w_{j'} - c_j = 0 \qquad \forall j \in \{1, \ldots, D\}, \tag{12}$$

where $A_{jj'} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)}$ and $c_j = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)}$. As computer scientists, we're done, because this gives us an algorithm for finding the optimal regression weights: we first compute all the values $A_{jj'}$ and c_j, and then solve the system of linear equations using a linear algebra library such as NumPy. (We'll give an implementation of this later in this lecture.)

Note that the solution we just derived is *very* particular to linear regression. In general, the system of equations will be *nonlinear*, and except in rare cases, systems of nonlinear equations don't have closed form solutions. Linear regression is very unusual, in that it has a closed-form solution. We'll only be able to come up with closed form solutions for a handful of the algorithms we cover in this course.

3.2 Gradient descent

Now let's minimize the cost function a different way: **gradient descent**. This is an example of an **iterative algorithm**, which means that we apply a certain update rule over and over again, and if we're lucky, our **iterates** will gradually improve according to our objective function. To do gradient descent, we initialize the weights to some value (e.g. all zeros), and repeatedly adjust them in the direction that most decreases the cost function. If we visualize the cost function as a surface, so that lower is better, this is the direction of **steepest descent**. We repeat this procedure until the iterates **converge**, or stop changing much. (Or, in practice, we often run it until we get tired of waiting.) If we're lucky, the final iterate will be close to the optimum.

In order to make this mathematically precise, we must introduce the **gradient**, the direction of steepest ascent (i.e. fastest increase) of a function. The entries of the gradient vector are simply the partial derivatives with respect to each of the variables:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix} \tag{13}$$

The reason that this formula gives the direction of steepest ascent is beyond the scope of this course. (You would learn about it in a multivariable calculus class.) But this suggests that to decrease a function as quickly as possible, we should update the parameters in the direction opposite the gradient.

We can formalize this using the following update rule, which is known as **gradient descent**:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}, \tag{14}$$

or in terms of coordinates,

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}. \tag{15}$$

The symbol $\leftarrow$ means that the left-hand side is updated to take the value on the right-hand side; the constant α is known as a **learning rate**. The larger it is, the larger a step we take. We'll talk in much more detail later about how to choose a learning rate, but in general it's good to choose a small value such as 0.01 or 0.001. If we plug in the formula for the partial derivatives of the regression model (Eqn. 8), we get the update rule:

$$w_j \leftarrow w_j - \alpha \frac{1}{N} \sum_{i=1}^{N} x_j (y^{(i)} - t^{(i)}) \tag{16}$$

In practice, we rarely if ever go through this last step. From a software engineering perspective, it's better to write our code in a modular way, where one function computes the gradient, and another function implements gradient descent, taking the gradient as given.

So we just repeat this update lots of times. What does gradient descent give us in the end? For analyzing iterative algorithms, it's useful to look for **fixed points**, i.e. points where the iterate doesn't change. By inspecting Eqn. 14, setting the left-hand side equal to the right-hand side, we see that the fixed points occur where $\partial \mathcal{J} / \partial \mathbf{w} = 0$. Since we know the gradient must be zero at the optimum, this is an encouraging sign that maybe it will converge to the optimum. But there are lots of things that could go wrong, such as divergence or local optima; we'll look at these in more detail in a later lecture.

Lecture 9 discusses optimization issues.

You might ask: by setting the partial derivatives to zero, we compute the exact solution. With gradient descent, we never actually reach the optimum, but merely approach it gradually. Why, then, would we ever prefer gradient descent? Two reasons:

1. We can only solve the system of equations explicitly for a handful of models. By contrast, we can apply gradient descent to any model for which we can compute the gradient. This is usually pretty easy to do efficiently. Importantly, it can usually be done *automatically*, so software packages like Theano and TensorFlow can save us from ever having to compute partial derivatives by hand.

2. Solving a large system of linear equations can be expensive, possibly many orders of magnitude more expensive than a single gradient descent update. Therefore, gradient descent can sometimes find a reasonable solution much faster than solving the linear system. Therefore, gradient descent is often more practical than computing exact solutions, even for models where we are able to derive the latter.

For these reasons, gradient descent will be our workhorse throughout the course. We will use it to train almost all of our models, with the exception of a handful for which we can derive exact solutions.

4 Vectorization

Now it's time to bring in linear algebra. We're going to rewrite the linear regression model, as well as both solution methods, in terms of operations on matrices and vectors. This process is known as **vectorization**. There are two reasons for doing this:

Vectorization takes a lot of practice to get used to. We'll cover a lot of examples in the first few weeks of the course. I'd recommend practicing these until they start to feel natural.

1. The formulas can be much simpler, more compact, and more readable in this form.

2. Vectorized code can be much faster than explicit `for`-loops, for several reasons.

   - High-level languages like Python can introduce a lot of interpreter overhead, and if we explicitly write a for-loop corresponding to Eqn. 16, this might be 10-100 times slower than the C equivalent. If we instead write the algorithm in terms of a much smaller number of linear algebra operations, then it can perform the same computations much faster with minimal interpreter overhead.
   - Since linear algebra is used all over the place, linear algebra libraries have been extremely well optimized for various computer architectures. Hence, they use much more efficient memory-access patterns than a naïve for-loop, even one written in C.
   - Matrix multiplication is inherently highly parallelizable and involve little control flow. Hence, it's ideal for **graphics processing unit (GPU)** architectures. We're not going to talk much about GPUs in this course, but just think "matrix multiplication + GPU = good". If you run vectorized code on a GPU using a framework like TensorFlow or PyTorch, it may run 50 times faster than the CPU version. As it turns out, most of the computation in deep learning is matrix multiplications, which is why it's been such an incredibly good match for GPUs.

First, we need to represent the data and model parameters in the form of matrices and vectors. If we have N training examples, each D-dimensional, we will represent the inputs as an $N \times D$ matrix $\mathbf{X}$. Each row of $\mathbf{X}$ corresponds to a training example, and each column corresponds to a single input dimension. The weights are represented as a D-dimensional vector $\mathbf{w}$, and the targets are represented as a N-dimensional vector $\mathbf{t}$.

In general, matrices will be denoted with capital boldface, vectors with lowercase boldface, and scalars with plain type.

The predictions are computed using a matrix-vector product

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}, \tag{17}$$

where $\mathbf{1}$ denotes a vector of all ones. We can express the cost function in

vectorized form:

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2 \tag{18}$$

$$= \frac{1}{2N}\|\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t}\|^2. \tag{19}$$

You should stop now and try to show that these equations are equivalent to Eqns. 3–5. The only way you get comfortable with this is by practicing.

Note that this is considerably simpler than Eqn. 5. Even more importantly, it saves us from having to explicitly sum over the indices i and j. As our models get more complicated, we would run out of convenient letters to use as indices if we didn't vectorize.

Now let's revisit the exact solution for linear regression. We derived a system of linear equations, with coefficients $A_{jj'} = \frac{1}{N}\sum_{i=1}^N x_j^{(i)} x_{j'}^{(i)}$ and $c_j = \frac{1}{N}\sum_{i=1}^N x_j^{(i)} t^{(i)}$. In terms of linear algebra, we can write these as the matrix $\mathbf{A} = \frac{1}{N}\mathbf{X}^\top\mathbf{X}$ and $\mathbf{c} = \frac{1}{N}\mathbf{X}^\top\mathbf{t}$. The solution to the linear system $\mathbf{A}\mathbf{w} = \mathbf{c}$ is given by $\mathbf{w} = \mathbf{A}^{-1}\mathbf{c}$ (assuming $\mathbf{A}$ is invertible), so this gives us a formula for the optimal weights:

$$\mathbf{w} = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{t}. \tag{20}$$

An exact solution which we can express with a formula is known as a **closed-form solution**.

Similarly, we can vectorize the gradient descent update from Eqn. 16:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N}\mathbf{X}^\top(\mathbf{y} - \mathbf{t}), \tag{21}$$

where $\mathbf{y}$ is computed as in Eqn. 17.

5 Feature mappings

Linear regression might sound pretty limited. What if the true relationship between inputs and targets is nonlinear? Fortunately, there's an easy way to use linear regression to learn nonlinear dependencies: use a feature mapping. I'll introduce this by way of an example. Suppose we want to approximate it with a cubic polynomial. In other words, we would compute the predictions as:

$$y = w_3 x^3 + w_2 x^2 + w_1 x + w_0. \tag{22}$$

This setting is known as **polynomial regression**.

Let's use the squared error loss function, just as with ordinary linear regression. The important thing to notice is that algorithmically, polynomial regression is *no different* from linear regression. We can apply any of the linear regression algorithms described above, using (x, x^2, x^3) as the inputs. Mathematically, we define a feature mapping $\boldsymbol{\psi}$, in this case

Just as in Section 3.1, we're including a constant feature to account for the bias term, since this simplifies the notation.

$$\boldsymbol{\psi}(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}, \tag{23}$$

and compute the predictions as $y = \mathbf{w}^\top\boldsymbol{\psi}(x)$ instead of $\mathbf{w}^\top\mathbf{x}$. The rest of the algorithm is completely unchanged.

Feature maps are a useful tool, but they're not a silver bullet, for several reasons:

- The features must be known in advance. It's not always easy to pick good features, and up until very recently, feature engineering would take up most of the time and ingenuity in building a practical machine learning system.
- In high dimensions, the feature representations can get very large. For instance, the number of terms in a cubic polynomial is *cubic* in the dimension!

It's possible to work with polynomial feature maps efficiently using something called the "kernel trick," but that's beyond the scope of this course.

In this course, rather than construct feature maps, we will use neural networks to learn nonlinear predictors directly from the raw inputs. In most cases, this eliminates the need for hand-engineering of features.

6 Generalization

We don't just want a learning algorithm to make correct predictions on the training examples; we'd like it to **generalize** to examples it hasn't seen before. The average squared error on novel examples is known as the **generalization error**, and we'd like this to be as small as possible.

Returning to the previous example, let's consider three different polynomial models: (a) a linear function, or equivalently, a degree 1 polynomial; (b) a cubic polynomial; (c) a degree-10 polynomial. The linear function may be too simplistic to describe the data; this is known as **underfitting**. The degree-10 polynomial may be able to fit every training example exactly, but only by learning a crazy function. It would make silly predictions everywhere except the observed data. This is known as **overfitting**. The cubic polynomial is a reasonable compromise. We need to worry about both underfitting and overfitting in pretty much every application of machine learning.

The terms underfitting and overfitting are a bit misleading, since they suggest the two phenomena are mutually exclusive. In fact, most machine learning models suffer from *both* problems simultaneously.

The degree of the polynomial is an example of a **hyperparameter**. Hyperparameters are values that we can't include in the training procedure itself, but which we need to set using some other means. In practice, we normally tune hyperparameters by partitioning the dataset into three different subsets:

Statisticians prefer the term *metaparameter* since *hyperparameter* has a different meaning in statistics.

1. The **training set** is used to train the model.
2. The **validation set** is used to estimate the generalization error of each hyperparameter setting.
3. The **test set** is used at the very end, to estimate the generalization error of the final model, once all hyperparameters have been chosen.

We will talk about validation and generalization in a lot more detail later on in this course.

Lecture 2, Part 2: Linear Classification

Roger Grosse

1 Introduction

Last time, we saw an example of a learning task called regression. There, the goal was to predict a scalar-valued target from a set of features. This time, we'll focus on a slightly different task: **binary classification**, where the goal is to predict a *binary-valued* target. Here are some examples of binary classification problems:

- You want to train a medical diagnosis system to predict whether a patient has a given disease. You have a training set consisting of a set of patients, a set of features for those individuals (e.g. presence or absence of various symptoms), and a label saying whether or not the patient had the disease.
- You are running an e-mail service, and want to determine whether a given e-mail is spam. You have a large collection of e-mails which have been hand-labeled as spam or non-spam.
- You are running an online payment service, and want to determine whether or not a given transaction is fraudulent. You have a labeled training dataset of fraudulent and non-fraudulent transactions; features might include the type of transaction, the amount of money, or the time of day.

Like regression, binary classification is a very restricted kind of task. Most learning problems you'll encounter won't fit nicely into one of these two categories. Our motivation for focusing on binary classification is to introduce several fundamental ideas that we'll use throughout the course. In this lecture, we discuss how to view both data points and linear classifiers as vectors. Next lecture, we discuss the perceptron, a particular classification algorithm, and use it as an example of how to efficiently implement a learning algorithm in Python. Starting next week, we'll look at supervised learning in full generality, and see that regression and binary classification are just special cases of a more general learning framework.

This lecture focuses on the geometry of classification. We'll look in particular at two spaces:

- The input space, where each data case corresponds to a vector. A classifier corresponds to a decision boundary, or a hyperplane such that the positive examples lie on one side, and negative examples lie on the other side.

- Weight space, where each set of classification weights corresponds to a vector. Each training case corresponds to a constraint in this space, where some regions of weight space are "good" (classify it correctly) and some regions are "bad" (classify it incorrectly).

The idea of weight space may seem pretty abstract, but it is very important that you become comfortable with it, since it underlies nearly everything we do in the course.

Using our understanding of input space and weight space, the limitations of linear classifiers will become immediately apparent. We'll see some examples of datasets which are not linearly separable (i.e. no linear classifier can correctly classify all the training cases), but which become linearly separable if we use a basis function representation.

1.1 Learning goals

- Know what is meant by binary linear classification.
- Understand why an explicit threshold for a classifier is redundant. Understand how we can get rid of the bias term by adding a "dummy" feature.
- Be able to specify weights and biases by hand to represent simple functions (*e.g.* AND, OR, NOT).
- Be familiar with input space and weight space.
  - Be able to plot training cases and classification weights in both input space and weight space.
- Be aware of the limitations of linear classifiers.
  - Know what is meant by convexity, and be able to use convexity to show that a given set of training cases is not linearly separable.
  - Understand how we can sometimes still separate the classes using a basis function representation.

2 Binary linear classifiers

We'll be looking at classifiers which are both **binary** (they distinguish between two categories) and **linear** (the classification is done using a linear function of the inputs). As in our discussion of linear regression, we assume each input is given in terms of D scalar values, called **input dimensions** or **features**, which we think summarize the important information for classification. (Some of the features, e.g. presence or absence of a symptom, may in fact be binary valued, but we're going to treat these as real-valued anyway.) The jth feature for the ith training example is denoted $x_j^{(i)}$. All of the features for a given training case are concatenated together to form a vector, which we'll denote $\mathbf{x}^{(i)}$. (Recall that vectors and matrices are shown in boldface.)

Associated with each data case is a binary-valued **target**, the thing we're trying to predict. By definition, a binary target takes two possible values,

which we'll call **classes**, and which are typically referred to as **positive** and **negative**. (E.g., the positive class might be "has disease" and the negative class might be "does not have disease.") Data cases belonging to these classes are called **positive examples** and **negative examples**, respectively. The **training set** consists of a set of N pairs $(\mathbf{x}^{(i)}, t^{(i)})$, where $\mathbf{x}^{(i)}$ is the input and $t^{(i)}$ is the binary-valued target, or **label**. Since the training cases come with labels, they're referred to as **labeled examples**. Confusingly, even though we talk about positive and negative examples, the $t^{(i)}$ typically take values in $\{0, 1\}$, where 0 corresponds to the "negative" class. Sorry, you'll just have to live with this terminology.

Our goal is to correctly classify all the training cases (and, hopefully, examples not in the training set). In order to do the classification, we need to specify a **model**, which determines how the predictions are computed from the inputs. As we said before, our model for this week is binary linear classifiers.

The way binary linear classifiers work is simple: they compute a linear function of the inputs, and determine whether or not the value is larger than some **threshold** r. Recall from Lecture 2 that a linear function of the input can be written as

$$w_1 x_1 + \cdots + w_D x_D + b = \mathbf{w}^T \mathbf{x} + b,$$

where $\mathbf{w}$ is a **weight vector** and b is a scalar-valued **bias**. Therefore, the prediction y can be computed as follows:

$$\begin{aligned} z &= \mathbf{w}^T \mathbf{x} + b \\ y &= \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases} \end{aligned}$$

This is the model we'll use for the rest of the week.

2.1 Thresholds and biases

Dealing with thresholds is rather inconvenient, but fortunately we can get rid of them entirely. In particular, observe that

$$\mathbf{w}^T \mathbf{x} + b \geq r \iff \mathbf{w}^T \mathbf{x} + b - r \geq 0.$$

In other words, we can obtain an equivalent model by replacing the bias with $b - r$ and setting r to 0. From now on, we'll assume (without loss of generality) that the threshold is 0. Therefore, we rewrite the model as follows:

$$\begin{aligned} z &= \mathbf{w}^T \mathbf{x} + b \\ y &= \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \end{aligned}$$

In fact, it's possible to eliminate the bias as well. We simply add another input dimension x_0, called a **dummy feature**, which always takes the value 1. Then

$$w_0 x_0 + w_1 x_1 + \cdots + w_D x_D = w_0 + w_1 x_1 + \cdots + w_D x_D,$$

so w_0 effectively plays the role of a bias. We can then simply write

$$z = \mathbf{w}^T \mathbf{x}.$$

Eliminating the bias often simplifies the statements of algorithms, so we'll sometimes use it for notational convenience. However, you should be aware that, when actually implementing the algorithms, the standard practice is to include the bias parameter explicitly.

2.2 Some examples

Let's look at some examples of how to represent simple functions using linear classifiers — specifically, AND, OR, and NOT.

Example 1. *Let's start with NOT, since it only involves a single input. Here's a "training set" of inputs and targets we're trying to match:*

x_1	t
0	*1*
1	*0*

Each of the training cases provides a constraint on the weights and biases. Let's start with the first training case. If $x_1 = 0$, then $t = 1$, so we need $z = w_1 x_1 + b = b \geq 0$. Technically we could satisfy this constraint with $b = 0$, but it's good practice to avoid solutions where z lies on the decision boundary. Therefore, let's tentatively set $b = 1$.

Now let's consider the second training case. The input is $x_1 = 1$ and the target is $t = 0$, so we need $z = w_1 \cdot 1 + b = w_1 + 1 < 0$. We can satisfy this inequality with $w_1 = -2$. This gives us our solution: $w_1 = -2$, $b = 1$.

Example 2. *Now let's consider AND. This is slightly more complicated, since we have 2 inputs and 4 training cases. The training cases are as follows:*

x_1	x_2	t
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Just like in the previous example, we can start by writing out the inequalities corresponding to each training case. We get:

$$\begin{aligned} b &< 0 \\ w_2 + b &< 0 \\ w_1 + b &< 0 \\ w_1 + w_2 + b &> 0 \end{aligned}$$

From these inequalities, we immediately see that $b < 0$ and $w_1, w_2 > 0$. The simplest way forward at this point is probably trial and error. Since the problem is symmetric with respect to w_1 and w_2, we might as well decide that $w_1 = w_2$. So let's try $b = -1, w_1 = w_2 = 1$ and see if it works. The first and fourth inequalities are clearly satisfied, but the second and third are not, since $w_1 + b = w_2 + b = 0$. So let's try making the bias a bit more negative. When we try $b = -1.5, w_1 = w_2 = 1$, we see that all four inequalities are satisfied, so we have our solution.

Following these examples, you should attempt the OR function on your own.

3 The geometric picture

Now let's move on to the main concepts of this lecture: data space and weight space. These are the spaces that the inputs and the weight vectors live in, respectively. It's very important to become comfortable thinking about these spaces, since we're going to treat the inputs and weights as vectors for the rest of the term.

In this lecture, we're going to focus on two-dimensional input and weight spaces. But keep in mind that this is a vast oversimplification: in practical settings, these spaces are typically many thousands, or even millions, of dimensions. It's pretty much impossible to visualize spaces this high-dimensional.

3.1 Data space

The first space to be familiar with is **data space**, or **input space**. Each point in this space corresponds to a possible input vector. (We're going to abuse mathematical terminology a bit by using "point" and "vector" interchangeably.) It's customary to represent positive and negative examples with the symbols "+" and "−", respectively.

Once we've chosen the weights $\mathbf{w}$ and bias b, we can divide the data space into a region where the points are classified as positive (the **positive region**), and a region where the points are classified as negative (the **negative region**). The boundary between these regions, i.e. the set where $\mathbf{w}^T\mathbf{x} + b = 0$, is called the **decision boundary**. Think back to your linear algebra class, and recall that the set determined by this equation is a **hyperplane**. The set of points on one side of the hyperplane is called a **half-space**. Examples are shown in Figure 1.

When we plot examples in two dimensions, the hyperplanes are actually lines. But you shouldn't think of them as lines — you should think of them as hyperplanes.

If it's possible to choose a linear decision boundary that correctly classifies all of the training cases, the training set is said to be **linearly separable**. As we'll see later, not all training sets are linearly separable.

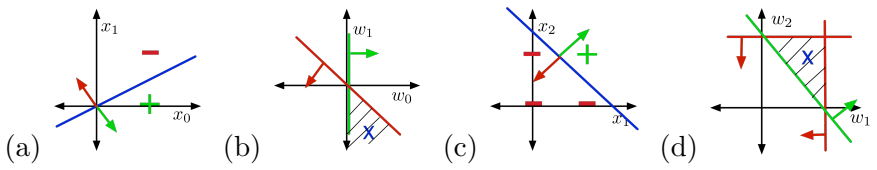


Figure 1: **(a)** Training examples and for NOT function, in data space. **(b)** NOT, in weight space. **(c)** Slice of data space for AND function corresponding to $x_0 = 1$. **(d)** Slice of weight space for AND function corresponding to $w_0 = -1$.

3.2 Weight space

As you'd expect from the name, weight vectors are also vectors, and the space they live in is called **weight space**. In this section, we'll assume there is no bias parameter unless stated otherwise. (See Section 2.1.) Each point in weight space is a possible weight vector.

Consider a positive training case $(\mathbf{x}, 1)$. The set of weight vectors which correctly classify this training case is given by the linear inequality $\mathbf{w}^T\mathbf{x} \geq 0$. (In fact, it's exactly the sort of inequality we derived in Examples 1 and 2.) Geometrically, the set of points satisfying this inequality is a half-space. For lack of a better term, we'll refer to the side which satisfies the constraint as the **good region**, and the other side as the **bad region**. Similarly, the set of weight vectors which correctly classify a negative training case $(\mathbf{x}, 0)$ is given by $\mathbf{w}^T\mathbf{x} < 0$; this is also a half-space. Examples are shown in Figure 1.

We're going to completely ignore the fact that one of these inequalities is strict and the other is not. The question of what happens on the decision boundaries isn't very interesting.

The set of weight vectors which correctly classify *all* of the training examples is the intersection of all the half-spaces corresponding to the individual examples. This set is called the **feasible region**. If the feasible region is nonempty, the problem is said to be **feasible**; otherwise it's said to be **infeasible**.

When we draw the constraints in two dimensions, we typically draw the line corresponding to the boundary of the constraint set, and then indicate the good region with an arrow. As with our data space visualizations, you should think of the boundary as a hyperplane, not as a line.

There's one constraint per training example. What happened to the fourth constraint in Figure 1(d)?

We can visualize three-dimensional examples by looking at slices. As shown in Figure 2, these slices will resemble our previous visualizations, except that the decision boundaries and constraints need not pass through the origin.

4 The perceptron learning rule

The perceptron is a kind of binary linear classifier. Recall from last lecture that this means it makes predictions by computing $\mathbf{w}^T\mathbf{x}+b$ and seeing if the result is positive or negative. Here, $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, and b is a scalar-valued bias. Recall as well that we can eliminate the bias by adding a dummy dimension to $\mathbf{x}$. For the perceptron algorithm, it will be convenient to represent the positive and negative classes with 1 and -1, instead of 1 and 0 as we use in the rest of the course. Therefore,

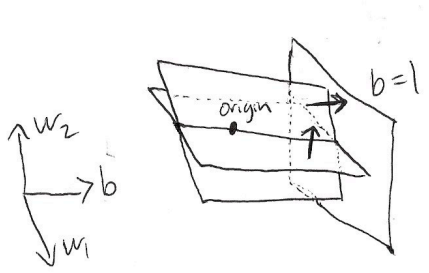


Figure 2: Visualizing a slice of a 3-dimensional weight space.

the classification model is as follows:

$$z = \mathbf{w}^T \mathbf{x} \tag{1}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{2}$$

Here's a rough sketch of the **perceptron algorithm**. We examine each of the training cases one at a time. For each input $\mathbf{x}^{(i)}$, we compute the prediction $y^{(i)}$ and see if it matches the target $t^{(i)}$. If the prediction is correct, we do nothing. If it is wrong, we adjust the weights in a direction that makes it more correct.

Now for the details. First of all, how do we determine if the prediction is correct? We could simply check if $y^{(i)} = t^{(i)}$, but this has a slight problem: if $\mathbf{x}^{(i)}$ lies exactly on the classification boundary, it is technically classified as positive according to the above definition. But we don't want our training cases to lie on the decision boundary, since this means the classification may change if the input is perturbed even slightly. We'd like our classifiers to be more robust than this. Instead, we'll use the stricter criterion

$$z^{(i)} t^{(i)} > 0. \tag{3}$$

You should now check that this criterion correctly handles the various cases that may occur.

The other question is, how do we adjust the weight vector? If the training case is positive and we classify it as negative, we'd like to increase the value of z. In other words, we'd like

$$z' = \mathbf{w}'^T \mathbf{x} > \mathbf{w}^T \mathbf{x} = z, \tag{4}$$

where $\mathbf{w}'$ and $\mathbf{w}$ are the new and old weight vectors, respectively. The perceptron algorithm achieves this using the update

$$\mathbf{w}' = \mathbf{w} + \alpha \mathbf{x}, \tag{5}$$

where $\alpha > 0$. We now check that (4) is satisfied:

$$\mathbf{w}'^T \mathbf{x} = (\mathbf{w} + \alpha \mathbf{x})^T \mathbf{x} \tag{6}$$

$$= \mathbf{w}^T \mathbf{x} + \alpha \mathbf{x}^T \mathbf{x} \tag{7}$$

$$= \mathbf{w}^T \mathbf{x} + \alpha \|\mathbf{x}\|^2. \tag{8}$$

Here, $\|\mathbf{x}\|$ represents the Euclidean norm of $\mathbf{x}$. Since the squared norm is always positive, we have $z' > z$.

Conversely, if it's a negative example which we mistakenly classified as positive, we want to decrease z, so we use a negative value of α. Since it's possible to show that the absolute value of α doesn't matter, we generally use $\alpha = 1$ for positive cases and $\alpha = -1$ for negative cases. We can denote this compactly with

$$\mathbf{w} \leftarrow \mathbf{w} + t\mathbf{x}. \tag{9}$$

This rule is known as the **perceptron learning rule**.

Now we write out the perceptron algorithm in full:

For each training case $(\mathbf{x}^{(i)}, t^{(i)})$,

$z^{(i)} \leftarrow \mathbf{w}^T \mathbf{x}^{(i)}$
If $z^{(i)} t^{(i)} \leq 0$,
    $\mathbf{w} \leftarrow \mathbf{w} + t^{(i)} \mathbf{x}^{(i)}$

In thinking about this algorithm, remember that we're denoting the classes with -1 and 1 (rather than 0 and 1, as we do in the rest of the course).

5 The limits of linear classifiers

Linear classifiers can represent a lot of things, but they can't represent everything. The classic example of what they can't represent is the XOR function. It should be pretty obvious from inspection that you can't draw a line separating the two classes. But how do we actually prove this?

5.1 Convex sets

An important geometric concept which helps us out here is **convexity**. A set S is convex if the line segment connecting any two points in S must lie within S. It's not too hard to show that if S is convex, then any **weighted average** of points in S must also lie within S. A weighted average of points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ is a point given by the linear combination

$$\mathbf{x}^{(avg)} = \lambda_1 \mathbf{x}^{(1)} + \cdots + \lambda_N \mathbf{x}^{(N)},$$

where $0 \leq \lambda_i \leq 1$ and $\lambda_1 + \cdots + \lambda_N = 1$. You can think of the weighted average as the center of mass, where the mass of each point is given by λ_i.

In the context of binary classification, there are two important sets that are always convex:

1. In data space, the positive and negative regions are both convex. Both regions are half-spaces, and it should be visually obvious that half-spaces are convex. This implies that if inputs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ are all in the positive region, then any weighted average must also be in the positive region. Similarly for the negative region.

2. In weight space, the feasible region is convex. The rough mathematical argument is as follows. Each good region (the set of weights which correctly classify one data point) is convex because it's a half-space. The feasible region is the intersection of all the good regions, so it must be convex because the intersection of convex sets is convex.

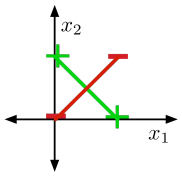


Figure 3: The XOR function is not linearly separable.

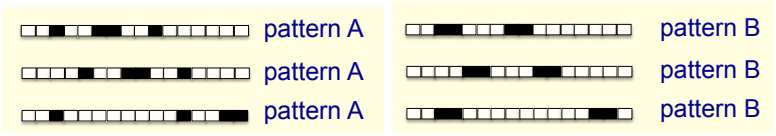


Figure 4: No linear hypothesis can separate these two patterns in all possible translations (with wrap-around).

5.2 Showing that functions aren't linearly separable

Now let's see how convexity can be used to show functions aren't linearly separable.

Example 3. *Let's return to the XOR example. Since the positive region is convex, if we draw the line segment connecting the two positive examples* $(0, 1)$ *and* $(1, 0)$*, this entire line segment must be classified as positive. Similarly, if we draw the line segment connecting the two negative examples* $(0, 0)$ *and* $(1, 1)$*, the entire line segment must be classified as negative. But these two line segments intersect at* $(0.5, 0.5)$*, which means this point must be classified as both positive and negative, which is impossible. (See Figure 3.) Therefore, XOR isn't linearly separable.*

Example 4. *Our last example was somewhat artificial. Let's now turn to a somewhat more troubling, and practically relevant, limitation of linear classifiers. Let's say we want to give a robot a vision system which can recognize objects in the world. Since the robot could be looking any given direction, it needs to be able to recognize objects regardless of their location in its visual field. I.e., it should be able to recognize a pattern in any possible translation.*

As a simplification of this situation, let's say our inputs are 16-dimensional binary vectors and we want to distinguish two patterns, A, and B (shown in Figure 4), which can be placed in any possible translation, with wrap-around. (I.e., if you shift the pattern right, then whatever falls off the right side reappears on the left.) Thus, there are 16 examples of A and 16 examples of B that our classifier needs to distinguish.

By convexity, if our classifier is to correctly classify all 16 instances of A, then it must also classify the average of all 16

instances as A. Since 4 out of the 16 values are on, the average of all instances is simply the vectors $(0.25, 0.25, \ldots, 0.25)$. *Similarly, for it to correctly classify all 16 instances of B, it must also classify their average as B. But the average is also* $(0.25, 0.25, \ldots, 0.25)$. *Since this vector can't possibly be classified as both A and B, this dataset must not be linearly separable.*

More generally, we can't expect any linear classifier to detect a pattern in all possible translations. This is a serious limitation of linear classifiers as a basis for a vision system.

5.3 Circumventing this problem by using feature representations

We just saw a negative result about linear classifiers. Let's end on a more positive note. In Lecture 2, we saw how linear regression could be made more powerful using a basis function, or feature, representation. The same trick applies to classification. Essentially, in place of $z = \mathbf{w}^T\mathbf{x} + b$, we use $z = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) + b$, where $\boldsymbol{\phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \ldots, \phi_D(\mathbf{x}))$ is a function mapping input vectors to feature vectors. Let's see how we can represent XOR using carefully selected features.

Example 5. *Consider the following feature representation for XOR:*

$$\begin{aligned}\phi_1(\mathbf{x}) &= x_1\\ \phi_2(\mathbf{x}) &= x_2\\ \phi_3(\mathbf{x}) &= x_1x_2\end{aligned}$$

In this representation, our training set becomes

$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$	$\phi_3(\mathbf{x})$	t
0	*0*	*0*	*0*
0	*1*	*0*	*1*
1	*0*	*0*	*1*
1	*1*	*1*	*0*

Using the same techniques as in Examples 1 and 2, we find that the following set of weights and biases correctly classifies all the training examples:

$$b = -0.5 \qquad w_1 = 1 \qquad w_2 = 1 \qquad w_3 = -2.$$

The only problem is, where do we get the features from? In this example, we just pulled them out of a hat. Unfortunately, there's no recipe for coming up with good features, which is part of what makes machine learning hard. But next week, we'll see how we can *learn* a set of features by training a multilayer neural net.

Lecture 2, Part 3: Training a Classifier

Roger Grosse

1 Introduction

Now that we've defined what binary classification is, let's actually train a classifier. We'll approach this problem in much the same way as we did linear regression: define a model and a cost function, and minimize the cost using gradient descent. The one thing that makes the classification case harder is that it's not obvious what loss function to use. We can't just use the classification error itself, because the gradient is zero almost everywhere! Instead, we'll define a *surrogate loss function*, i.e. an alternative loss function which is easier to optimize.

1.1 Learning Goals

- Understand why classification error and squared error are problematic cost functions for classification.
- Know what cross-entropy is and understand why it can be easier to optimize than squared error (assuming a logistic activation function).
- Be able to derive the gradient descent updates for all of the models and cost functions mentioned in this lecture and to implement the learning algorithms in Python.
- Know what hinge loss is, and how it relates to cross-entropy loss.
- Understand how binary logistic regression can be generalized to multiple variables.
- Know what it means for a function to be convex, how to check convexity visually, and why it's important for optimization.
  - Algebraically proving functions to be convex is beyond the scope of this course.
- Know how to check the correctness of gradients using finite differences.

2 Choosing a cost function

Recall the setup from the previous lecture. We have a set of training examples $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^N$, where the $\mathbf{x}^{(i)}$ are vector-valued inputs, and $t^{(i)}$ are binary-valued targets. We would like to learn a binary linear classifier, where we compute a linear function of $\mathbf{x}^{(i)}$ and threshold it at zero:

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

In the last lecture, our goal was to correctly classify every training example. But this might be impossible if the dataset isn't linearly separable. Even if it's possible to correctly classify every training example, it may be undesirable since then we might just overfit!

How can we define a sensible learning criterion when the dataset isn't linearly separable? One natural criterion is to minimize the number of misclassified training examples. We can formalize this with the **classification error** loss, or the **0-1 loss**:

$$\mathcal{L}_{0-1}(y,t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

As always, the cost function is just the loss averaged over the training examples; in this case, that corresponds to the **error rate**, or fraction of misclassified examples. How do we make this small?

2.1 Attempt 1: 0-1 loss

As a first attempt, let's try to minimize 0-1 loss directly. In order to compute the gradient descent updates, we need to compute the partial derivatives $\partial\mathcal{L}_{0-1}/\partial w_j$. Rather than mechanically deriving this derivative, let's think about what it means. It means, how much does $\mathcal{L}_{0-1}$ change if you make a very small change to w_j? As long as we're not on the classification boundary, making a small enough change to w_j will have *no effect* on $\mathcal{L}_{0-1}$, because the prediction won't change. This implies that $\partial\mathcal{L}_{0-1}/\partial w_j = 0$, as long as we're not on the boundary. Gradient descent will go nowhere. (If we are on the boundary, the cost is discontinuous, which certainly isn't any better!) OK, we certainly can't optimize 0-1 loss with gradient descent.

Near the end of the course, when we discuss reinforcement learning, we'll see an algorithm which can minimize 0-1 loss directly. It's nowhere near as efficient as gradient descent, though, so we still need the techniques of this lecture!

2.2 Attempt 2: linear regression

Since that didn't work, let's try using something we already know: linear regression. Recall that this assumes a linear model and the squared error loss function:

$$y = \mathbf{w}^\top \mathbf{x} + b \tag{3}$$

$$\mathcal{L}_{\text{SE}}(y,t) = \frac{1}{2}(y-t)^2 \tag{4}$$

We've already seen two ways of optimizing this: gradient descent, and a closed-form solution. But does it make sense for classification? One obvious problem is that the predictions are real-valued rather than binary. But that's OK, since we can just pick some scheme for binarizing them, such as thresholding at $y = 1/2$. When we replace a loss function we trust with another one we trust less but which is easier to optimize, the replacement one is called a **surrogate loss function**.

But there's still a problem. Suppose we have a positive example, i.e. $t = 1$. If we predict $y = 1$, we get a cost of 0, whereas if we make the wrong prediction $y = 0$, we get a cost of $1/2$; so far, so good. But suppose we're really confident that this is a positive example, and predict $y = 9$. Then we pay a cost of $\frac{1}{2}(9-1)^2 = 32$. This is far higher than the cost for $y = 0$, so the learning algorithm will try very hard to prevent this from happening.

That's not bad in itself, but it means that something else might need to be sacrificed, if it's impossible to match all of the targets exactly. Perhaps the sacrifice will be that it incorrectly classifies some other training examples.

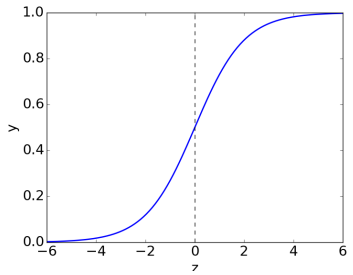
2.3 Attempt 3: logistic nonlinearity

The problem with linear regression is that the predictions were allowed to take arbitrary real values. But it makes no sense to predict anything smaller than 0 or larger than 1. Let's fix this problem by applying a **nonlinearity**, or **activation function**, which squashes the predictions to be between 0 and 1. In particular, we'll use something called the **logistic function**:

If you predict $y > 1$, then regardless of the target, you can decrease the loss by setting $y = 1$. Similarly for $y < 0$.

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{5}$$

This is a kind of **sigmoidal**, or S-shaped, function:



What's important about this function is that it increases monotonically, with asymptotes at 0 and 1. (Plus, it's smooth, so we can compute derivatives.)

Another advantage of the logistic function is that calculations tend to work out very nicely.

We refine the model as follows:

$$z = \mathbf{w}^\top \mathbf{x} + b \tag{6}$$

$$y = \sigma(z) \tag{7}$$

$$\mathcal{L}_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2. \tag{8}$$

Notice that this model solves the problem we observed with linear regression. As the predictions get more and more confident on the correct answer, the loss continues to decrease.

To derive the gradient descent updates, we'll need the partial derivatives of the cost function. We'll do this by applying the Chain Rule twice: first to compute $\mathrm{d}\mathcal{L}_{\text{SE}}/\mathrm{d}z$, and then again to compute $\partial\mathcal{L}_{\text{SE}}/\partial w_j$. But first, let's note the convenient fact that

This is equivalent to the elegant identity $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

$$\begin{aligned}\frac{\partial y}{\partial z} &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= y(1 - y).\end{aligned} \tag{9}$$

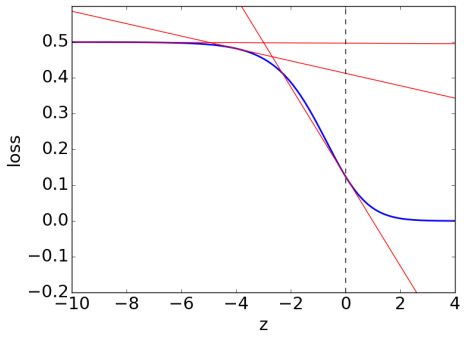


Figure 1: Visualization of derivatives of squared error loss with logistic nonlinearity, for a training example with $t = 1$. The derivative $\mathrm{d}\mathcal{J}/\mathrm{d}z$ corresponds to the slope of the tangent line.

Now for the Chain Rule:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} \\ &= (y-t)y(1-y) \qquad (10) \\ \frac{\partial\mathcal{L}_{\mathrm{SE}}}{\partial w_j} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\frac{\partial z}{\partial w_j} \\ &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\cdot x_j. \qquad (11)\end{aligned}$$

Done! Why don't we go one step further and plug Eqn. 10 into Eqn. 11? The reason is that our goal isn't to compute a *formula* for $\partial\mathcal{L}_{\mathrm{SE}}/\partial w_j$; as computer scientists, our goal is to come up with a *procedure* for computing it. The two formulas above give us a procedure which we can implement directly in Python. One advantage of doing it this way is that we can reuse some of the work we've done in computing the derivative with respect to the bias:

At this point, you should stop and sanity check the equations we just derived, e.g. checking that they have the sign that they ought to. Get in the habit of doing this.

Reusing computation of derivatives is one of the main insights behind backpropagation, one of the central algorithms in this course.

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}b} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\frac{\partial z}{\partial b} \\ &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z} \qquad (12)\end{aligned}$$

If we had expanded out the entire formula, it might not be obvious to us that we can reuse computation like this.

So far, so good. But there's one hitch. Let's suppose you classify one of the training examples extremely wrong, e.g. you confidently predict a negative label with $z = -5$, which gives $y \approx 0.0067$, for a positive example (i.e. $t = 1$). Plugging these values into Eqn 10, we find that $\partial\mathcal{L}_{\mathrm{SE}}/\partial z \approx -0.0066$. This is a pretty small value, considering how big the mistake was. As shown in Figure 1, the more confident the wrong prediction, the *smaller* the gradient is! The most badly misclassified examples will have hardly any effect on the training. This doesn't seem very good. We say the learning algorithm does not have a strong **gradient signal** for those training examples.

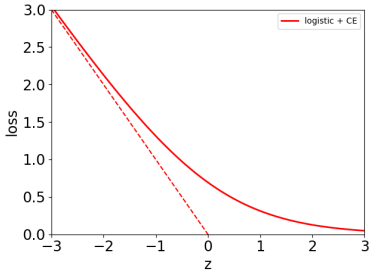


Figure 2: Plot of cross-entropy loss as a function of the input z to the logistic activation function.

The problem with squared error loss in the classification setting is that it doesn't distinguish bad predictions from extremely bad predictions. If $t = 1$, then a prediction of $y = 0.01$ has roughly the same squared-error loss as a prediction of $y = 0.00001$, even though in some sense the latter is more wrong. This isn't necessarily a problem in terms of the cost function itself: whether 0.00001 is inherently much worse than 0.01 depends on the situation. (If all we care about is classification error, they're essentially equivalent.) But *from the perspective of optimization*, the fact that the losses are nearly equivalent is a big problem. If we can increase y from 0.00001 to 0.0001, that means we're "getting warmer," but this doesn't show up in the squared-error loss. We'd like a loss function which reflects our intuitive notion of getting warmer.

Think about how the argument in this paragraph relates to the one in the previous paragraph.

Actually, the effect discussed here can also be beneficial, because it makes the algorithm *robust*, in that it can learn to ignore mislabeled examples. Cost functions like this are sometimes used for this reason. However, when you do use it, you should be aware of the optimization difficulties it creates!

2.4 Final touch: cross-entropy loss

The problem with squared-error loss is that it treats $y = 0.01$ and $y = 0.00001$ as nearly equivalent (for a positive example). We'd like a loss function which makes these very different. One such loss function is **cross-entropy (CE)**. This is defined as follows:

You'll sometimes see cross-entropy abbreviated XE.

$$\mathcal{L}_{\text{CE}}(y, t) = \begin{cases} -\log y & \text{if } t = 1 \\ -\log 1 - y & \text{if } t = 0 \end{cases} \tag{13}$$

In our earlier example, we see that $\mathcal{L}_{\text{CE}}(0.01, 1) = 4.6$, whereas $\mathcal{L}_{\text{CE}}(0.00001, 1) = 11.5$, so cross-entropy treats the latter as much worse than the former.

When we do calculations, it's cumbersome to use the case notation, so we instead rewrite Eqn. 13 in the following form. You should check that they are equivalent:

$$\mathcal{L}_{\text{CE}}(y, t) = -t \log y - (1 - t) \log 1 - y. \tag{14}$$

See if you can derive the equations for the asymptote lines.

Remember, the logistic function squashes y to be between 0 and 1, but cross-entropy draws big distinctions between probabilities close to 0 or 1. Interestingly, these effects cancel out: Figure 2 plots the loss as a function of z. You get a sizable gradient signal even when the predictions are very wrong.

When we combine the logistic activation function with cross-entropy loss, you get **logistic regression**:

$$\begin{aligned} z &= \mathbf{w}^\top \mathbf{x} + b \\ y &= \sigma(z) \\ \mathcal{L}_{\text{CE}} &= -t \log y - (1-t) \log 1 - y. \end{aligned} \tag{15}$$

Now let's compute the derivatives. We'll do it two different ways: the mechanical way, and the clever way. Let's do the mechanical way first, as an example of the chain rule for derivatives. Remember, our job here isn't to produce a formula for the derivatives, the way we would in calculus class. Our job is to give a procedure for computing the derivatives which we could translate into NumPy code. The following does that:

The second step of this derivation uses Eqn. 9.

$$\begin{aligned} \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y} &= -\frac{t}{y} + \frac{1-t}{1-y} \\ \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z} &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} \\ &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y} \cdot y(1-y) \\ \frac{\partial \mathcal{L}_{\text{CE}}}{\partial w_j} &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z}\frac{\partial z}{\partial w_j} \\ &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z} \cdot x_j \end{aligned} \tag{16}$$

This can be translated directly into NumPy (exercise: how do you vectorize this?). If we were good little computer scientists, we would stop here. But today we're going to be naughty computer scientists and break the abstraction barrier between the activation function (logistic) and the cost function (cross-entropy).

2.5 Logistic-cross-entropy function

There's a big problem with Eqns. 15 and 16: what happens if we have a positive example ($t = 1$), but we confidently classify it as a negative example ($z \ll 0$, implying $y \approx 0$)? This is likely to happen at the very beginning of training, so we should be able to handle it. But if y is small enough, it could be smaller than the smallest floating point value, i.e. **numerically zero**. Then when we compute the cross-entropy, we take the log of 0 and get $-\infty$. Or if this doesn't happen, think about Eqn. 16. Since y appears in the denominator, $\mathrm{d}\mathcal{L}_{\text{CE}}/\mathrm{d}y$ will be extremely large in magnitude, which again can cause numerical difficulties. These bugs are very subtle, and can be hard to track down if you don't expect them.

What we do instead is combine the logistic function and cross-entropy loss into a single function, which we term **logistic-cross-entropy**:

$$\mathcal{L}_{\text{LCE}}(z,t) = \mathcal{L}_{\text{CE}}(\sigma(z),t) = t \log(1+e^{-z}) + (1-t)\log(1+e^{z}) \tag{17}$$

This still isn't numerically stable if we implement it directly, since e^z could blow up. But most scientific computing environments provide a numerically

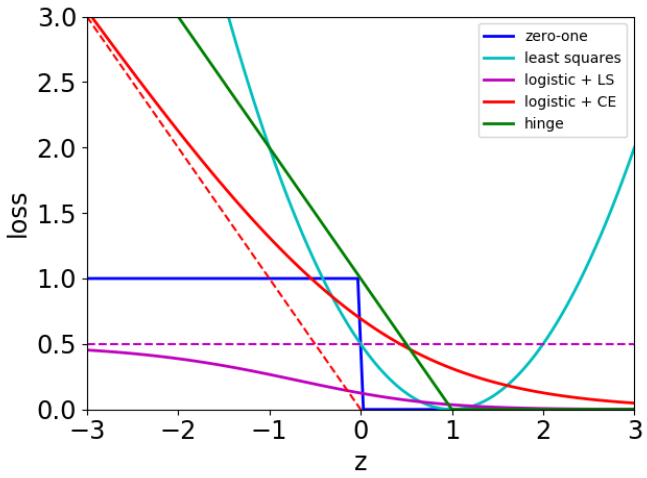


Figure 3: Comparison of the loss functions considered so far.

stable **log-sum-exp** routine.[1] In numpy, this is `np.logaddexp`. So the following code would compute the logistic-cross-entropy:

```
E = t * np.logaddexp(0, -z) + (1-t) * np.logaddexp(0, z)
```

Now to compute the loss derivatives:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{LCE}}}{\mathrm{d}z} &= \frac{\mathrm{d}}{\mathrm{d}z}\left[t\log(1+e^{-z}) + (1-t)\log(1+e^{z})\right] \\ &= -t\cdot\frac{e^{-z}}{1+e^{-z}} + (1-t)\frac{e^{z}}{1+e^{z}} \\ &= -t(1-y) + (1-t)y \\ &= y - t\end{aligned} \tag{18}$$

This is like magic! We took a somewhat complicated formula for the logistic activation function, combined it with a somewhat complicated formula for the cross-entropy loss, and wound up with a stunningly simple formula for the loss derivative! Observe that this is exactly the same formula as for $\mathrm{d}\mathcal{L}_{\mathrm{SE}}/\mathrm{d}y$ in the case of linear regression. And it has the same intuitive interpretation: if $y > t$, you made too positive a prediction, so you want to shift your prediction in the negative direction. Conversely, if $y < t$, you want to shift your prediction in the positive direction.

This isn't a coincidence. The reason it happens is beyond the scope of this course, but if you're curious, look up "generalized linear models."

2.6 Another alternative: hinge loss

Another loss function you might encounter is **hinge loss**. Here, y is a real value, and $t \in \{-1, 1\}$.

$$\mathcal{L}_{\mathrm{H}}(y, t) = \max(0, 1 - ty)$$

Hinge loss is plotted in Figure 3 for a positive example. One useful property of hinge loss is that it's an upper bound on 0–1 loss; this is a useful property

[1] The log-sum-exp trick is pretty neat. https://hips.seas.harvard.edu/blog/2013/01/09/computing-log-sum-exp/

for a surrogate loss function, since it means that if you make the hinge loss small, you've also made 0–1 loss small. A linear model with hinge loss is known as a **support vector machine (SVM)**:

$$y = \mathbf{w}^\top \mathbf{x} + b \tag{19}$$

$$\mathcal{L}_{\text{H}} = \max(0, 1 - ty) \tag{20}$$

If you take CSC411, you'll learn a lot about SVMs, including their statistical motivation, how to optimize them efficiently and how to make them nonlinear (using something called the "kernel trick"). But you already know one optimization method: you already know enough to derive the gradient descent updates for an SVM.

Interestingly, even though SVMs came from a different community and had a different sort of motivation from logistic regression, the algorithms behave very similarly in practice. The reason has to do with the loss functions. Figure 3 compares hinge loss to cross-entropy loss; even though cross-entropy is smoother, the asymptotic behavior is the same, suggesting the loss functions are basically pretty similar.

All of the loss functions covered so far is shown in Figure 3. Take the time to review them, to understand their strengths and weaknesses.

3 Multiclass classification

So far we've talked about binary classification, but most classification problems involve more than two categories. Fortunately, this doesn't require any new ideas: everything pretty much works by analogy with the binary case. The first question is how to represent the targets. We could represent them as integers, but it's more convenient to use a **one-hot vector**, also called a **one-of-K encoding**:

$$\mathbf{t} = \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{\text{entry } k \text{ is } 1} \tag{21}$$

Now let's design the whole model by analogy with the binary case.

First of all, consider the linear part of the model. We have K outputs and D inputs. To represent a linear function, we'll need a $K \times D$ **weight matrix**, as well as a K-dimensional **bias vector**. We first compute the intermediate quantities as follows:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{22}$$

This is the general form of a linear function from $\mathbb{R}^D$ to $\mathbb{R}^K$.

Next, the activation function. We saw that the logistic function was a good thing to use in the binary case. There's a multivariate generalization of the logistic function called the **softmax function**:

Try plugging in $K = 2$ to figure out how the softmax relates to the logistic function.

$$y_k = \text{softmax}(z_1, \ldots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} \tag{23}$$

Importantly, the outputs of the softmax function are nonnegative and sum to 1, so they can be interpreted as a probability distribution over the K

classes (just like the output of the logistic could be interpreted as a probability). The inputs to the softmax are called the **logits**. Note that when one of the z_k's is much larger than the others, the output of the softmax will be approximately the argmax, in the one-hot encoding. Hence, a more accurate name might be "soft-argmax."

Think about the logits as the "log-odds", because when you exponentiate them you get the odds ratios of the probabilities.

Finally, the loss function. Cross-entropy can be generalized to the multiple-output case:

$$\begin{aligned}\mathcal{L}_{\text{CE}}(\mathbf{y}, \mathbf{t}) &= -\sum_{k=1}^{K} t_k \log y_k \\ &= -\mathbf{t}^\top (\log \mathbf{y}).\end{aligned}$$

You'll sometimes see $\sigma(\mathbf{z})$ used to denote the softmax function, by analogy with the logistic. But in this course, it will always denote the logistic function.

Here, $\log \mathbf{y}$ represents the elementwise log. Note that only one of the t_k's is 1 and the rest are 0, so the summation has the effect of picking the relevant entry of the vector $\log \mathbf{y}$. (See how convenient the one-hot notation is?) Note that this loss function only makes sense for predictions which sum to 1; if you eliminate that constraint, you could trivially minimize the loss by making all the y_k's large.

Try plugging in $K = 2$ to see how this relates to binary cross-entropy.

When we put these things together, we get **multiclass logistic regression**, or **softmax regression**:

$$\begin{aligned}\mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} &= \text{softmax}(\mathbf{z}) \\ \mathcal{L}_{\text{CE}} &= -\mathbf{t}^\top (\log \mathbf{y})\end{aligned}$$

We won't go through the derivatives in detail, but it basically works out exactly like logistic regression. The softmax and cross-entropy functions interact nicely with each other, so we always combine them into a single **softmax-cross-entropy** function $\mathcal{L}_{\text{SCE}}$ for purposes of numerical stability. The derivatives of $\mathcal{L}_{\text{SCE}}$ have the same elegant formula we've been seeing repeatedly, except this time remember that $\mathbf{t}$ and $\mathbf{y}$ are both vectors:

$$\frac{\partial \mathcal{L}_{\text{SCE}}}{\partial \mathbf{z}} = \mathbf{y} - \mathbf{t} \tag{24}$$

Softmax regression is an elegant learning algorithm which can work very well in practice.

4 Convex Functions

An important criterion we often use to compare different loss functions is convexity. Recall that a set $\mathcal{S}$ is convex if the line segment connecting any two points in $\mathcal{S}$ lies entirely within $\mathcal{S}$. Mathematically, this means that for $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$,

$$(1-\lambda)\mathbf{x}_0 + \lambda \mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \le \lambda \le 1.$$

The definition of a **convex function** is closely related. A function f is convex if for any $\mathbf{x}_0, \mathbf{x}_1$ in the domain of f,

$$f((1-\lambda)\mathbf{x}_0 + \lambda \mathbf{x}_1) \le (1-\lambda) f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1) \tag{25}$$

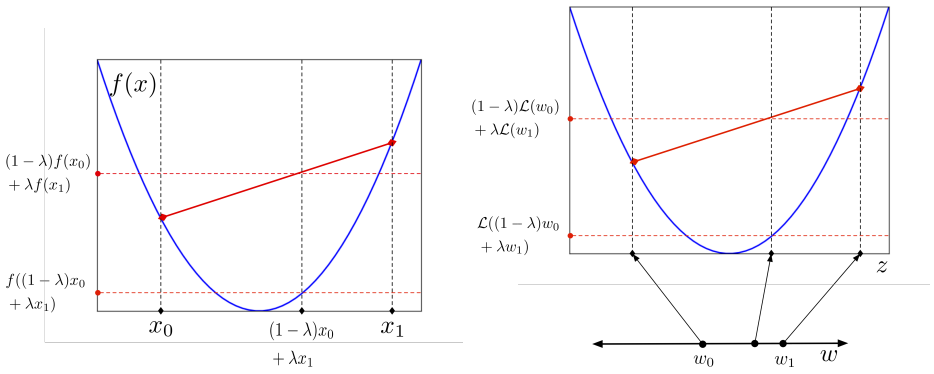


Figure 4: **Left:** Definition of convexity. **Right:** Proof-by-picture that if the model is linear and $\mathcal{L}$ is a convex function of $z = \mathbf{w}^\top \mathbf{x} + b$, then it's also convex as a function of $\mathbf{w}$ and b.

This is shown graphically in Figure 4. Another way to phrase this requirement is that the line segment connecting any two points on the graph of f must lie above the graph of f. Equivalently, the set of points lying above the graph of f must be a convex set. Intuitively, convex functions are bowl-shaped.

Convexity is a really important property from the standpoint of optimization. There are two main reasons for this:

1. All critical points are global minima, so if you can set the derivatives to zero, you've solved the problem.

2. Gradient descent will always converge to the global minimum.

We'll talk in more detail in a later lecture about what can go wrong when the cost function is not convex. Look back at our comparison of loss functions in Figure 3. You can see visually that squared error, hinge loss, and the logistic regression objective are all convex; 0–1 loss, and logistic-with-least-squares are not convex. It's not a coincidence that the loss functions we might actually try to optimize are the convex ones. There is an entire field of research on convex optimization, which comes up with better ways to minimize convex functions over convex sets, as well as ways to formulate various kinds of problems in terms of convex optimization.

Note that even though convexity is important, most of the optimization problems we'll consider in this course will be non-convex, because training a deep neural network is a non-convex problem, even when the loss function is convex. Nonetheless, convex loss functions somehow still tend to be advantageous from the standpoint of optimization.

5 Gradient Checking with Finite Differences

We've derived a lot of gradients so far. How do we know if they're correct? Fortunately, there's an easy and effective procedure for testing them. Recall

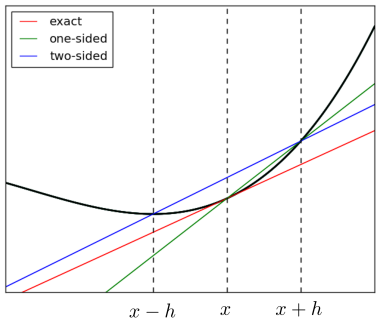


Figure 5: Estimating a derivative using one-sided and two-sided finite differences.

the definition of the partial derivative:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h} \tag{26}$$

We can check the derivatives numerically by plugging in a small value of h, such as 10^{-10}. This is known as the method of **finite differences**.

You don't want to implement your actual learning algorithm using finite differences, because it's very slow, but it's great for testing.

It's actually better to use the two-sided definition of the partial derivative than the one-sided one, since it is much more accurate:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h} \tag{27}$$

An example is shown in Figure 5 of estimating a derivative using the one-sided and two-sided formulas.

Gradient checking is really important! In machine learning, your algorithm can often seem to learn well even if the gradient calculation is totally wrong. This might lead you to skip the correctness checks. But it might work even better if the derivatives are correct, and this is important when you're trying to squeeze out the last bit of accuracy. Wrong derivatives can also lead you on wild goose chases, as you make changes to your system which appear to help significantly, but actually are only helping because they compensate for errors in the gradient calculations. If you implement derivatives by hand, gradient checking is the single most important thing you need to do to get your algorithm to work well.

Lecture 3: Multilayer Perceptrons

Roger Grosse

1 Introduction

So far, we've only talked about linear models: linear regression and linear binary classifiers. We noted that there are functions that can't be represented by linear models; for instance, linear regression can't represent quadratic functions, and linear classifiers can't represent XOR. We also saw one particular way around this issue: by defining features, or basis functions. E.g., linear regression can represent a cubic polynomial if we use the feature map $\boldsymbol{\psi}(x) = (1, x, x^2, x^3)$. We also observed that this isn't a very satisfying solution, for two reasons:

1. The features need to be specified in advance, and this can require a lot of engineering work.
2. It might require a very large number of features to represent a certain set of functions; e.g. the feature representation for cubic polynomials is cubic in the number of input features.

In this lecture, and for the rest of the course, we'll take a different approach. We'll represent complex nonlinear functions by connecting together lots of simple processing units into a *neural network*, each of which computes a linear function, possibly followed by a nonlinearity. In aggregate, these units can compute some surprisingly complex functions. By historical accident, these networks are called *multilayer perceptrons*.

Some people would claim that the methods covered in this course are really "just" adaptive basis function representations. I've never found this a very useful way of looking at things.

1.1 Learning Goals

- Know the basic terminology for neural nets
- Given the weights and biases for a neural net, be able to compute its output from its input
- Be able to hand-design the weights of a neural net to represent functions like XOR
- Understand how a hard threshold can be approximated with a soft threshold
- Understand why shallow neural nets are universal, and why this isn't necessarily very interesting

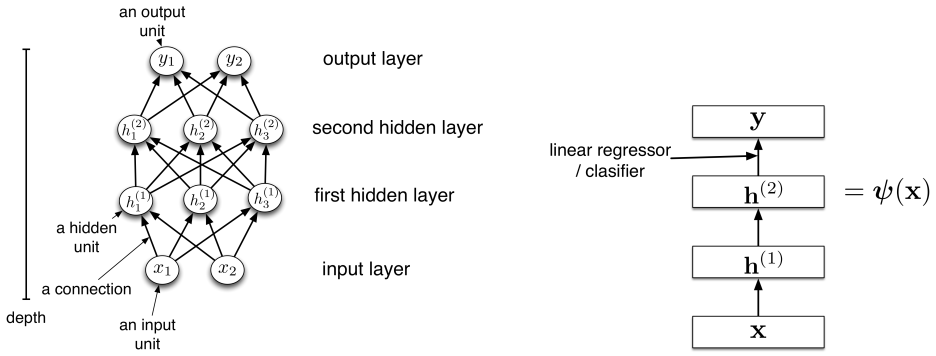


Figure 1: A multilayer perceptron with two hidden layers. **Left:** with the units written out explicitly. **Right:** representing layers as boxes.

2 Multilayer Perceptrons

In the first lecture, we introduced our general neuron-like processing unit:

$$a = \phi\left(\sum_j w_j x_j + b\right),$$

where the x_j are the inputs to the unit, the w_j are the weights, b is the bias, ϕ is the nonlinear activation function, and a is the unit's activation. We've seen a bunch of examples of such units:

- Linear regression uses a linear model, so $\phi(z) = z$.
- In binary linear classifiers, ϕ is a hard threshold at zero.
- In logistic regression, ϕ is the logistic function $\sigma(z) = 1/(1 + e^{-z})$.

A **neural network** is just a combination of lots of these units. Each one performs a very simple and stereotyped function, but in aggregate they can do some very useful computations. For now, we'll concern ourselves with **feed-forward neural networks**, where the units are arranged into a graph without any cycles, so that all the computation can be done sequentially. This is in contrast with **recurrent neural networks**, where the graph can have cycles, so the processing can feed into itself. These are much more complicated, and we'll cover them later in the course.

The simplest kind of feed-forward network is a **multilayer perceptron (MLP)**, as shown in Figure 1. Here, the units are arranged into a set of **layers**, and each layer contains some number of identical units. Every unit in one layer is connected to every unit in the next layer; we say that the network is **fully connected**. The first layer is the **input layer**, and its units take the values of the input features. The last layer is the **output layer**, and it has one unit for each value the network outputs (i.e. a single unit in the case of regression or binary classifiation, or K units in the case of K-class classification). All the layers in between these are known as **hidden layers**, because we don't know ahead of time what these units should compute, and this needs to be discovered during learning. The units

MLP is an unfortunate name. The *perceptron* was a particular algorithm for binary classification, invented in the 1950s. Most multilayer perceptrons have very little to do with the original perceptron algorithm.

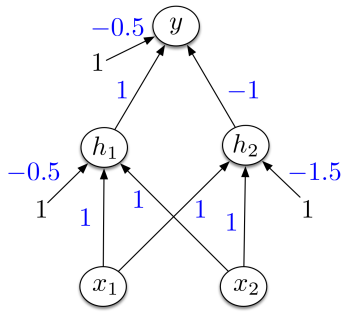


Figure 2: An MLP that computes the XOR function. All activation functions are binary thresholds at 0.

in these layers are known as **input units**, **output units**, and **hidden units**, respectively. The number of layers is known as the **depth**, and the number of units in a layer is known as the **width**. As you might guess, "deep learning" refers to training neural nets with many layers.

Terminology for the depth is very inconsistent. A network with one hidden layer could be called a one-layer, two-layer, or three-layer network, depending if you count the input and output layers.

As an example to illustrate the power of MLPs, let's design one that computes the XOR function. Remember, we showed that linear models cannot do this. We can verbally describe XOR as "one of the inputs is 1, but not both of them." So let's have hidden unit h_1 detect if at least one of the inputs is 1, and have h_2 detect if they are both 1. We can easily do this if we use a hard threshold activation function. You know how to design such units — it's an exercise of designing a binary linear classifier. Then the output unit will activate only if $h_1 = 1$ and $h_2 = 0$. A network which does this is shown in Figure 2.

Let's write out the MLP computations mathematically. Conceptually, there's nothing new here; we just have to pick a notation to refer to various parts of the network. As with the linear case, we'll refer to the activations of the input units as x_j and the activation of the output unit as y. The units in the ℓth hidden layer will be denoted $h_i^{(\ell)}$. Our network is fully connected, so each unit receives connections from all the units in the previous layer. This means each unit has its own bias, and there's a weight for every *pair* of units in two consecutive layers. Therefore, the network's computations can be written out as:

$$\begin{aligned} h_i^{(1)} &= \phi^{(1)}\left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)}\right) \\ h_i^{(2)} &= \phi^{(2)}\left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right) \\ y_i &= \phi^{(3)}\left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}\right) \end{aligned} \tag{1}$$

Note that we distinguish $\phi^{(1)}$ and $\phi^{(2)}$ because different layers may have different activation functions.

Since all these summations and indices can be cumbersome, we usually

write the computations in vectorized form. Since each layer contains multiple units, we represent the activations of all its units with an **activation vector** $\mathbf{h}^{(\ell)}$. Since there is a weight for every pair of units in two consecutive layers, we represent each layer's weights with a **weight matrix** $\mathbf{W}^{(\ell)}$. Each layer also has a **bias vector** $\mathbf{b}^{(\ell)}$. The above computations are therefore written in vectorized form as:

$$\begin{aligned}\mathbf{h}^{(1)} &= \phi^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \\ \mathbf{h}^{(2)} &= \phi^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right) \\ \mathbf{y} &= \phi^{(3)}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right)\end{aligned} \tag{2}$$

When we write the activation function applied to a vector, this means it's applied independently to all the entries.

Recall how in linear regression, we combined all the training examples into a single matrix $\mathbf{X}$, so that we could compute all the predictions using a single matrix multiplication. We can do the same thing here. We can store all of each layer's hidden units for all the training examples as a matrix $\mathbf{H}^{(\ell)}$. Each row contains the hidden units for one example. The computations are written as follows (note the transposes):

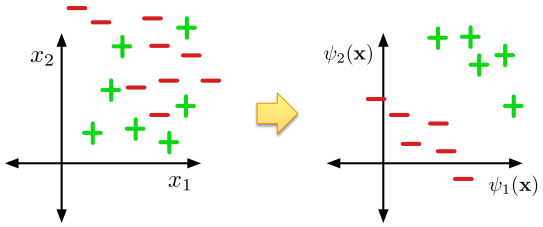
If it's hard to remember when a matrix or vector is transposed, fear not. You can usually figure it out by making sure the dimensions match up.

$$\begin{aligned}\mathbf{H}^{(1)} &= \phi^{(1)}\left(\mathbf{X}\mathbf{W}^{(1)\top} + \mathbf{1}\mathbf{b}^{(1)\top}\right) \\ \mathbf{H}^{(2)} &= \phi^{(2)}\left(\mathbf{H}^{(1)}\mathbf{W}^{(2)\top} + \mathbf{1}\mathbf{b}^{(2)\top}\right) \\ \mathbf{Y} &= \phi^{(3)}\left(\mathbf{H}^{(2)}\mathbf{W}^{(3)\top} + \mathbf{1}\mathbf{b}^{(3)\top}\right)\end{aligned} \tag{3}$$

These equations can be translated directly into NumPy code which efficiently computes the predictions over the whole dataset.

3 Feature Learning

We already saw that linear regression could be made more powerful using a feature mapping. For instance, the feature mapping $\boldsymbol{\psi}(x) = (1, x, x^2, x^e)$ can represent third-degree polynomials. But static feature mappings were limited because it can be hard to design all the relevant features, and because the mappings might be impractically large. Neural nets can be thought of as a way of learning nonlinear feature mappings. E.g., in Figure 1, the last hidden layer can be thought of as a feature map $\boldsymbol{\psi}(\mathbf{x})$, and the output layer weights can be thought of as a linear model using those features. But the whole thing can be trained end-to-end with backpropagation, which we'll cover in the next lecture. The hope is that we can learn a feature representation where the data become linearly separable:



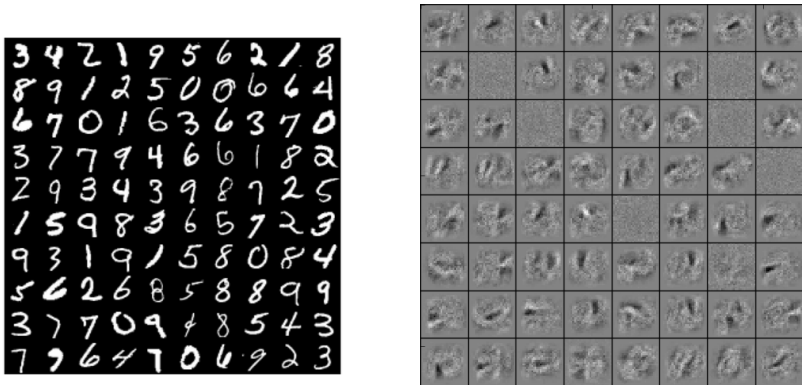


Figure 3: **Left:** Some training examples from the MNIST handwritten digit dataset. Each input is a 28×28 grayscale image, which we treat as a 784-dimensional vector. **Right:** A subset of the learned first-layer features. Observe that many of them pick up oriented edges.

Consider training an MLP to recognize handwritten digits. (This will be a running example for much of the course.) The input is a 28×28 grayscale image, and all the pixels take values between 0 and 1. We'll ignore the spatial structure, and treat each input as a 784-dimensional vector. This is a multiway classification task with 10 categories, one for each digit class. Suppose we train an MLP with two hidden layers. We can try to understand what the first layer of hidden units is computing by visualizing the weights. Each hidden unit receives inputs from each of the pixels, which means the weights feeding into each hidden unit can be represented as a 784-dimensional vector, the same as the input size. In Figure 3, we display these vectors as images.

Later on, we'll talk about convolutional networks, which use the spatial structure of the image.

In this visualization, positive values are lighter, and negative values are darker. Each hidden unit computes the dot product of these vectors with the input image, and then passes the result through the activation function. So if the light regions of the filter overlap the light regions of the image, and the dark regions of the filter overlap the dark region of the image, then the unit will activate. E.g., look at the third filter in the second row. This corresponds to an **oriented edge**: it detects vertical edges in the upper right part of the image. This is a useful sort of feature, since it gives information about the locations and orientation of strokes. Many of the features are similar to this; in fact, oriented edges are a very commonly learned by the first layers of neural nets for visual processing tasks.

It's harder to visualize what the second layer is doing. We'll see some tricks for visualizing this in a few weeks. We'll see that higher layers of a neural net can learn increasingly high-level and complex features.

4 Expressive Power

Linear models are fundamentally limited in their expressive power: they can't represent functions like XOR. Are there similar limitations for MLPs? It depends on the activation function.

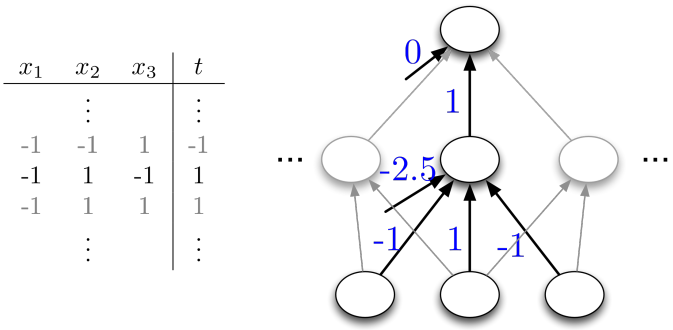


Figure 4: Designing a binary threshold network to compute a particular function.

4.1 Linear networks

Deep linear networks are no more powerful than shallow ones. The reason is simple: if we use the linear activation function $\phi(x) = x$ (and forget the biases for simplicity), the network's function can be expanded out as $\mathbf{y} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdots \mathbf{W}^{(1)}\mathbf{x}$. But this could be viewed as a single linear layer with weights given by $\mathbf{W} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdots \mathbf{W}^{(1)}$. Therefore, a deep linear network is no more powerful than a single linear layer, i.e. a linear model.

4.2 Universality

As it turns out, nonlinear activation functions give us much more power: under certain technical conditions, even a shallow MLP (i.e. one with a single hidden layer) can represent arbitrary functions. Therefore, we say it is **universal**.

This argument can easily be made into a rigorous proof, but this course won't be concerned with mathematical rigor.

Let's demonstrate universality in the case of binary inputs. We do this using the following game: suppose we're given a function mapping input vectors to outputs; we will need to produce a neural network (i.e. specify the weights and biases) which matches that function. The function can be given to us as a table which lists the output corresponding to every possible input vector. If there are D inputs, this table will have 2^D rows. An example is shown in Figure 4. For convenience, let's suppose these inputs are ± 1, rather than 0 or 1. All of our hidden units will use a hard threshold at 0 (but we'll see shortly that these can easily be converted to soft thresholds), and the output unit will be linear.

Our strategy will be as follows: we will have 2^D hidden units, each of which recognizes one possible input vector. We can then specify the function by specifying the weights connecting each of these hidden units to the outputs. For instance, suppose we want a hidden unit to recognize the input $(-1, 1, -1)$. This can be done using the weights $(-1, 1, -1)$ and bias -2.5, and this unit will be connected to the output unit with weight 1. (Can you come up with the general rule?) Using these weights, any input pattern will produce a set of hidden activations where exactly one of the units is active. The weights connecting inputs to outputs can be set based on the input-output table. Part of the network is shown in Figure 4.

Universality is a neat property, but it has a major catch: the network required to represent a given function might have to be extremely large (in particular, exponential). In other words, not all functions can be represented **compactly**. We desire compact representations for two reasons:

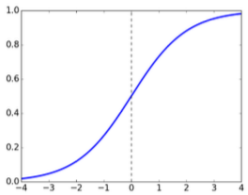
1. We want to be able to compute predictions in a reasonable amount of time.

2. We want to be able to train a network to *generalize* from a limited number of training examples; from this perspective, universality simply implies that a large enough network can memorize the training set, which isn't very interesting.

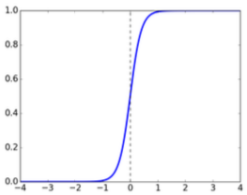
4.3 Soft thresholds

In the previous section, our activation function was a step function, which gives a hard threshold at 0. This was convenient for designing the weights of a network by hand. But recall from last lecture that it's very hard to directly learn a linear classifier with a hard threshold, because the loss derivatives are 0 almost everywhere. The same holds true for multilayer perceptrons. If the activation function for any unit is a hard threshold, we won't be able to learn that unit's weights using gradient descent. The solution is the same as it was in last lecture: we replace the hard threshold with a soft one.

Does this cost us anything in terms of the network's expressive power? No it doesn't, because we can approximate a hard threshold using a soft threshold. In particular, if we use the logistic nonlinearity, we can approximate a hard threshold by scaling up the weights and biases:



$y = \sigma(x)$



$y = \sigma(5x)$

4.4 The power of depth

If shallow networks are universal, why do we need deep ones? One important reason is that deep nets can represent some functions more *compactly* than shallow ones. For instance, consider the parity function (on binary-valued inputs):

$$f_{\text{par}}(x_1, \ldots, x_D) = \begin{cases} 1 & \text{if } \sum_j x_j \text{ is odd} \\ 0 & \text{if it is even.} \end{cases} \tag{4}$$

We won't prove this, but it requires an exponentially large shallow network to represent the parity function. On the other hand, it can be computed by a deep network whose size is *linear* in the number of inputs. Designing such a network is a good exercise.

Lecture 4: Backpropagation

Roger Grosse

1 Introduction

So far, we've seen how to train "shallow" models, where the predictions are computed as a linear function of the inputs. We've also observed that deeper models are much more powerful than linear ones, in that they can compute a broader set of functions. Let's put these two together, and see how to train a multilayer neural network. We will do this using backpropagation, the central algorithm of this course. Backpropagation ("backprop" for short) is a way of computing the partial derivatives of a loss function with respect to the parameters of a network; we use these derivatives in gradient descent, exactly the way we did with linear regression and logistic regression.

If you've taken a multivariate calculus class, you've probably encountered the Chain Rule for partial derivatives, a generalization of the Chain Rule from univariate calculus. In a sense, backprop is "just" the Chain Rule — but with some interesting twists and potential gotchas. This lecture and Lecture 6 focus on backprop. (In between, we'll see a cool example of how to use it.) This lecture covers the mathematical justification and shows how to implement a backprop routine by hand. Implementing backprop can get tedious if you do it too often. In Lecture 6, we'll see how to implement an *automatic differentiation* engine, so that derivatives even of rather complicated cost functions can be computed automatically. (And just as efficiently as if you'd done it carefully by hand!)

This will be your least favorite lecture, since it requires the most tedious derivations of the whole course.

1.1 Learning Goals

- Be able to compute the derivatives of a cost function using backprop.

1.2 Background

I would highly recommend reviewing and practicing the Chain Rule for partial derivatives. I'd suggest Khan Academy[1], but you can also find lots of resources on Metacademy[2].

[1] https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/multivariable-chain-rule/v/multivariable-chain-rule

[2] https://metacademy.org/graphs/concepts/chain_rule

2 The Chain Rule revisited

Before we get to neural networks, let's start by looking more closely at an example we've already covered: a linear classification model. For simplicity, let's assume we have univariate inputs and a single training example (x, t). The predictions are a linear function followed by a sigmoidal nonlinearity. Finally, we use the squared error loss function. The model and loss function are as follows:

$$z = wx + b \tag{1}$$

$$y = \sigma(z) \tag{2}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{3}$$

Now, to change things up a bit, let's add a *regularizer* to the cost function. We'll cover regularizers properly in a later lecture, but intuitively, they try to encourage "simpler" explanations. In this example, we'll use the regularizer $\frac{\lambda}{2}w^2$, which encourages w to be close to zero. (λ is a hyperparameter; the larger it is, the more strongly the weights prefer to be close to zero.) The cost function, then, is:

$$\mathcal{R} = \frac{1}{2}w^2 \tag{4}$$

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}. \tag{5}$$

In order to perform gradient descent, we wish to compute the partial derivatives $\partial\mathcal{J}/\partial w$ and $\partial\mathcal{J}/\partial b$.

This example will cover all the important ideas behind backprop; the only thing harder about the case of multilayer neural nets will be the cruftier notation.

2.1 How you would have done it in calculus class

Recall that you can calculate partial derivatives the same way you would calculate univariate derivatives. In particular, we can expand out the cost function in terms of w and b, and then compute the derivatives using re-

peated applications of the univariate Chain Rule.

$$
\begin{aligned}
\mathcal{L}_{\text{reg}} &= \frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2 \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x + \lambda w \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) + 0 \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

This gives us the correct answer, but hopefully it's apparent from this example that this method has several drawbacks:

1. The calculations are very cumbersome. In this derivation, we had to copy lots of terms from one line to the next, and it's easy to accidentally drop something. (In fact, I made such a mistake while writing these notes!) While the calculations are doable in this simple example, they become impossibly cumbersome for a realistic neural net.

2. The calculations involve lots of redundant work. For instance, the first three steps in the two derivations above are nearly identical.

3. Similarly, the final expressions have lots of repeated terms, which means lots of redundant work if we implement these expressions directly. For instance, $wx+b$ is computed a total of four times between $\partial\mathcal{J}/\partial w$ and $\partial\mathcal{J}/\partial b$. The larger expression $(\sigma(wx+b)-t)\sigma'(wx+b)$ is computed twice. If you happen to notice these things, then perhaps you can be clever in your implementation and factor out the repeated expressions. But, as you can imagine, such efficiency improvements might not always jump out at you when you're implementing an algorithm.

Actually, even in this derivation, I used the "efficiency trick" of not expanding out σ'. If I had expanded it out, the expressions would be even more hideous, and would involve *six* copies of $wx+b$.

The idea behind backpropagation is to share the repeated computations wherever possible. We'll see that the backprop calculations, if done properly, are very clean and modular.

2.2 Multivariable chain rule: the easy case

We've already used the univariate Chain Rule a bunch of times, but it's worth remembering the formal definition:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(g(t)) = f'(g(t))g'(t). \tag{6}$$

Roughly speaking, increasing t by some infinitesimal quantity h_1 "causes" g to change by the infinitesimal $h_2 = g'(t)h_1$. This in turn causes f to change by $f'(g(t))h_2 = f'(g(t))g'(t)h_1$.

The multivariable Chain Rule is a generalization of the univariate one. Let's say we have a function f in two variables, and we want to compute $\frac{d}{dt} f(x(t), y(t))$. Changing t slightly has two effects: it changes x slightly, and it changes y slightly. Each of these effects causes a slight change to f. For infinitesimal changes, these effects combine additively. The Chain Rule, therefore, is given by:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{7}$$

2.3 An alternative notation

It will be convenient for us to introduce an alternative notation for the derivatives we compute. In particular, notice that the left-hand side in all of our derivative calculations is $\mathrm{d}\mathcal{L}/\mathrm{d}v$, where v is some quantity we compute in order to compute $\mathcal{L}$. (Or substitute for $\mathcal{L}$ whichever variable we're trying to compute derivatives of.) We'll use the notation

$$\overline{v} \triangleq \frac{\partial \mathcal{L}}{\partial v}. \tag{8}$$

This notation is less crufty, and also emphasizes that $\overline{v}$ is a value we compute, rather than a mathematical expression to be evaluated. This notation is nonstandard; see the appendix if you want more justification for it.

We can rewrite the multivariable Chain rule (Eqn. 7) using this notation:

$$\overline{t} = \overline{x}\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{9}$$

Here, we use $\mathrm{d}x/\mathrm{d}t$ to mean we should actually evaluate the derivative algebraically in order to determine the formula for $\overline{t}$, whereas $\overline{x}$ and $\overline{y}$ are values previously computed by the algorithm.

2.4 Using the computation graph

In this section, we finally introduce the main algorithm for this course, which is known as **backpropagation**, or **reverse mode automatic differentiation (autodiff)**.[3]

[3] Automatic differentiation was invented in 1970, and backprop in the late 80s. Originally, backprop referred to the special case of reverse mode autodiff applied to neural nets, although the derivatives were typically written out by hand (rather than using an autodiff package). But in the last few years, neural nets have gotten so diverse that we basically think of them as compositions of functions. Also, very often, backprop is now implemented using an autodiff software package. For these reasons, the distinction between autodiff and backprop has gotten blurred, and we will use the terms interchangeably in this course. Note that there is also a forward mode autodiff, but it's rarely used in neural nets, and we won't cover it in this course.

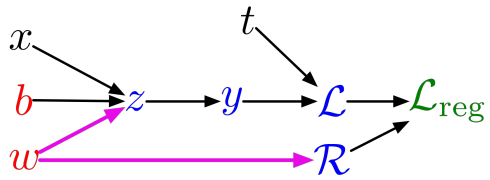


Figure 1: Computation graph for the regularized linear regression example in Section 2.4. The magenta arrows indicate the case which requires the multivariate chain rule because w is used to compute both z and $\mathcal{R}$.

Now let's return to our running example, written again for convenience:

$$
\begin{aligned}
z &= wx + b \\
y &= \sigma(z) \\
\mathcal{L} &= \frac{1}{2}(y - t)^2 \\
\mathcal{R} &= \frac{1}{2}w^2 \\
\mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda\mathcal{R}.
\end{aligned}
$$

Let's introduce the **computation graph**. The nodes in the graph correspond to all the values that are computed, with edges to indicate which values are computed from which other values. The computation graph for our running example is shown in Figure 1.

Note that the computation graph is *not* the network architecture. The nodes correspond to values that are computed, rather than to units in the network.

The goal of backprop is to compute the derivatives $\overline{w}$ and $\overline{b}$. We do this by repeatedly applying the Chain Rule (Eqn. 9). Observe that to compute a derivative using Eqn. 9, you first need the derivatives for its *children* in the computation graph. This means we must start from the result of the computation (in this case, $\mathcal{J}$) and work our way backwards through the graph. It is because we work backward through the graph that backprop and reverse mode autodiff get their names.

Let's start with the formal definition of the algorithm. Let $v_1, \ldots, v_N$ denote all of the nodes in the computation graph, in a topological ordering. (A topological ordering is any ordering where parents come before children.) We wish to compute all of the derivatives $\overline{v_i}$, although we may only be interested in a subset of these values. We first compute all of the values in a **forward pass**, and then compute the derivatives in a **backward pass**. As a special case, v_N denotes the result of the computation (in our running example, $v_N = \mathcal{J}$), and is the thing we're trying to compute the derivatives of. Therefore, by convention, we set $\overline{v_N} = 1$. The algorithm is as follows:

$\overline{\mathcal{J}} = 1$ because increasing the cost by h increases the cost by h.

For $i = 1, \ldots, N$

    Compute v_i as a function of $\text{Pa}(v_i)$

$v_N = 1$

For $i = N - 1, \ldots, 1$

    $\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

Here $\mathrm{Pa}(v_i)$ and $\mathrm{Ch}(v_i)$ denote the parents and children of v_i.

This procedure may become clearer when we work through the example in full:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} \\
&= \overline{\mathcal{L}_{\text{reg}}}\, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} \\
&= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}} \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \\
&= \overline{\mathcal{L}}\, (y - t) \\
\overline{z} &= \overline{y} \frac{\mathrm{d}y}{\mathrm{d}z} \\
&= \overline{y}\, \sigma'(z) \\
\overline{w} &= \overline{z} \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} \\
&= \overline{z}\, x + \overline{\mathcal{R}}\, w \\
\overline{b} &= \overline{z} \frac{\partial z}{\partial b} \\
&= \overline{z}
\end{aligned}$$

Since we've derived a procedure for computing $\overline{w}$ and $\overline{b}$, we're done. Let's write out this procedure without the mess of the derivation, so that we can compare it with the naïve method of Section 2.1:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}}\, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}}\, (y - t) \\
\overline{z} &= \overline{y}\, \sigma'(z) \\
\overline{w} &= \overline{z}\, x + \overline{\mathcal{R}}\, w \\
\overline{b} &= \overline{z}
\end{aligned}$$

The derivation, and the final result, are much cleaner than with the naïve method. There are no redundant computations here. Furthermore, the procedure is *modular*: it is broken down into small chunks that can be reused for other computations. For instance, if we want to change the loss function, we'd only have to modify the formula for $\overline{y}$. With the naïve method, we'd have to start over from scratch.

Actually, there's one redundant computation, since $\sigma(z)$ can be reused when computing $\sigma'(z)$. But we're not going to focus on this point.

3 Backprop on a multilayer net

Now we come to the prototypical use of backprop: computing the loss derivatives for a multilayer neural net. This introduces no new ideas beyond

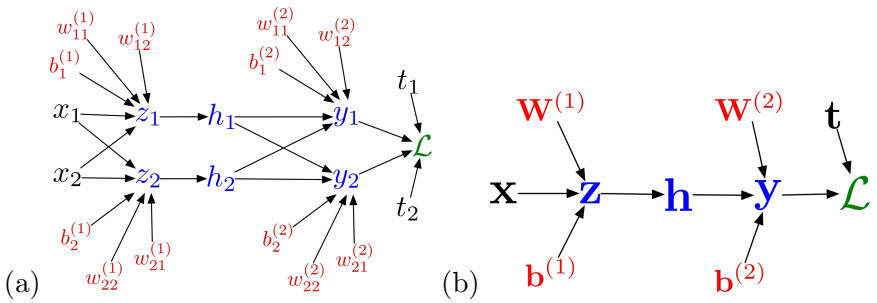


Figure 2: **(a)** Full computation graph for the loss computation in a multilayer neural net. **(b)** Vectorized form of the computation graph.

what we've already discussed, so think of it as simply another example to practice the technique. We'll use a multilayer net like the one from the previous lecture, and squared error loss with multiple output units:

$$
\begin{aligned}
z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\
h_i &= \sigma(z_i) \\
y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\
\mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2
\end{aligned}
$$

As before, we start by drawing out the computation graph for the network. The case of two input dimensions and two hidden units is shown in Figure 2(a). Because the graph clearly gets pretty cluttered if we include all the units individually, we can instead draw the computation graph for the vectorized form (Figure 2(b)), as long as we can mentally convert it to Figure 2(a) as needed.

Based on this computation graph, we can work through the derivations of the backwards pass just as before.

One you get used to it, feel free to skip the step where we write down $\overline{\mathcal{L}}$.

$$
\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\
\overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\
\overline{b_k^{(2)}} &= \overline{y_k} \\
\overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\
\overline{z_i} &= \overline{h_i}\, \sigma'(z_i) \\
\overline{w_{ij}^{(1)}} &= \overline{z_i}\, x_j \\
\overline{b_i^{(1)}} &= \overline{z_i}
\end{aligned}
$$

Focus especially on the derivation of $\overline{h_i}$, since this is the only step which actually uses the multivariable Chain Rule.

Once we've derived the update rules in terms of indices, we can find the vectorized versions the same way we've been doing for all our other calculations. For the forward pass:

$$\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h} &= \sigma(\mathbf{z}) \\
\mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2
\end{aligned}$$

And the backward pass:

$$\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{\mathbf{y}} &= \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t}) \\
\overline{\mathbf{W}^{(2)}} &= \overline{\mathbf{y}}\mathbf{h}^\top \\
\overline{\mathbf{b}^{(2)}} &= \overline{\mathbf{y}} \\
\overline{\mathbf{h}} &= \mathbf{W}^{(2)\top}\overline{\mathbf{y}} \\
\overline{\mathbf{z}} &= \overline{\mathbf{h}} \circ \sigma'(\mathbf{z}) \\
\overline{\mathbf{W}^{(1)}} &= \overline{\mathbf{z}}\mathbf{x}^\top \\
\overline{\mathbf{b}^{(1)}} &= \overline{\mathbf{z}}
\end{aligned}$$

4 Appendix: why the weird notation?

Recall that the partial derivative $\partial\mathcal{J}/\partial w$ means, how much does $\mathcal{J}$ change when you make an infinitesimal change to w, holding everything else fixed? But this isn't a well-defined notion, because it depends what we mean by "holding everything else fixed." In particular, Eqn. 5 defines the cost as a function of two arguments; writing this explicitly,

$$\mathcal{J}(\mathcal{L}, w) = \mathcal{L} + \frac{\lambda}{2}w^2. \tag{10}$$

Computing the partial derivative of this function with respect to w,

$$\frac{\partial\mathcal{J}}{\partial w} = \lambda w. \tag{11}$$

But in the previous section, we (correctly) computed

$$\frac{\partial\mathcal{J}}{\partial w} = (\sigma(wx + b) - t)\sigma'(wx + b)x + \lambda w. \tag{12}$$

What gives? Why do we get two different answers?

The problem is that mathematically, the notation $\partial\mathcal{J}/\partial w$ denotes the partial derivative of a *function* with respect to one of its *arguments*. We make an infinitesimal change to one of the arguments, while holding the rest of the arguments fixed. When we talk about partial derivatives, we need to be careful about what are the arguments to the function. When we compute the derivatives for gradient descent, we treat $\mathcal{J}$ as a function of the parameters of the model — in this case, w and b. In this context,

$\partial \mathcal{J}/\partial w$ means, how much does $\mathcal{J}$ change if we change w while holding b fixed? By contrast, Eqn. 10 treats $\mathcal{J}$ as a function of $\mathcal{L}$ and w; in Eqn. 10, we're making a change to the second argument to $\mathcal{J}$ (which happens to be denoted w), while holding the first argument fixed.

Unfortunately, we need to refer to *both* of these interpretations when describing backprop, and the partial derivative notation just leaves this difference implicit. Doubly unfortunately, our field hasn't consistently adopted any notational conventions which will help us here. There are dozens of explanations of backprop out there, most of which simply ignore this issue, letting the meaning of the partial derivatives be determined from context. This works well for experts, who have enough intuition about the problem to resolve the ambiguities. But for someone just starting out, it might be hard to deduce the meaning from context.

That's why I picked the bar notation. It's the least bad solution I've been able to come up with.

Lecture 5: Distributed Representations

Roger Grosse

1 Introduction

We'll take a break from derivatives and optimization, and look at a particular example of a neural net that we can train using backprop: the neural probabilistic language model. Here, the goal is to model the distribution of English sentences (a task known as language modeling), and we do this by reducing it to a sequential prediction task. I.e., we learn to predict the distribution of the next word in a sentence given the previous words. This lecture will also serve as an example of one of the most important concepts about neural nets, that of a distributed representation. We can understand this in contrast with a localized representation, where a particular piece of information is stored in only one place. In a distributed representation, information is spread throughout the representation. This turns out to be really useful, since it lets us share information between related entities — in the case of language modeling, between related words.

2 Motivation: Language Modeling

Language modeling is the problem of modeling the probability distribution of natural language text. I.e., we would like to be able to determine how likely a given sentence is to be uttered. This is an instance of the more general problem of **distribution modeling**, i.e. learning a model which tries to approximate the distribution which some dataset is drawn from. Why would we want to fit such a model? One of the most important use cases is Bayesian inference.

Suppose we are building a speech recognition system. I.e., given an acoustic signal $\mathbf{a}$, we'd like to infer the sentence $\mathbf{s}$ (or a set of candidate sentences) that was probably spoken. One way to do this is to build a **generative model**. In this case, such a model consists of two probability distributions:

- The **observation model**, represented as $p(\mathbf{a} \mid \mathbf{s})$, which tells us how likely a sentence is to lead to a given acoustic signal. You might, for instance, build a model of the human vocal system. A lot of work has gone into this, but we're not going to talk about it here.

The notation $p(\cdot \mid \cdot)$ denotes the conditional distribution.

- The **prior**, represented as $p(\mathbf{s})$, which tells us how likely a given sentence is to be spoken, before we've seen $\mathbf{a}$. This is the thing we're trying to estimate when we do language modeling.

Given these two distributions, we can combine them using **Bayes' Rule** to infer the **posterior distribution** over sentences, i.e. the probability

distribution over sentences taking into account the observations. Recall that Bayes' Rule is as follows:

$$p(\mathbf{s} \mid \mathbf{a}) = \frac{p(\mathbf{s})\, p(\mathbf{a} \mid \mathbf{s})}{\sum_{\mathbf{s}'} p(\mathbf{s}')\, p(\mathbf{a} \mid \mathbf{s}')}. \tag{1}$$

The denominator is simply a normalization term, and we rarely ever have to compute it or deal with it explicitly. So we can leave the normalization implicit, using the notation $\propto$ to denote proportionality:

$$p(\mathbf{s} \mid \mathbf{a}) \propto p(\mathbf{s})\, p(\mathbf{a} \mid \mathbf{s}). \tag{2}$$

Hence, Bayes' Rule lets us combine our prior beliefs with an observation model in a principled and elegant way.

Having a good prior distribution $p(\mathbf{s})$ is very useful, since speech signals are inherently ambiguous. E.g., "recognize speech" sounds very similar to "wreck a nice beach", but the former is much more likely to be spoken. This is the sort of thing we'd like our language models to capture.

2.1 Autoregressive Models

Now we're going to recast the distribution modeling task as a sequential prediction task. Suppose we're given a corpus of sentences $\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(N)}$. We'll make the simplifying assumption that the sentences are independent. This means that their probabilities multiply:

$$p(\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(N)}) = \prod_{i=1}^{N} p(\mathbf{s}^{(i)}). \tag{3}$$

Hence, we can talk instead about modeling the distribution over *sentences*.

We'll try to fit a model which represents a distribution $p_{\boldsymbol{\theta}}(\mathbf{s})$, parameterized by $\boldsymbol{\theta}$. The **maximum likelihood** criterion says we'd like to choose the $\boldsymbol{\theta}$ which maximizes the **likelihood**, or the probability of the observed data:

$$\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p_{\boldsymbol{\theta}}(\mathbf{s}^{(i)}). \tag{4}$$

At this point, you might be concerned that the probability of any particular sentence will be vanishingly small. This is true, but we can fix that problem by working with log probabilities. Then the probability of the corpus conveniently decomposes as a sum:

$$\log \prod_{i=1}^{N} p(\mathbf{s}^{(i)}) = \sum_{i=1}^{N} \log p(\mathbf{s}^{(i)}). \tag{5}$$

Since it's easier to work with positive numbers, and log probabilities are negative, we often rephrase maximum likelihood as minimizing negative log probabilities.

The *log* probability of monkeys typing the entire works of Shakespeare is on a scale we can reasonably work with. And if slightly better trained monkeys are slightly more likely to type *Hamlet*, it will give us a smooth training criterion we can optimize with gradient descent.

What is this probability, under the assumption that they type all keys uniformly at random?

A sentence is a sequence of words A sentence is a sequence of words $w_1, w_2, \ldots, w_T$. The **chain rule of conditional probability** implies that

$p(\mathbf{s})$ factorizes as the products of conditional probabilities of individual words:

$$p(\mathbf{s}) = p(w_1, \ldots, w_T) = p(w_1)p(w_2 \mid w_1) \cdots p(w_T \mid w_1, \ldots, w_{T-1}). \tag{6}$$

Note that the Chain Rule applies to any distribution, i.e. we're not making any assumptions here.

Hence, the language modeling problem is equivalent to being able to predict the next word!

We typically make a **Markov assumption**, i.e. that the distribution over the next word only depends on the preceding few words. I.e., if we use a context of length 3, this means

$$p(w_t \mid w_1, \ldots, w_{t-1}) = p(w_t \mid w_{t-3}, w_{t-2}, w_{t-1}). \tag{7}$$

Such a model is called **memoryless**, since it has no memory of what occurred earlier in the sentence. When we decompose the distribution modeling problem into a sequential prediction task with limited context lengths, we call that an **autoregressive model**. "Regressive" because it's a prediction problem, and "auto" because the sequences are used as both the inputs and the targets.

Statisticians use "regression" to refer to general supervised prediction problems, not just least squares.

2.2 n-Gram Language Models

The simplest sort of Markov model is a **conditional probability table (CPT)**, where we explicitly represent the distribution over the next word given the context words. This is a table with a row for every possible context word senence, and a column for every word, and the entry gives the conditional probability. Since each row represents a probability distribution, the entries must be nonnegative, and the entries in each row must sum to 1. Otherwise, the numbers can be anything.

The simplest way to estimate a CPT is using the **empirical counts**, i.e. the number of times a sequence of words occurs in the training corpus. For instance,

$$p(w_3 = \text{cat} \mid w_1 = \text{the}, w_2 = \text{fat}) = \frac{\text{count(the fat cat)}}{\text{count(the fat)}} \tag{8}$$

We'll show later in the course that the formula corresponds to the maximum likelihood estimate of the CPT.

This requires counting the number of occurrences of all sequences of length 2 and 3. Sequences of length n are called n-grams, and a model based on counting such sequences is called an n**-gram model**. For $n = 1, 2, 3$, these are called unigram, bigram, and trigram models. See here[1] for some examples of language models. Notice that unigram models are totally incoherent (since they sample all the words independently from the marginal distribution over words), but trigram models capture a fair amount of syntactic structure.

Gotcha: this example is a 3-gram model, even though it uses a context of length 2.

Observe that the number of possible contexts grows exponentially in n. This means that except for very small n, you're unlikely to see all possible n-grams in the training corpus, and many or most of the counts will be 0. This problem is referred to as **data sparsity**. The model described above is somewhat of a straw man, and natural language processing researchers came

[1] https://lagunita.stanford.edu/c4x/Engineering/CS-224N/asset/slp4.pdf#page=10

up with a variety of clever ways for dealing with data sparsity, including adding imaginary counts of all the words, and combining the predictions of different context lengths.

But there's one problem fundamental to the n-gram approach: it's hard to share information between related words. If we see the sentence "The cat got squashed in the garden on Friday", we should estimate a higher probability o seeing the sentence "The dog got flattened in the yard on Monday", even though these two sentences have few words in common. Distributed representations give a great way of doing this.

2.3 Distributed Representations

Conditional probability tables are a kind of **localist representation**, which means a given piece of information (e.g. the probability of seeing "cat" after "the fat") is stored in just one place. If we'd like to share information between related words, we might want to use a **distributed representation**, where the same piece of information would be distributed throughout the whole representation. E.g., suppose we build a table of attributes of words:

	academic	**politics**	**plural**	**person**	**building**
students	1	0	1	1	0
colleges	1	0	1	0	1
legislators	0	1	1	1	0
schoolhouse	1	0	0	0	1

as well as the effect (+ or −) of those attributes on the probabilities of seeing possible next words:

	bill	**is**	**are**	**papers**	**built**	**standing**
academic	−			+		
politics	+			−		
plural		−	+			
person						+
building					+	+

Information about the distribution over the next word is distributed throughout the representation. E.g., the fact that "students" is likely to be followed by "are" comes from the fact that "students" is plural, combined with the fact that plural nouns are likely to be followed by "are". Since "colleges" is also plural, this information is shared between "students" and "colleges".

3 Neural Probabilistic Language Model

Now let's talk about a network that learns distributed representations of language, called the **neural probabilistic language model**, or just **neural language model**. This network is basically a multilayer perceptron. It's an autoregressive model, so we have a prediction task where the input is the sequence of context words, and the output is the distribution over the next word. We associate each word in the dictionary with a unique and arbitrary integer index.

If we write out the negative log-likelihood for a sentence, it decomposes as the sum of cross-entropies for predicting each word:

$$-\log p(\mathbf{s}) = -\log \prod_{t=1}^{T} p(w_t \mid w_1, \ldots, w_{t-1}) \tag{9}$$

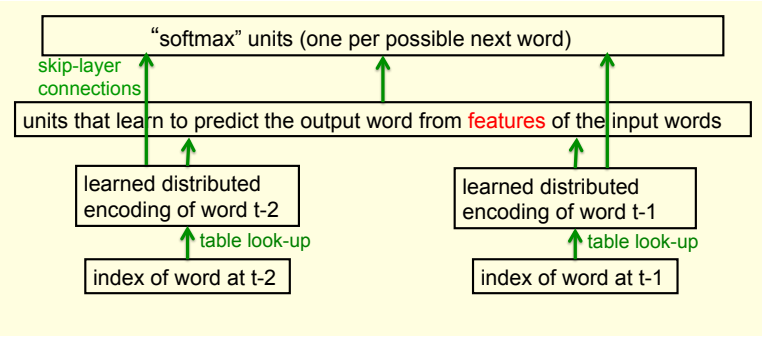
$$= -\sum_{t=1}^{T} \log p(w_t \mid w_1, \ldots, w_{t-1}) \tag{10}$$

$$= -\sum_{t=1}^{T} \log y_{tw_t} \tag{11}$$

$$= -\sum_{t=1}^{T} \sum_{v=1}^{V} t_{tv} \log y_{tv}, \tag{12}$$

where $y_{tv} = p(v \mid w_1, \ldots, w_{t-1})$ is the predicted probability of the next word, and t_{tv} is the one-hot encoding of the target word. So this justifies using cross-entropy loss, just as we did in multiway classification.

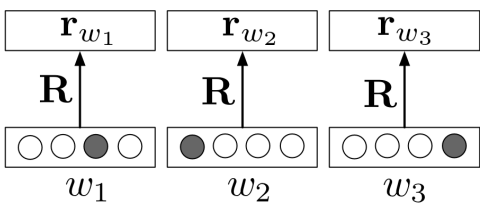
The neural language model uses the following architecture:



The only new concept here is the table look-up in the first layer. The network learns a **representation** of every word in the dictionary as a vector, and keeps these in a lookup table. This can be seen as a matrix $\mathbf{R}$, where each column gives the vector representation of one word. The network does one table lookup for each of the context words, and the activation vector for the **embedding layer** is the concatenation of the representations of all the context words.

There's another way to think of the embedding layer: suppose the context words are represented with one-hot encodings. Then we can think of the embedding layer as basically a linear layer whose weights are shared between all the context words. Recall that a linear layer just computes a matrix-vector product. In this case, we're multiplying the representation matrix $\mathbf{R}$ by the one-hot vectors, which corresponds to pulling out the corresponding column of $\mathbf{R}$.

You sho
this is t



After the embedding layer, there's a hidden layer, followed by a softmax output layer, which is what we'd expect if we're using cross-entropy loss. This architecture also includes a skip connection from the embedding layer to the output layer; we'll talk about skip connections later in the course, but roughly speaking, they help information travel faster through the network. This whole network can be trained using backpropagation, exactly as we've discussed in the previous lecture. You'll implement this for your first homework assignment.

There are various synonyms for word representation:

- **Embedding**, to emphasize that it's a location in a high-dimensional space. As we'll see, semantically related words should be close together.
- **Feature vector**, to emphasize that it picks out semantically relevant features that might be useful for downstream tasks. This is analogous to the polynomial feature mappings for polynomial regression, or the oriented edge filters in our MNIST classifier.
- **Encoding**, to emphasize that it's a sort of code, and that we can go back and forth between the words and their encodings.

Observe that unlike n-gram models, the neural language model is very compact, even for long context lengths. While the size of the CPTs grows exponentially in the context length, the size of the network (number of weights, or number of units) grows linearly in the context length. This means that we can efficiently account for much longer context lengths, such as 10.

If all goes well, the learned representations will reflect the semantic relationships between words. Here are two common ways to measure this:

- If two words are similar, the dot product of their representations, $\mathbf{r}_1^\top \mathbf{r}_2$, should be large.
- If two words are dissimilar, the Euclidean distance between their representations, $\|\mathbf{r}_1 - \mathbf{r}_2\|$, should be large.

These two criteria aren't equivalent in general, but they are equivalent in the case where $\mathbf{r}_1$ and $\mathbf{r}_2$ are both unit vectors:

$$\|\mathbf{r}_1 - \mathbf{r}_2\|^2 = (\mathbf{r}_1 - \mathbf{r}_2)^\top (\mathbf{r}_1 - \mathbf{r}_2) \tag{13}$$

$$= \mathbf{r}_1^\top \mathbf{r}_1 - 2\mathbf{r}_1^\top \mathbf{r}_2 + \mathbf{r}_2^\top \mathbf{r}_2 \tag{14}$$

$$= 2 - 2\mathbf{r}_1^\top \mathbf{r}_2 \tag{15}$$

To visualize the learned word vectors, we need to somehow map them down to two dimensions. There's an algorithm called tSNE that does just that. Roughly speaking, it tries to assign locations to all the words in two dimensions to match the high-dimensional distances as closely as possible. This is impossible to do exactly (e.g. you can't map the vertices of a cube to 2 dimensions while preserving all the distances), and the low-dimensional representation introduces distortions. E.g., words that are far away in high

dimensions might be put close together in 2-D. But it is still a pretty instructive visualization. Here[2] is an example of a tSNE visualization of word representations learned by a different model, but one based on similar principles. Notice that semantically similar words get grouped together.

[2] http://www.cs.toronto.edu/ hinton/turian.png

Lecture 6: Automatic Differentiation

Roger Grosse

1 Introduction

Last week, we saw how the backpropagation algorithm could be used to compute gradients for basically any neural net architecture, as long as it's a feed-forward computation and all the individual pieces of the computation are differentiable. Backprop is based on the computation graph, and it basically works backwards through the graph, applying the chain rule at each node. In that lecture, we would derive the algorithm by hand, in a form that could be translated into a NumPy program. This is how one would have implemented a neural net 10 years ago.

Implementing backprop manually is like writing assembly language: it's good to do a couple times so that you understand how things work beneath the hood.

But, as you may have noticed, the procedure we developed was entirely mechanical. In this lecture, we'll see how to write an *automatic differentiation engine* — a program that builds the computation graph and applies the backprop updates, all without us having to calculate any derivatives by hand. Over the past decade, automatic differentiation frameworks such as Theano, Autograd, TensorFlow, and PyTorch have made it incomparably easier to implement backprop for fancy neural net architectures, thereby dramatically expanding the range and complexity of network architectures we're able to train.

Autodiff has existed since the 70s, but somehow ML people never picked it up until the past decade, despite spending countless hours calculating derivatives.

In this lecture, we'll focus on Autograd[1], a lightweight automatic differentiation system written by Dougal Maclaurin, David Duvenaud, Matt Johnson, and Jamie Townsend. While the major neural net frameworks (TensorFlow, PyTorch, etc.) have giant codebases, much of their complexity comes from supporting a wide variety of neural net layers and operations, and from making sure everything gets the maximum performance out of the GPU. By contrast, Autograd is just an autodiff package: it doesn't include GPU support, and it's not aiming for comprehensive coverage of modern neral net architectures. But this means the implementation is very clean and simple. We'll actually be using a stripped-down, pedagogical implementation of Autograd called Autodidact[2], whose core funcationality is implemented in less than 200 lines of Python code!

Currently, Autograd isn't used much for production neural nets due to its lack of GPU support. For the remainder of the course, we'll use PyTorch, a more comprehensive neural net framework whose autodiff functionality is loosely based on Autograd. But if you wish you could just use Autograd for everything, you're in luck: some of the Autograd creators are working on a framework called JAX[3] which compiles Autograd computation graphs into efficient GPU code. It's a bit experimental now (having been released

[1] https://github.com/HIPS/autograd
[2] https://github.com/mattjj/autodidact
[3] https://github.com/google/jax

December 2018), but you might see this gain more traction over the next few years because of its slick, NumPy-like API.

Some clarifications about terminology are in order. In our field, we usually use "backprop" to refer to the mathematical algorithm and "autodiff" to refer to a software implementation, but the terms are somewhat interchangeable. Sometimes "backprop" just refers to autodiff applied to neural nets, though really the algorithm is no different from the more general case. "Autograd" is the name of a particular software package, but it's often used incorrectly as a generic term for autodiff (e.g. "PyTorch Autograd"). The particular kind of autodiff we use to compute gradients is known as **reverse mode autodiff** because it goes backwards through the computation graph. There's also a **forward mode autodiff**, which is used to compute directional derivatives. We won't used forward mode in this class.

In one of the most elegant pieces of code I've ever seen, it's possible to implement forward mode autodiff in only three lines by calling reverse mode twice. https://github.com/renmengye/tensorflow-forward-ad/issues/2

2 What Autodiff Isn't

One way to motivate autodiff is to contrast it with some related but distinct concepts.

2.1 Autodiff is not finite differences

An easy way to compute derivatives is to approximate them numerically using **finite differences**, or **numerical differentiation**. We just use the definition of derivatives in terms of a limit, but simply plug in a small value of h. The most obvious is the one-sided definition:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}$$

There's also a more numerically stable two-sided version:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h}$$

The approximations are illustrated in Figure 1. Finite differences are commonly used to test the implementation of gradient computations. I.e., we check that the analytic form for the derivatives is close to the finite difference approximation. Since finite differences only requires calculating function values, this is a pretty easy test to write.

However, finite differences is not a practical way to compute derivatives for neural net training. The most obvious reason is that it's very expensive: you need to do a separate forward pass for *each* partial derivative. It's also numerically unstable, since you first subtract two very close values and then divide by a small number. By contrast, autodiff is both efficient and numerically stable.

2.2 Autodiff is not symbolic differentiation

If you've used a mathematical computing environment such as Mathematica or Maple, it's likely you've made use of **symbolic differentiation**. Here, the aim is to take a mathematical expression and return a mathematical expression for the derivative. This is convenient for relatively simple expressions:

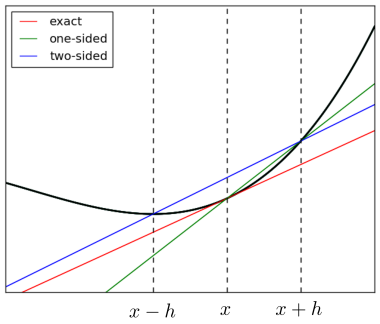


Figure 1: One-sided vs. two-sided finite differences.

```
In[11]:= D[Log[1 + Exp[w * x + b]], w]
Out[11]= e^(b+w x) w / (1 + e^(b+w x))
```

but can get unwieldy for more complex expressions:

```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
Out[19]= e^(b1+b2+w1 x+w2 Log[1+e^(b1+w1 x)]) w2 x / ((1 + e^(b1+w1 x)) (1 + e^(b2+w2 Log[1+e^(b1+w1 x)])))
```

This is unavoidable in symbolic differentiation, since there might not *be* a convenient expression for the derivative.

But notice that part of the problem here is that the expression has a bunch of repeated terms. Imagine we instead defined variables to represent some of these terms, e.g. `z = b1 + w1 x`, and simply referenced those variables in the expression. If you do this consistently enough, you'll basically wind up with autodiff.

3 What autodiff is

Recall from Lecture 4, when we came up with a procedure to compute the derivatives of a simple univariate cost function:

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\overline{\mathcal{L}} = 1$$
$$\overline{y} = y - t$$
$$\overline{z} = \overline{y}\,\sigma'(z)$$
$$\overline{w} = \overline{z}\,x$$
$$\overline{b} = \overline{z}$$

Previously, we would implement a procedure like this as a Python program. But an autodiff package would build up data structures to represent these computations, and then it can simply execute the right-hand side.

One thing we'll change from Lecture 4 is the level of granularity. When we derived backprop by hand, we drew a computation graph in terms of high-level variables that were meaningful in terms of the neural net architecture, and each node corresponded to a mathematical expression that might involve multiple operations. But if the computer is going to do all the work, we might as well define a more fine-grained computation graph, where the nodes correspond to individual **primitive operations**, or **ops**. Ops are basically simple operations, such as multiplication, for which we directly implement the derivatives. An autodiff package would typically break the above computation down into a sequence of primitive ops:

Original program:

$$z = wx + b$$
$$y = \frac{1}{1 + \exp(-z)}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Sequence of ops:

$$t_1 = wx$$
$$z = t_1 + b$$
$$t_3 = -z$$
$$t_4 = \exp(t_3)$$
$$t_5 = 1 + t_4$$
$$y = 1/t_5$$
$$t_6 = y - t$$
$$t_7 = t_6^2$$
$$\mathcal{L} = t_7/2$$

This is all invisible to the user. To the user, it feels like you just write ordinary code for the forward pass, and then call a function to compute the derivatives. E.g., Figure 2 shows an implementation of gradient descent for logistic regression in Autograd. This program includes a call to `grad`, which takes in a function and spits out another function which computes its derivatives. Essentially, this one function is the *only* API you need to learn to use Autograd. Other than that, writing Autograd code looks and feels like writing NumPy. Autograd achieves this by defining its own NumPy package (`autograd.numpy`), which exposes roughly the same API as NumPy, but which does a bunch of additional bookkeeping to build the computation graph needed by `grad`.

4 Implementing autodiff

We now see how to implement a simple but powerful autodiff system. We'll follow the implementation of Autodidact[4], a simpler pedagogical implementation of Autograd. But despite its simplicity, it still gives the full power of autodiff; for instance, you can easily differentiate through a backprop computation (i.e. compute second derivatives by calling `grad` twice), something which is actually fairly awkward to do in TensorFlow or PyTorch! You're encouraged to read the code; the core functionality is less than 200 lines of Python!

There are basically three parts to the Autodidact implementation:

1. Tracing the computation to build the computation graph

[4] https://github.com/mattjj/autodidact

```
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

very sneaky!

… (load the data) …

```
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print  "Trained loss:", training_loss(weights)
```

Autograd constructs a function for computing derivatives

Figure 2: Logistic regression implemented in Autograd.

2. Implementing vector-Jacobian products for each primitive op
3. Backprop itself

4.1 Tracing the computation

Since backprop is done on the computation graph, any autodiff package must first somehow build the computation graph. The approach taken by TensorFlow is for the user to build the graph directly. I.e., the user explicitly builds the graph node by node, and then executes it. PyTorch and Autograd instead build the graph implicitly by **tracing** the computation in the forward pass. This results in a much cleaner interface, because the user doesn't have to worry about any distinction between graph building and execution phases.

The main building block of the computation graph is the `Node` class, which (as you might guess) represents one node of the graph. It keeps track of four pieces of information:

1. `value`, the actual value computed on a particular set of inputs
2. `fun`, the primitive operation defining the node
3. `args` and `kwargs`, the arguments the op was called with
4. `parents`, the parent `Nodes`

In order to implicitly build the computation graph, the `autograd.numpy` module defines an API which looks and feels like NumPy, but where each op

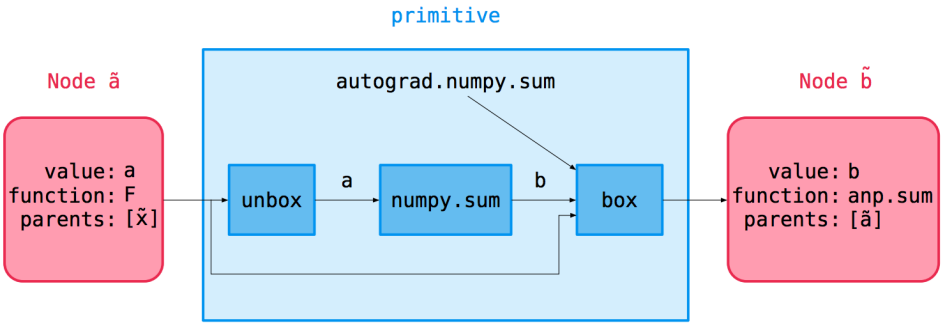


Figure 3: `autograd.numpy` wraps around NumPy operations to implicitly build the computation graph.

```
def logistic(z):
    return 1. / (1. + np.exp(-z))

# that is equivalent to:
def logistic2(z):
    return np.reciprocal(np.add(1, np.exp(np.negative(z))))

z = 1.5
y = logistic(z)
```

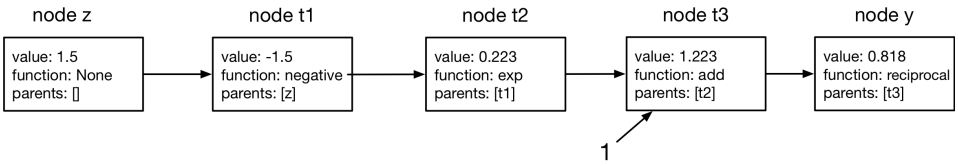


Figure 4: Computation graph built for a simple program. The function `logistic2` is simply an explicit representation of the NumPy functions called when you use arithmetic operators.

wraps around the corresponding NumPy function and, instead of returning an array, returns a `Node` instance containing the array. In particular, it first **unboxes** its inputs (retrieves the values from the `Node` objects), feeds those values to the NumPy function, and then **boxes** the result into a `Node` instance, as shown in Figure 3. Figure 4 shows an example of a small computation graph built up in this way.

The code that builds the computation graph requires the fanciest Python gymnastics of the whole package. We're not going to look at it in detail, but I encourage you to read it.

4.2 Vector-Jacobian products

Recall from Lecture 4 that the vectorized version of Backprop is defined in terms of **vector-Jacobian products (VJPs)**. The **Jacobian matrix** is

the matrix of partial derivatives:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Recall that the error signal for each node is computed using the formula

$$\overline{\mathbf{v}_i} = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}_j}.$$

This can be equivalently written as

$$\overline{\mathbf{v}_i}^\top = \sum_{j \in \mathrm{Ch}(\mathbf{v}_i)} \overline{\mathbf{v}_j}^\top \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i},$$

emphasizing that each piece of the computation involves multiplying a vector by the Jacobian. Hence the term VJP.

Note that we *do not* explicitly construct the Jacobian in order to compute a VJP. For instance, if a node represents a simple elementwise operation, e.g.

$$\mathbf{y} = \exp(\mathbf{z}),$$

then the Jacobian is a diagonal matrix:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} \exp(z_1) & & 0 \\ & \ddots & \\ 0 & & \exp(z_D) \end{pmatrix}.$$

Think about why the Jacobian is diagonal. Why isn't this true of, say, softmax? How would you efficiently implement a VJP for softmax?

This matrix is size $D \times D$, and the explicit matrix-vector product would require $\mathcal{O}(D^2)$ operations. But the VJP itself can be implemented in linear time:

$$\overline{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \overline{\mathbf{y}} = \exp(\mathbf{z}) \circ \overline{\mathbf{y}}.$$

Hence, we need to write procedures to compute VJPs, but we never actually construct the Jacobian.

Take a look at `numpy/numpy_vjps.py`. This module defines VJPs for each NumPy op. Each line is a call to `defvjp` which is a thin wrapper which just adds stuff to a Python dict which stores all the VJP functions. For each op, we need to specify a VJP for *each* of its arguments. The VJP is represented as a function which takes in the output error signal `g`, the value of the node `ans`, and the arguments to the op (which, remember, are `Node` instances). The function returns the input error signal for the corresponding argument. Some examples are shown in Figure 5.

Exercise: write VJPs for division and elementwise log.

4.3 Backprop

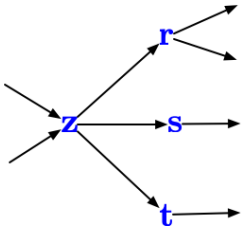
Now we can finally implement backprop! Tracing the computation required the fanciest Python tricks, and implementing VJPs required the most lines of code. It turns out that backprop itself is pretty straightforward.

The one conceptual leap is to think about it as a message-passing procedure. Consider the following computation graph:

```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp,      lambda g, ans, x: ans * g)
defvjp(log,      lambda g, ans, x: g / x)

defvjp(add,         lambda g, ans, x, y : g,
                    lambda g, ans, x, y : g)
defvjp(multiply,    lambda g, ans, x, y : y * g,
                    lambda g, ans, x, y : x * g)
defvjp(subtract,    lambda g, ans, x, y : g,
                    lambda g, ans, x, y : -g)
```

Figure 5: Some example VJPs defined in `numpy/numpy_vjps.py`.

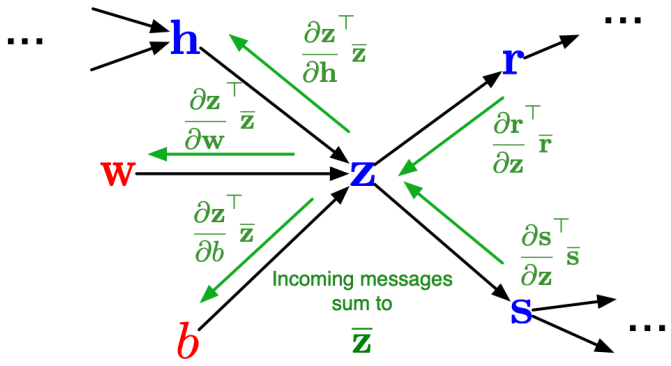


The way we described it in Lecture 4, in order to compute $\bar{\mathbf{z}}$, you'd find the expressions for its child nodes, and differentiate each of them weigh respect to $\mathbf{z}$:

$$\bar{\mathbf{z}} = \frac{\partial \mathbf{r}}{\partial \mathbf{z}}\bar{\mathbf{r}} + \frac{\partial \mathbf{s}}{\partial \mathbf{z}}\bar{\mathbf{s}} + \frac{\partial \mathbf{t}}{\partial \mathbf{z}}\bar{\mathbf{t}}$$

The problem is, this breaks modularity because now the implementation of $\mathbf{z}$ needs to understand the implementations of all its child nodes in order to compute its error signal.

Instead, we can reformulate the algorithm in terms of messages being passed between nodes. Consider the following portion of a graph:



$\mathbf{z}$ receives messages from each of its children, corresponding to their VJPs. But $\mathbf{z}$ doesn't know the messages correspond to VJPs; all it knows is that it needs to sum them together to compute $\bar{\mathbf{z}}$. Once it does that, it computes the VJPs with respect to each of its arguments, and sends them as messages to each of its parent nodes. If you think about it, this is exactly the same sequence of computations as we discussed in Lecture 4, but now each node only has to understand how to compute its own VJPs.

Finally, with this understanding, here's the code which does backprop. It takes two arguments: `end_node`, the output node of the computation graph, and `g`, the output error signal $\overline{\mathcal{L}}$. In this course, we never have any reason to use anything other than the scalar value 1 for the output error signal. However, Autograd lets you use vector-valued outputs, and you can specify any error signal you like.

One use for more general output error signals is if you want to hardcode the backprop procedure for a network while still using autodiff to backprop through individual pieces of it. One example of this is RevNets (Gomez et al., "The reversible residual network: backprop without storing activations"), which uses a special backprop procedure that avoids storing the activations in memory.

```
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad

def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```

The `grad` function is simply a shallow wrapper around `backward_pass`. We haven't talked about it this way, but the entire backprop algorithm is really just computing a VJP, where the Jacobian is of the entire forward computation (viewed as a function mapping parameters to loss). Here's the code:

```
def make_vjp(fun, x):
    """Trace the computation to build the computation graph, and return
    a function which implements the backward pass."""
    start_node = Node.new_root()
    end_value, end_node = trace(start_node, fun, x)
    def vjp(g):
        return backward_pass(g, end_node)
    return vjp, end_value

def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
```

So, that's it. That describes the whole implementation of the core functionality of an autodiff package. You're encouraged to read the Autodidact code[5]. Also, definitely check out the Autograd examples page[6], which contains lots of fun examples like computing higher-order derivatives and differentiating through a fluid dynamics simulator.

[5] https://github.com/mattjj/autodidact
[6] https://github.com/HIPS/autograd

Lecture 7: Optimization

Roger Grosse

1 Introduction

Now that we've seen how to compute derivatives of the cost function with respect to model parameters, what do we do with those derivatives? In this lecture, we're going to take a step back and look at optimization problems more generally. We've briefly discussed gradient descent and used it to train some models, but what exactly is the gradient, and why is it a good idea to move opposite it? We also introduce stochastic gradient descent, a way of obtaining noisy gradient estimates from a small subset of the data.

Using modern neural network libraries, it is easy to implement the backprop algorithm so that it correctly computes the gradient. It's not always so easy to get it to work well. In this lecture, we'll make a list of things that can go drastically wrong in neural net training, and talk about how we can spot them. This includes: learning rates that are too large or too small, symmetries, dead or saturated units, and badly conditioned curvature. We discuss tricks to ameliorate all of these problems. In general, debugging a learning algorithm is like debugging any other complex piece of software: if something goes wrong, you need to make hypotheses about what might have happened, and look for evidence or design experiments to test those hypotheses. This requires a thorough understanding of the principles of optimization.

Understanding the principles of neural nets and being able to diagnose failure modes are what distinguishes someone who's finished CSC421 from someone who's merely worked through the TensorFlow tutorial.

Our style of thinking in this lecture will be very different from that in the last several lectures. When we discussed backprop, we looked at the gradient computations *algebraically*: we derived mathematical equations for computing all the derivatives. We also looked at the computations *implementationally*, seeing how to implement them efficiently (e.g. by vectorizing the computations), and designing an automatic differentiation system which separated the backprop algorithm itself from the design of a network architecture. In this lecture, we'll look at gradient descent *geometrically*: we'll reason qualitatively about optimization problems and about the behavior of gradient descent, without thinking about how the gradients are actually computed. I.e., we *abstract away* the gradient computation. One of the most important skills to develop as a computer scientist is the ability to move between different levels of abstraction, and to figure out which level is most appropriate for the problem at hand.

1.1 Learning goals

- Be able to interpret visualizations of cross-sections of an error surface.
- Know what the gradient is and how to draw it geometrically.

- Know why stochastic gradient descent can be faster than batch gradient descent, and understand the tradeoffs in choosing the mini-batch size.
- Know what effect the learning rate has on the training process. Why can it be advantageous to decay the learning rate over time?
- Be aware of various potential failure modes of gradient descent. How might you diagnose each one, and how would you solve the problem if it occurs?
  - slow progress
  - instability
  - fluctuations
  - dead or saturated units
  - symmetries
  - badly conditioned curvature
- Understand why momentum can be advantageous.

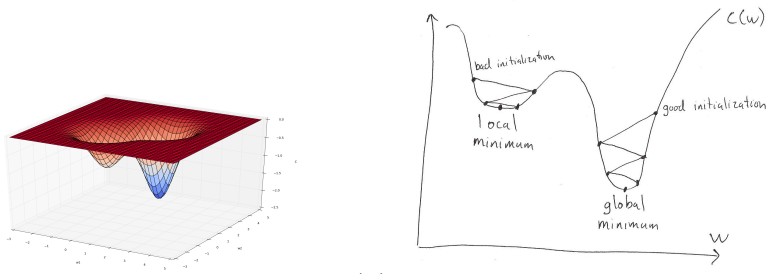
2 Visualizing gradient descent

When we train a neural network, we're trying to minimize some cost function $\mathcal{E}$, which is a function of the network's parameters, which we'll denote with the vector $\boldsymbol{\theta}$. In general, $\boldsymbol{\theta}$ would contain all of the network's weights and biases, and perhaps a few other parameters, but for the most part, we're not going to think about what the elements of $\boldsymbol{\theta}$ represent in this lecture. We're going to think about optimization problems in the abstract. In general, the cost function will be the sum of losses over the training examples; it may also include a regularization term (which we'll discuss in the next lecture). But for the most part, we're not going to think about the particulars of the cost function.

In order to think qualitatively about optimization problems, we'll need some ways to visualize them. Suppose $\boldsymbol{\theta}$ consists of two weights, w_1 and w_2. One way to visualize $\mathcal{E}$ is to draw the **cost surface**, as in Figure 1(a); this is an example of a **surface plot**. This particular cost function has two **local minima**, or points which minimize the cost within a small neighborhood. One of these local optima is also a **global optimum**, a point which achieves the minimum cost over all values of $\boldsymbol{\theta}$. In the context of optimization, local and global minima are also referred to as **local and global optima**.

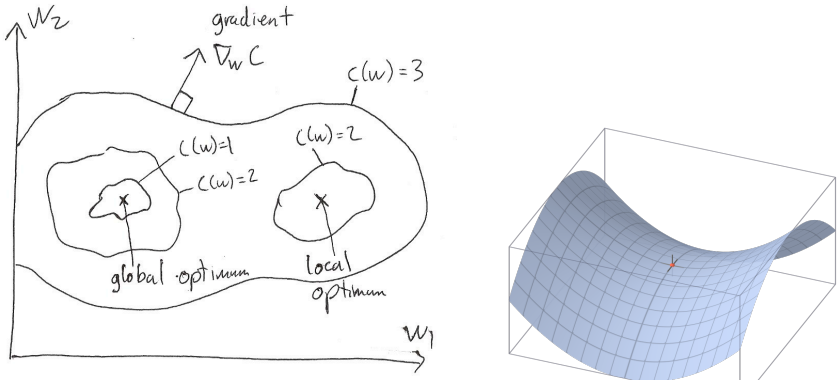
A function can have multiple global minima if there are multiple points that achieve the minimum cost. Technically speaking, global optima are also local optima, but informally when we refer to "local optima," we usually mean the ones which aren't global optima.

Surface plots can be hard to interpret, so we're only going to use them when we absolutely have to. Instead, we'll primarily rely on two other visualizations. First, suppose we have a one-dimensional optimization problem, i.e. $\boldsymbol{\theta}$ consists of a single weight w. We can visualize this by plotting $\mathcal{E}$ as a function of w, as in Figure 1(b). This figure also shows the gradient descent **iterates** (i.e. the points the algorithm visits) starting from two different initializations. One of these sequences converges to the global optimum, and the other one converges to the other local optimum. In general, gradient descent greedily tries to move downhill; by historical accident, it is



(a) (b)

Figure 1: **(a)** Cost surface for an optimization problem with two local minima, one of which is the global minimum. **(b)** Cartoon plot of a one-dimensional optimization problem, and the gradient descent iterates starting from two different initializations, in two different basins of attraction.



(a) (b)

Figure 2: **(a)** Contour plot of a cost function. **(b)** A saddle point.

referred to as a **hill-climbing algorithm**. It's important to choose a good initialization, because we'd like to converge to the global optimum, or at least a good local optimum. The set of weights which lead to a given local optimum are known as a **basin of attraction**.

Figure 1(a) also shows a different feature of the cost function, known as a **plateau** (plural = plateaux). This is a region where the function is flat, or nearly flat, i.e. the derivative is zero or very close to zero. Gradient descent can perform very badly on plateaux, because the parameters change very slowly. In neural net training, plateaux are generally a bigger problem than local optima: while most local optima tend to be good enough in practice, plateaux can cause the training to get stuck on a very bad solution.

Remember when we observed that the gradient of 0–1 loss is zero almost everywhere? That's an example of a plateau.

Figure 2(a) shows a different visualization of a two-dimensional optimization problem: a **contour plot**. Here, the axes correspond to w_1 and w_2; this means we're visualizing weight space (just like we did in our lecture on linear classifiers). Each of the contours represents a **level set**, or set of parameters where the cost takes a particular value.

Check your understanding: how can you (approximately) see local optima on a countour plot? How do you tell which one is the global optimum?

One of the most important things we can visualize on a contour plot is the **gradient**, or the direction of **steepest ascent**, of the cost function,

denoted $\nabla_{\boldsymbol{\theta}} \mathcal{E}$. This is the direction which goes directly uphill, i.e. the direction which increases the cost the fastest relative to the distance moved. We can't determine the magnitude of the gradient from the contour plot, but it is easy to determine its direction: *the gradient is always orthogonal (perpendicular) to the level sets*. This gives an easy way to draw it on a contour plot (e.g. see Figure 2(a)). Algebraically, the gradient is simply the vector of partial derivatives of the cost function:

In this context, $\mathcal{E}$ is taken as a function of the parameters, not of the loss $\mathcal{L}$. Therefore, the partial derivatives correspond to the values $\overline{w_{ij}}$, $\overline{b_i}$, etc., computed from backpropagation.

$$\nabla_{\boldsymbol{\theta}} \mathcal{E} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = \begin{pmatrix} \partial \mathcal{E} / \partial \theta_1 \\ \vdots \\ \partial \mathcal{E} / \partial \theta_M \end{pmatrix} \tag{1}$$

The fact that the vector of partial derivatives gives the steepest ascent direction is far from obvious; you would see the derivation in a multivariable calculus class, but here we will take it for granted.

The **gradient descent** update rule (which we've already seen multiple times) can be written in terms of the gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{E}, \tag{2}$$

where α is the scalar-valued **learning rate**. This shows directly that gradient descent moves opposite the gradient, or in the direction of **steepest descent**. Too large a learning rate can cause instability, whereas too small a learning rate can cause slow progress. In general, the learning rate is one of the most important hyperparameters of a learning algorithm, so it's very important to **tune** it, i.e. look for a good value. (Most commonly, one tries a bunch of values and picks the one which works the best.)

Recall that hyperparameters are parameters which aren't part of the model and which aren't tuned with gradient descent.

For completeness, it's worth mentioning one more possible feature of a cost function, namely a **saddle point**, shown in Figure 2(b). This is a point where the gradient is zero, but which isn't a local minimum because the cost increases in some directions and decreases in others. If we're exactly on a saddle point, gradient descent won't go anywhere because the gradient is zero.

3 Stochastic gradient descent

In machine learning, our cost function generally consists of the average of costs for individual training examples. By linearity of derivatives, the gradient is the average of the gradients for individual examples:

$$\mathcal{E} = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_n \tag{3}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(y^{(n)}, \hat{y}^{(n)}) \tag{4}$$

$$\nabla_{\boldsymbol{\theta}} \mathcal{E} = \nabla_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_n \tag{5}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{E}_n \tag{6}$$

If we use this formula directly, we must visit every training example to compute the gradient. This is known as **batch training**, since we're treating the entire training set as a batch. But this can be very time-consuming, and it's also unnecessary: we can get a stochastic estimate of the gradient from a single training example. In **stochastic gradient descent (SGD)**, we pick a training example, and update the parameters opposite the gradient for that example:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{E}_n. \tag{7}$$

This is identical to the gradient descent update rule, except that $\mathcal{E}$ is replaced with $\mathcal{E}_n$.

SGD is able to make a lot of progress even before the whole training set has been visited. A lot of datasets are so large that it can take hours or longer to make a single pass over the training set; in such cases, batch training is impractical, and we need to use a stochastic algorithm.

In practice, we don't compute the gradient on a single example, but rather average it over a batch of B training examples known as a **mini-batch**. Typical mini-batch sizes are on the order of 100. Why mini-batches? Observe that the number of operations required to compute the gradient for a mini-batch is *linear* in the size of the mini-batch (since mathematically, the gradient for each training example is a separate computation). Therefore, if all operations were equally expensive, one would always prefer to use $B = 1$. In practice, there are two important reasons to use $B > 1$:

- Operations on mini-batches can be vectorized by writing them in terms of matrix operations. This reduces the interpreter overhead, and makes use of efficient and carefully tuned linear algebra libraries.

- Most large neural networks are trained on GPUs or some other architecture which enables a high degree of parallelism. There is much more parallelism to exploit when B is large, since the gradients can be computed independently for each training example.

In previous lectures, we already derived vectorized forms of batch gradient descent. The same formulas can be applied in mini-batch mode.

On the flip side, we don't want to make B too large, because then it takes too long to compute the gradients. In the extreme case where $B = N$, we get batch gradient descent. (The activations for large mini-batches may also be too large to store in memory.)

4 Problems, diagnostics, and workarounds

Now we get to the most important part of this lecture: debugging gradient descent training. When you first learned to program, whenever something didn't work, you might have looked through your code line by line to try and spot the mistake. This might have worked for 10-line programs, but it probably became unworkable for more complex programs. Line-by-line inspection doesn't work very well in machine learning either — not just because the programs are complicated, but also because most of the problems we're going to talk about can occur *even for a correctly implemented training algorithm.* E.g., if the problem is that you set the learning rate too small, you're not going to be able to deduce this by looking at your code, since you don't know ahead of time what the right learning rate is.

Let's make a list of various things that can go wrong, and how to diagnose and fix them.

4.1 Incorrect gradient computations

If your computed gradients are wrong, then all bets are off. If you're lucky, the training will fail completely, and you'll notice that something is wrong. If you're unlucky, it will sort of work, but it will also somehow be broken. This is much more common than you might expect: it's not unusual for an incorrectly implemented learning algorithm to perform reasonably well. But it will perform a bit worse than it should; furthermore, it will make it harder to tune, since some of the diagnostics might give misleading results if the gradients are wrong. Therefore, *it's completely useless to do anything else until you're sure the gradients are correct.*

Fortunately, it's possible to be confident in the correctness of the gradients. We've already covered finite difference methods, which are pretty reliable (see the lecture "Training a Classifier"). If you're using one of the major neural net frameworks, you're pretty safe, because the gradients are being computed automatically by a system which has been thoroughly tested. For the rest of this discussion, we'll assume the gradient computation is correctly implemented.

4.2 Local optima

We're trying to minimize the cost function, and one of the ways we can fail to do this is if we get stuck in a local optimum. Actually, that formulation isn't quite precise, since we rarely converge exactly to any optimum (local or global) when training neural nets. A more precise statement would be, we might wind up in a bad basin of attraction, and therefore not achieve as low a cost as we would be able to in the best basin of attraction.

In general, it's very hard to diagnose if you're in a bad basin of attraction. In many areas of machine learning, one tries to ameliorate the issue using **random restarts**: initialize the training from several random locations, run the training procedure from each one, and pick whichever result has the lowest cost. This is sometimes done in neural net training, but more often we just ignore the problem. In practice, the local optima are usually fine, so we think about training in terms of converging faster to a local optimum, rather than finding the global optimum.

4.3 Symmetries

Suppose we initialize all the weights and biases of a neural network to zero. All the hidden activations will be identical, and you can check by inspection (see the lecture on backprop) that all the weights feeding into a given hidden unit will have identical derivatives. Therefore, these weights will have identical values in the next step, and so on. With nothing to distinguish different hidden units, no learning will occur. This phenomenon is perhaps the most important example of a saddle point in neural net training.

Fortunately, the problem is easy to deal with, using any sort of **symmetry breaking**. Once two hidden units compute slightly different things, they will probably get a gradient signal driving them even farther apart. (Think of this in terms of the saddle point picture; if you're exactly on the saddle point, you get zero gradient, but if you're slightly to one side,

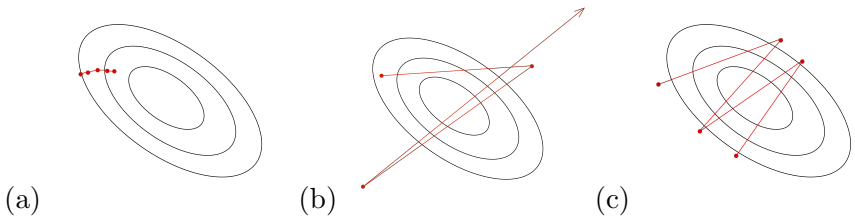


Figure 3: **(a)** Slow progress due to a small learning rate. **(b)** Instability due to a large learning rate. **(c)** Oscillations due to a large learning rate.

you'll move away from it, which gives you a larger gradient, and so on.) In practice, we typically initialize all the weights randomly.

4.4 Slow progress

If the learning rate is too small, gradient descent makes very slow progress, as shown in Figure 3(a). When you plot the training curve, this may show up as a cost which decreases very slowly, but at an approximately linear rate. If you see this happening, then try increasing the learning rate.

4.5 Instability and oscillations

Conversely, if the learning rate is too large, the gradient descent step will overshoot. In some cases, it will overshoot so much that the gradient gets larger, a situation known as **instability**. If this repeats itself, the parameter values and gradient can quickly blow up; this is visualized in Figure 3(b). In the training curve, the cost may appear to suddenly shoot up to infinity. If this is happening, you should decrease the learning rate.

If the learning rate is too large, yet not enough to cause instability, you might get **oscillations**, as shown in Figure 3(c). While the phenomenon might seem easy to spot based on this picture, it's actually pretty hard in practice — keep in mind that weight space is very high-dimensional, and it might not be obvious in which direction to look for oscillations. Also note that oscillations in weight space don't necessarily lead to oscillations in the training curve.

Since we can't detect oscillations, we simply try to tune the learning rate, finding the best value we can. Typically, we do this using a grid search over values spaced approximately by factors of 3, i.e. $\{0.3, 0.1, 0.03, \ldots, 0.0001\}$. The learning rate is one of the most important parameters, and one of the hardest to choose a good value for a priori, so it is usually worth tuning it carefully.

As it happens, there's one more idea which can dampen oscillations while also speeding up training: **momentum**. The physical intuition is as follows: the parameter vector $\boldsymbol{\theta}$ is treated as a particle which is moving through a field whose potential energy function is the cost $\mathcal{E}$. The gradient does not determine the velocity of the particle (as it would in SGD), but rather the acceleration. As a rough intuition, imagine you've built a surface in 3-D corresponding to a 2-D cost function, and you start a frictionless ball rolling from somewhere on that surface. If the surface is sufficiently flat,

the dynamics are essentially those described above. (The potential energy is the height of the surface.)

We can simulate these dynamics with the following update rule, known as **gradient descent with momentum**. (Momentum can be used with either the batch version or with SGD.)

$$\mathbf{p} \leftarrow \mu\mathbf{p} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{E}_n \tag{8}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p} \tag{9}$$

Just as with ordinary SGD, there is a learning rate α. There is also another parameter μ, called the **momentum parameter**, satisfying $0 \leq \mu \leq 1$. It determines the timescale on which momentum decays. In terms of the physical analogy, it determines the amount of friction (with $\mu = 1$ being frictionless). As usual, it's useful to think about the edge cases:

- $\mu = 0$ yields standard gradient descent.
- $\mu = 1$ is frictionless, so momentum never decays. This is problematic because of conservation of energy. We would like to minimize the cost function, but whenever the particle gets near the optimum, it has low potential energy, and hence high kinetic energy, so it doesn't stay there very long. We need $\mu < 1$ in order for the energy to decay.

In practice, $\mu = 0.9$ is a reasonable value. Momentum sometimes helps a lot, and it hardly ever hurts, so using momentum is standard practice.

4.6 Fluctuations

All of the problems we've discussed so far occur both in batch training and in SGD. But in SGD, we have the further problem that the gradients are stochastic; even if they point in the right direction on average, individual stochastic gradients are noisy and may even increase the cost function. The effect of this noise is to push the parameters in a random direction, causing them to **fluctuate**. Note the difference between oscillations and fluctuations: oscillations are a systematic effect caused by the cost surface itself, whereas fluctuations are an effect of the stochasticity in the gradients.

Fluctuations often show up as fluctuations in the cost function, and can be seen in the training curves. One solution to fluctuations is to decrease the learning rate; however, this can slow down the progress too much. It's actually fine to have fluctuations during training, since the parameters are still moving in the right direction "on average."

A better approach to deal with fluctuations is **learning rate decay**. My favorite approach is to keep the learning rate relatively high throughout training, but then at the very end, to decay it using an exponential schedule, i.e.

$$\alpha_t = \alpha_0 e^{-t/\tau}, \tag{10}$$

where α_0 is the initial learning rate, t is the iteration count, τ is the **decay timescale**, and $t = 0$ corresponds to the start of the decay.

I should emphasize that we *don't begin the decay until late in training*, when the parameters are already pretty good "on average" and we merely have a high cost because of fluctuations. Once you start decaying α, progress

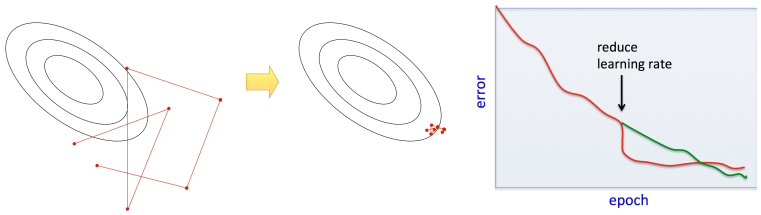


Figure 4: If you decay the learning rate too soon, you'll get a sudden drop in the loss as a result of reducing fluctuations, but the algorithm will stop making progress towards the optimum, leading to slower convergence in the long run. This is a big problem in practice, and we haven't figured out any good ways to detect if this is happening.

slows down drastically. If you decay α too early, you may get a sudden improvement in the cost from reducing fluctuations, at the cost of failure to converge in the long term. This phenomenon is illustrated in Figure 4.

Another neat trick for dealing with fluctuations is **iterate averaging**. Separate from the training process, we keep an **exponential moving average** $\tilde{\boldsymbol{\theta}}$ of the iterates, as follows:

$$\tilde{\boldsymbol{\theta}} \leftarrow \left(1 - \frac{1}{\tau}\right) \tilde{\boldsymbol{\theta}} + \frac{1}{\tau} \boldsymbol{\theta}. \tag{11}$$

τ is a hyperparameter called the **timescale**. Iterate averaging doesn't change the training algorithm itself at all, but when we apply or evaluate the network, we use $\tilde{\boldsymbol{\theta}}$ rather than $\boldsymbol{\theta}$. In practice, iterate averaging can give a huge performance boost by reducing the fluctuations.

4.7 Dead and saturated units

Another tricky problem is that of **saturated units**, i.e. units whose activations are nearly always near the ends of their dynamic range (i.e. the range of possible values). An important special case is that of **dead units**, units whose activations are always very close to zero. To understand why saturated units are problematic, we need to revisit one of the equations we derived for backprop. Suppose $h_i = \phi(z_i)$, where ϕ is a sigmoidal nonlinearity (such as the logistic function). Then:

$$\overline{z_i} = \overline{h_i} \frac{\mathrm{d}h_i}{\mathrm{d}zi} = \overline{h_i}\, \phi'(z_i). \tag{12}$$

If h is near the edge of its dynamic range, then $\phi'(z)$ is very small. (Think about why this is the case.) Therefore, $\overline{z}$ is also very small, and no gradient signal will pass through this node in the computation graph. In particular, all the weights that feed into z_i will get no gradient signal:

$$\overline{w_{ij}} = \overline{z_i}\, x_j \approx 0 \tag{13}$$

$$\overline{b_i} = \overline{z_i} \approx 0. \tag{14}$$

If the incoming weights and bias don't change, then this unit can stay saturated for a long time. In terms of our visualizations, this situation corresponds to a plateau.

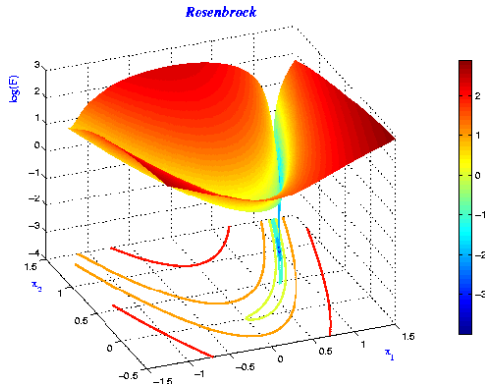


Figure 5: The Rosenbrock function, a function which is commonly used as an optimization benchmark and demonstrates badly conditioned curvature (i.e. a ravine).

Diagnosing saturated units is simple: just look at a histogram of the average activations, and make sure they're not concentrated at the endpoints.

Preventing saturated units is pretty hard, but there are some tricks that help. One trick is to carefully choose the scale of the random initialization of the weights so that the activations are in the middle of their dynamic range. One such trick is the "Xavier initialization", named after one of its inventors[1].

Another way to avoid saturation is to use an activation function which doesn't saturate. Linear activation functions would fit the bill, but unfortunately we saw that deep linear networks aren't any more powerful than shallow ones. Instead, consider **rectified linear units (ReLUs)**, which have the activation function

$$\phi(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0. \end{cases} \tag{15}$$

ReLU units don't saturate for positive z, which is convenient. Unfortunately, they can die if z is consistently negative, so it helps to initialize the biases to a small positive value (such as 0.1).

4.8 Badly conditioned curvature

All of the problems we've discussed so far are fairly specific things that can be attenuated using simple tricks. But there's one more problem that's fundamentally very hard to deal with: badly conditioned curvature. Let's unpack this. Intuitively, **curvature** refers to how fast the function curves upwards when you move in a given direction. In directions of high curvature, you want to take a small step, because you can overshoot very quickly.

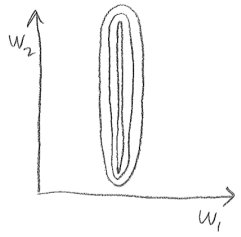
In directions of low curvature, you want to take a large step, because there's a long distance you need to travel. But what actually happens in gradient descent is precisely the opposite: it likes to take large steps in high curvature directions and small steps in low curvature directions. If

The curvature and its conditioning are formalized in terms of the eigenvalues of the matrix of second derivatives of $\mathcal{E}$, but we won't go into that here.

[1] X. Glorot and Y. Bengio, 2010. Understanding the difficulty of training deep feedforward neural networks. AISTATS

(a)

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮



(b)

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮

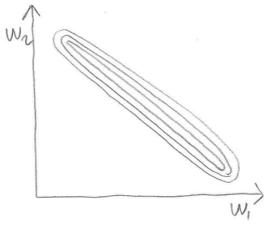


Figure 6: Unnormalized data can lead to badly conditioned curvature. **(a)** The two inputs have vastly different scales. Changing w_1 has a much bigger effect on the model's predictions than changing w_2, so the cost function curves more rapidly along that dimension. **(b)** The two inputs are offset by about the same amount. Changing the weights in a direction that preserves $w_1 + w_2$ has little effect on the predictions, while changing $w_1 + w_2$ has a much larger effect.

the curvature is very different in different directions, we say the curvature is **badly conditioned**. An example is shown in Figure 5. A region with badly conditioned curvature is sometimes called a **ravine**, because of what it looks like in a surface plot of the cost function. Think about the effect this has on optimization. You need to set α small enough that you don't get oscillations or instability in the high curvature directions. But if α is small, then progress will be very slow in the low curvature directions.

In practice, neural network training is very badly conditioned. This is likely a big part of why modern neural nets can take weeks to train. Much effort has been spent researching second-order optimization methods, alternatives to SGD which attempt to correct for the curvature. Unfortunately, these methods are complicated and pretty hard to tune (in the context of neural nets), so SGD is still the go-to algorithm, and we just have to live with badly conditioned curvature.

However, we can at least try to eliminate particular egregious instances of badly conditioned curvature. One way in which badly conditioned curvature can arise is if the inputs have very different scales or are off-center. See Figure 6 for examples of this effect in linear regression problems. Such examples could arise because inputs represent arbitrary units, such as feet or years. This framing almost immediately suggests a workaround: **normalize** the inputs so that they have zero mean and unit variance. I.e., take

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}, \tag{16}$$

where $\mu_j = \mathbb{E}[x_j]$ and $\sigma_j^2 = \mathrm{Var}(x_j)$.

It's worth mentioning two very popular algorithms which help with badly

It is perhaps less intuitive why having the *means* far from zero causes badly conditioned curvature, but rest assured this is an important effect, and worth combating.

conditioned curvature: batch normalization and Adam. We won't cover them properly, but the original papers are very readable, in case you're curious.[2] Batch normalization normalizes the activations of each layer of a network to have zero mean and unit variance. This can help significantly for the reason outlined above. (It can also attenuate the problem of saturated units.) Adam separately adapts the learning rate of each individual parameter, in order to correct for differences in curvature along individual coordinate directions.

4.9 Recap

Here is a table to summarize all the pitfalls, diagnostics, and workarounds that we've covered:

Problem	**Diagnostics**	**Workarounds**
incorrect gradients	finite differences	fix them, or use an autodiff package
local optima	(hard)	random restarts
symmetries	visualize $\mathbf{W}$	initialize $\mathbf{W}$ randomly
slow progress	slow, linear training curve	increase α
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of $\mathbf{W}$; ReLU
badly conditioned curvature	(hard)	normalization; momentum; Adam; second-order opt.

[2] D. P. Kingma and J. L. Ba, 2015. Adam: a method for stochastic optimization. ICLR
S. Ioffe and C. Szegedy, 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift.

Lecture 9: Convolutional Networks

Roger Grosse

1 Introduction

So far, all the neural networks we've looked at consisted of layers which computed a linear function followed by a nonlinearity:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x}). \tag{1}$$

We never gave these layers a name, since they're the only thing we used. Now we will. They're called **fully connected** layers, because every one of the input units is connected to every one of the output units. While fully connected layers are useful, they're not always what we want. Here are some reasons:

- They require a lot of connections: if the input layer has M units and the output layer has N units, then we need MN connections. This can be quite a lot; for instance, suppose the input layer is an image consisting of $M = 256 \times 256 = 65563$ grayscale pixels, and the output layer consists of $N = 1000$ units (modest by today's standards). A fully connected layer would require 65 million connections. This causes two problems:
  - Computing the hidden activations requires one add-multiply operation per connection in the network, so large numbers of connections can be expensive.
  - Each connection has a separate weight parameter, so we would need a huge number of training examples in order to avoid overfitting.
- If we're trying to classify an image or an audio waveform, there's certain structure we'd like to make use of. For instance, features (such as edges) which are useful at one image location are likely to be useful at other locations as well. We would like to **share structure** between different parts of the network. Another property we'd like to make use of is **invariance**: if the image or waveform is transformed slightly (e.g. by shifting it a few pixels), the classification shouldn't change. Both of these properties should be encoded into the network's architecture if possible.

For the next three lectures, we'll talk about a particular kind of network architecture which deals with all these issues: the **convolutional network**, or **conv net** for short. Like the name suggests, the architecture is inspired by a mathematical operator called **convolution** (which we'll explain shortly).

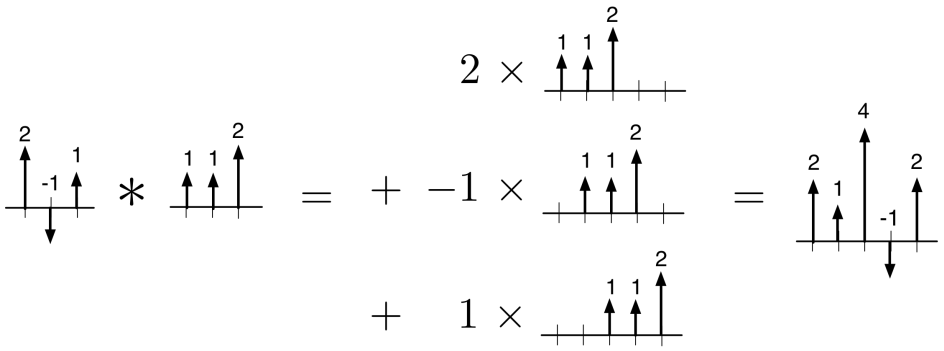


Figure 1: Translate-and-scale interpretation of convolution of one-dimensional signals.

Conv nets revolutionized the field of computer vision in 2012, and by now, the vast majority of papers published in top computer vision conferences use conv nets in some way. Fortunately, the ideas aren't terribly complicated, and by the end of these three lectures, you'll understand how these things work. With a relatively small number of lines of code in a framework like PyTorch or TensorFlow, you can build a computer vision system more powerful than the state-of-the-art just a few years ago.

2 Convolution

Before we talk about conv nets, let's introduce convolution. Suppose we have two **signals** x and w, which you can think of as arrays, with elements denoted as $x[t]$ and so on. As you can guess based on the letters, you can think of x as an input signal (such as a waveform or an image) and w as a set of weights, which we'll refer to as a **filter** or **kernel**. Normally the signals we work with are finite in extent, but it is sometimes convenient to treat them as infinitely large by treating the values as zero everywhere else; this is known as **zero padding**.

Let's start with the one-dimensional case. The **convolution** of x and w, denoted $x * w$, is a signal with entries given by

$$(x * w)[t] = \sum_{\tau} x[t - \tau]\, w[\tau]. \tag{2}$$

There are two ways to think about this equation. The first is **translate-and-scale**: the signal $x * w$ is composed of multiple copies of x, translated and scaled by various amounts according to the entries of w. An example of this is shown in Figure 1.

A second way to think about it is **flip-and-filter**. Here we generate each of the entries of $x * w$ by flipping w, shifting it, and taking the dot product with x. An example is shown in Figure 2.

The two-dimensional case is exactly analogous to the one-dimensional case; we apply the same definition, but with more indices:

$$(x * w)[s, t] = \sum_{\sigma,\tau} x[s - \sigma, t - \tau]\, w[\sigma, \tau]. \tag{3}$$

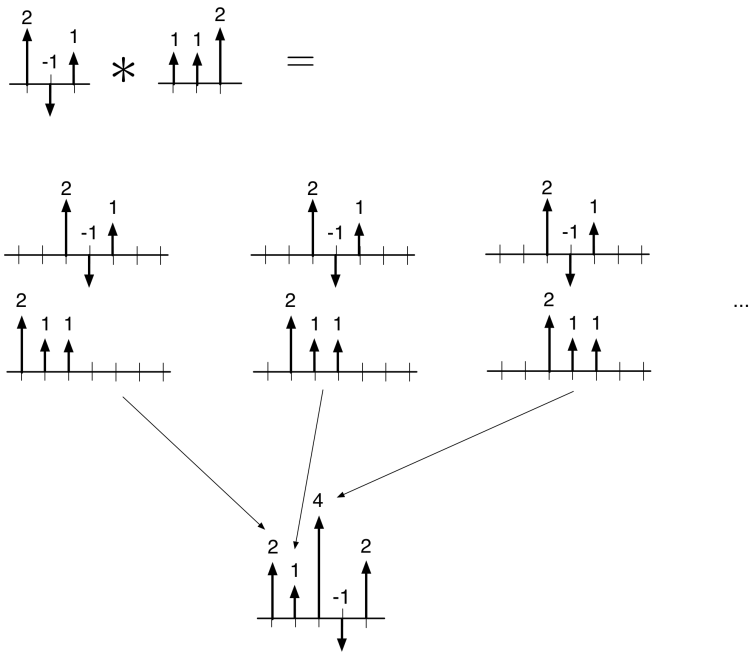
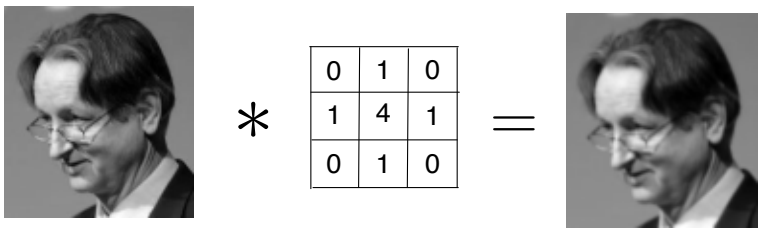


Figure 2: Flip-and-filter interpretation of convolution of one-dimensional signals.

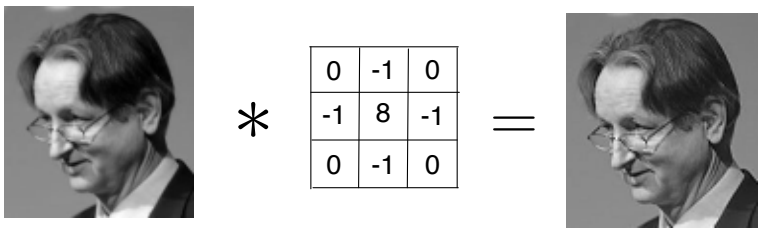
This is shown graphically in Figures 3 and 4.

2.1 Examples

Despite the simplicity of the operation, convolution can do some pretty interesting things. For instance, we can blur an image:



We can sharpen it:



If we change the values slightly, we get a very different effect. (Why? What is the difference from the previous example?) This is a center-surround filter, and it responds only to boundaries.

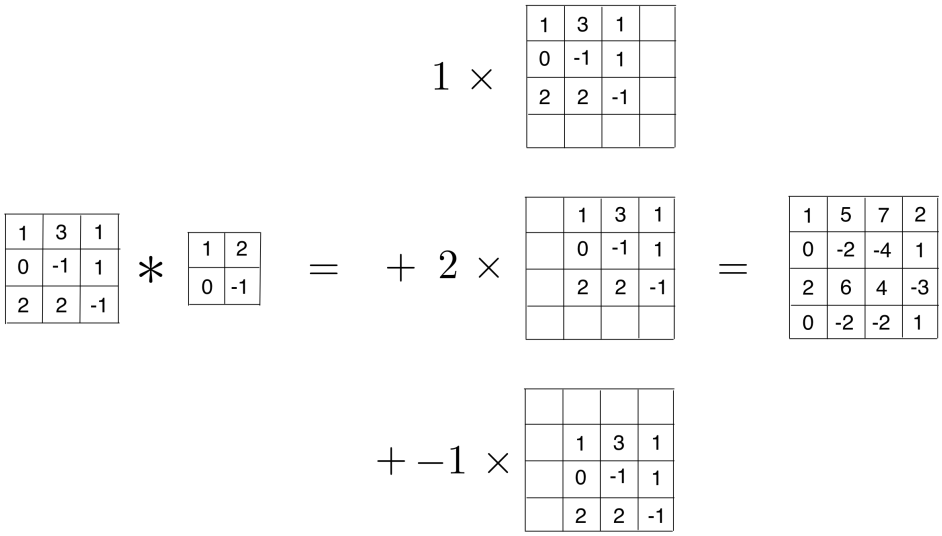


Figure 3: Translate-and-scale interpretation of convolution of two-dimensional signals.

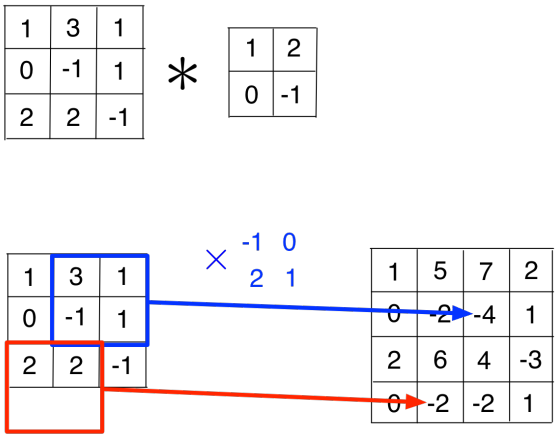
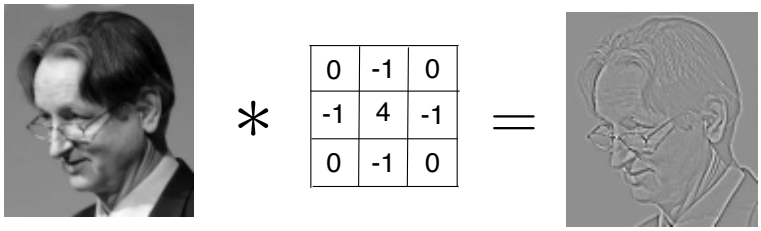
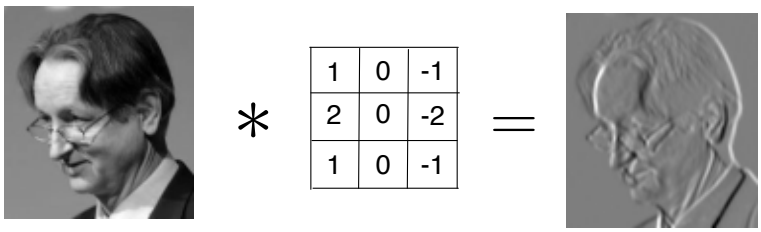


Figure 4: Flip-and-filter interpretation of convolution of two-dimensional signals.



We can detect edges. (That is, edges in the image itself, rather than edges in the world. Detecting edges in the world is a very hard problem.) This filter is known as a Sobel filter.



2.2 Properties of convolution

Now that we've seen some examples of convolution, let's note some useful properties. First of all, it behaves like multiplication, in that it's commutative and associative:

It's a good exercise to verify both properties from the definition.

$$u * v = v * u \tag{4}$$

$$(u * v) * w = u * (v * w). \tag{5}$$

While both properties follow easily from the definition, they're a bit surprising and counterintuitive when you think about flip-and-filter. For instance, let's say you blur the image and then run a horizontal edge filter, represented as $(x * w_{\text{blur}}) * w_{\text{horz}}$. By commutativity and associativity, this is equivalent to first running the edge filter, and then blurring the result, i.e. $(x * w_{\text{horz}}) * w_{\text{blur}}$. It's also equivalent to convolving the image with a single kernel which is obtained by blurring the edge kernel: $x*(w_{\text{horz}}*w_{\text{blur}})$.

Another useful property of convolution is that it is **linear**:

$$(ax + bx') * w = ax * w + bx' * w \tag{6}$$

$$x * (aw + bw') = ax * w + bx * w'. \tag{7}$$

This is convenient, because linear operations are often easier to deal with. But it also shows an inherent limit to convolution: if you have a neural net which computes lots of convolutions in sequence, it can still only compute linear functions. In order to compute more complex operations, we'll need to apply some sort of nonlinear activation function in each layer. (More on this later.)

One last property of convolution is that it's **equivariant** to translation. This means that if we shift, or translate, x by some amount, then the output $x*w$ is shifted by the same amount. This is a useful property in the context of neural nets, because it means the network's computations behave in a well-defined way as we transform the inputs.

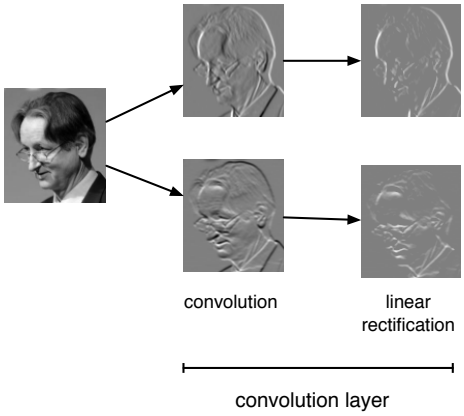


Figure 5: Detecting horizontal and vertical edge features.

2.3 Convolutional feature detection

As alluded to above, convolutions are even more powerful when they're paired with nonlinearities. A sequence of convolutions can only compute a linear function, but a sequence of convolutions alternated with nonlinearities can do fancier things. E.g., consider the following sequence of operations:

1. Convolve the image with a horizontal edge filter
2. Apply the linear rectification nonlinearity

$$\phi(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{8}$$

3. Blur the result.

This sequence of steps, shown in Figure 5, gives a map of horizontalness in various parts of an image; the same can be done for verticalness. You can hopefully imagine this being a useful feature for further processing. Because the resulting output can be thought of as a map of the feature strength over parts of an image, we refer to it as a **feature map**.

3 Convolution layers

We just saw that a convolution, followed by a nonlinear activation function, followed by another convolution, could compute something interesting. This motivates the **convolution layer**, a neural net layer which computes convolutions followed by a nonlinear activation function. Since convolution layers can be thought of as doing feature detection, they're sometimes referred to as **detection layers**. First, let's see how we can think about convolution in terms of units and connections.

Confusingly, the way they're standardly defined, convolution layers don't actually compute convolutions, but a closely related operation called **filtering**:

$$(x \star w)[t] = \sum_{\tau} x[t+\tau]\, w[\tau]. \tag{9}$$

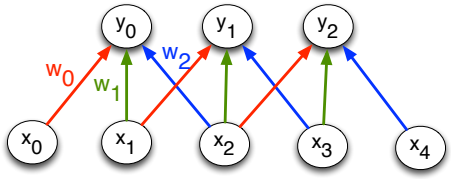


Figure 6: A convolution layer, shown in terms of units and connections.

Like the name suggests, filtering is essentially like flip-and-filter, but without the flipping. (I.e., $x * w = x \star \text{flip}(w)$.) The two operations are basically equivalent — the difference is just a matter of how the filter (or kernel) is represented.

In the above example, we computed a single feature map, but just as we normally use more than one hidden unit in fully connected layers, convolution layers normally compute multiple feature maps $z_1, \ldots, z_M$. The input layers also consist of multiple feature maps $x_1, \ldots, x_D$; these could be different color channels of an RGB image, or feature maps computed by another convolution layer. There is a separate filter w_{ij} associated with each pair of an input and output feature map. The activations are computed as follows:

$$z_i = \sum_j x_j \star w_{ij} \tag{10}$$

$$h_i = \phi(z_i) \tag{11}$$

The activation function ϕ is applied elementwise.

We can think about filtering as a layer of a neural network by thinking of the elements of x and $x * w$ as units, and the elements of w as connection weights. Such an interpretation is visualized in Figure 6 for a one-dimensional example. Each of the units in this network computes its activations in the standard way, i.e. by summing up each of the incoming units multiplied by their connection weights. This shows that a convolution layer is like a fully connected layer, except with two additional features:

- **Sparse connectivity**: not every input unit is connected to every output unit.
- **Weight sharing**: the network's weights are each shared between multiple connections.

Missing connections can be thought of as connections with weight 0. This highlights an important fact: *any function computed by a convolution layer can be computed by a fully connected layer.*

This means convolution layers don't increase the representational capacity, relative to a fully connected layer with the same number of input and output units. But they can reduce the numbers of weights and connections. For instance, suppose we have 32 input feature maps and 16 output feature maps, all of size 50×50, and the filters are of size 5×5. (These are all plausible sizes for a conv net.) The number of weights for the convolution layer is

$$5 \times 5 \times 16 \times 32 = 12,800.$$

The number of connections is approximately

$$50 \times 50 \times 5 \times 5 \times 16 \times 32 = 32 \text{ million.}$$

By contrast, the number of connections (and hence also the number of weights) required for a fully connected layer with the same set of units would be

$$(32 \times 50 \times 50) \times (16 \times 50 \times 50) = 3.2 \text{ billion.}$$

Hence, using the convolutional structure reduces the number of connections by a factor of 100 and the number of weights by almost a factor of a million!

4 Pooling layers

In the introduction to this lecture, we observed that a neural network's classifications ought to be invariant to small transformations of an image, such as shifting it by a few pixels. In order to achieve invariance, we introduce another kind of layer: the **pooling layer**. Pooling layers summarize (or compress) the feature maps of the previous layer by computing a simple function over small regions of the image. Most commonly, this function is taken to be the maximum, so the operation is known as **max-pooling**.

Suppose we have input feature maps $x_1, \ldots, x_N$. Each unit of the output map computes the maximum over some region (called a **pooling group**) of the input map. (Typically, the region could be 3×3.) In order to shrink the representation, we don't consider all offsets, but instead we space them by a **stride** S along each dimension. This results in the representation being shrunk by a factor of approximately S along each dimension. (A typical value for the stride is 2.)

Figure 7 shows an example of how pooling can provide partial invariance to translations of the input.

Pooling also has the effect of increasing the size of units' **receptive fields**, or the regions of the input image which influence their activations. For instance, consider the network architecture in Figure 8, which alternates between convolution and pooling layers. Suppose all the filters are 5×5 and the pooling layer uses a stride of 2. Then each unit in the first convolution layer has a receptive field of size 5×5. But each unit in the second convolution layer has a receptive field of size approximately 10×10, since it does 5×5 filtering over a representation which was shrunken by a factor of 2 along each dimension. A third convolution layer would have 20×20 receptive fields. Hence, pooling allows small filters to account for information over large regions of an image.

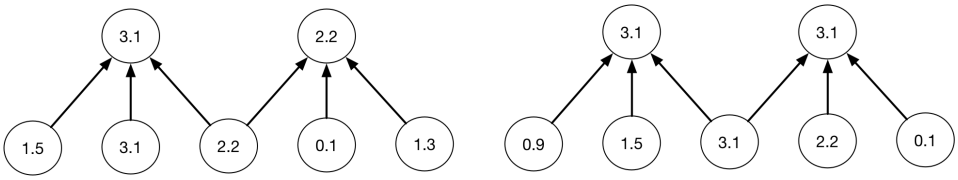


Figure 7: An example of how pooling can provide partial invariance to translations of the input. Observe that the first output does not change, since the maximum value remains within its pooling group.

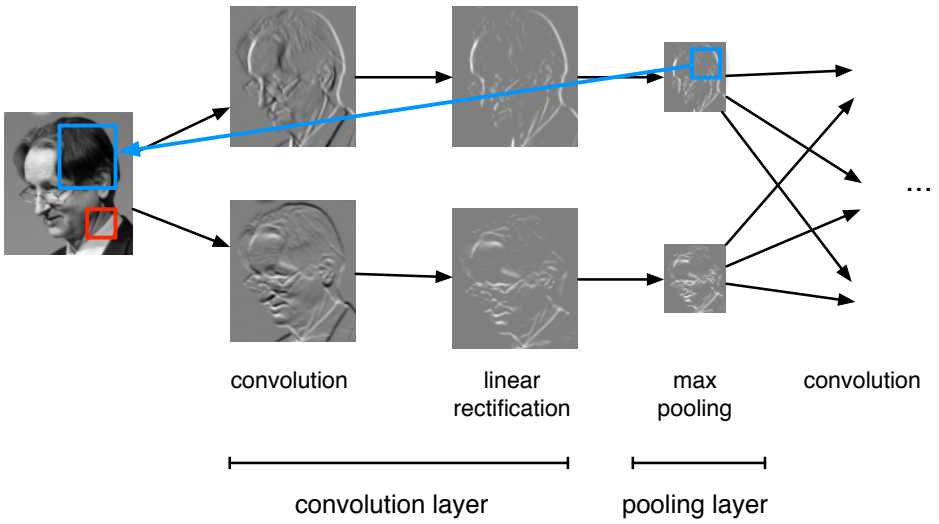


Figure 8: Schematic of a conv net with convolution and pooling layers. Pooling layers expand the receptive fields of units in subsequent convolution layers.

Lecture 10: Image Classification

Roger Grosse

1 Introduction

Vision feels so easy, since we do it all day long without thinking about it. But think about just how hard the problem is, and how amazing it is that we can see. A grayscale image is just a two dimensional array of intensity values, and somehow we can recover from that a three-dimensional understanding of a scene, including the types of objects and their locations, which particular people are present, what materials things are made of, and so on. In order to see, we have to deal with all sorts of "nuisance" factors, such as change in pose or lighting. It's amazing that the human visual system does this all so seamlessly that we don't even have to think about it.

Even talking about "images" masks a lot of complexity; the human retina has to deal with 11 orders of magnitude in intensity variation and uses fancy optics that let us recover detailed information in the fovea of our visual field, for a variety of wavelengths of light.

There is a large and active field of research called computer vision which tries to get machines to see. The field has made rapid progress in the past decade, largely because of increasing sophistication of machine learning techniques and the availability of large image collections. They've formulated hundreds of interesting visual "tasks" which encapsulate some of the hidden complexity we deal with on a daily basis, such as estimating the calorie content of a plate of food or predicting whether a structure is likely to fall down. But there's one task which has received an especially large amount of attention for the past 30 years and which has driven a lot of the progress in the field: **object recognition**, the task of classifying an image into a set of object categories.

Object recognition is also a useful example for looking at how conv nets have changed over the years, since they were a state-of-the-art tool in the early days, and in the last five years, they have re-emerged as the state-of-the-art tool for object recognition as well as dozens of other vision tasks. When conv nets took over the field of computer vision, object recognition was the first domino to fall. Computers have gotten dramatically faster during this time, and the networks have gotten correspondingly bigger and more powerful, but they're still based on more or less the same design principles. This lecture will talk about some of those design principles.

2 Object recognition datasets

Recall that object recognition is a kind of supervised learning problem, which means there's a particular behavior we would like our system to achieve (labeling an image with the correct category), and we need to provide the system with labeled examples of the correct behavior. This means we need to come up with a **dataset**, a set of images with their corresponding labels. This raises questions such as: how do we choose the set of categories? What sorts of images do we allow, how many do we need, and where do

we get them? Do we **preprocess** them in some way to make life easier for the algorithm? We'll look at just a few examples of particularly influential datasets, but we'll ignore dozens more, which each have their virtues and drawbacks.

2.1 USPS and MNIST

Before machine learning algorithms were good enough to recognize objects in images, researchers' attention focused on a simpler image classification problem: **handwritten digit recognition**. In the 1980s, the US Postal Service was interested in automatically reading zip codes on envelopes. This task is a bit harder than handwritten digit recognition, since one also has to identify the locations and orientations of the individual digits, but clearly digit recognition would be a useful step towards solving the problem. They collected a dataset of images of handwritten digits (now called the **USPS Dataset**) by hand-segmenting individual digits from handwritten zip codes. To make things easier for the algorithm, the digits were **normalized** to be a consistent size and orientation. Despite this normalization, the dataset still included a lot of sources of variability: digits were written in a variety of writing styles and using different kinds of writing instruments. Many of the digits are ambiguous, even to humans.

Classifying USPS digits became the first practical use of conv nets: in 1989, a group of researchers at Bell Labs introduced a conv net architecture, which involved several convolution and subsampling layers, followed by a fully connected layer. This network was able to classify the digits with 91.9% accuracy.

Almost a decade later, researchers created a slightly larger handwritten digit dataset. They made some modifications to a dataset produced by the National Institute of Standards and Technology, so the dataset was called Modified NIST, or **MNIST**. Similar to the USPS Dataset, MNIST images were normalized by centering the digits within the image and normalizing them to a standard size. The main difference is that the dataset is larger: there were 70,000 examples, of which 60,000 are used for training and 10,000 are used for testing. Yann LeCun and colleagues introduced a larger conv net architecture called **LeNet** which was able to classify images with 98.9% accuracy, and used this network in the context of a larger system for automatically reading the numbers on checks. (Because LeNet was trained on segmented and normalized digit images, this system had to solve the problems of automatic segmentation and normalization, among other things.) This was the first automatic check reading system that was accurate enough to be practically useful. This was one of the big success stories of AI in the 1990s — and interestingly, it happened during the "neural net winter", showing that good ideas can still work even when they fall out of fashion.[1]

LeCun is now one of the leading researchers in the field, and directs Facebook AI Research.

Apart from its initial practical uses, MNIST has served as one of the most widely used machine learning benchmarks for two decades. Even though the test errors have long been low enough to be practically meaningless, MNIST has driven a lot of progress in neural net research. As recently

[1] LeCun et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

as 2012, Geoff Hinton and collaborators introduced dropout (a regularization method discussed in Lecture 9) on MNIST; this turned out to work well on a lot of other problems, and has become one of the standard tools in the neural net toolbox.

2.2 Caltech101 and the perils of dataset design

In 2003, researchers at Caltech released the first major object recognition dataset which was used to train and benchmark object recognition algorithms. Since their dataset included 101 object categories, they called it **Caltech101**.[2] Here's how they approached some of the key questions of dataset design.

- *Which object categories to consider?* They chose a set of 101 object categories by opening a dictionary to random pages and choosing from the nouns which were associated with images.
- *Where do the images come from?* They used Google Image Search to find candidate images, and then filtered by hand which images actually represented the object category.
- *How many images?* They didn't target a particular number of objects per category, but just collected as many as possible. The numbers of objects per category were very unbalanced as a result, but in practice, when the dataset is used for benchmarking, most systems are trained with a fixed number of images per category (which is usually between 1 and 20).
- *How to normalize the images?* They normalized the images in a variety of ways to make things simpler for the learning algorithms. Images were scaled to be about 300 pixels wide. In order to reduce variability in pose, they flipped some of the images so that a given object was always facing the same direction. More controversially, images of certain object categories were rotated because the authors' proposed method had trouble dealing with vertically oriented objects.

For about 5 years, Caltech101 was widely used as a benchmark dataset for object recognition, and academic papers showed rapid improvements in classification accuracy. Unfortunately, the dataset had a number of idiosyncrasies, known as **dataset biases**. E.g., for some reason, objects always appeared at a consistent location within an image, with the result that if one averages the raw pixel values, the average image still resembles the object category.[3] Also, as mentioned above, images of certain categories were rotated, leading to distinctive rotation artifacts.

Dataset bias results in a kind of overfitting which is different from what we've talked about so far. In our lecture on generalization, we observed that a training set might happen to have certain accidental regularities which don't occur in the test set; algorithms can overfit if they exploit these regularities. If the training and test images are drawn from the same

[2] https://www.vision.caltech.edu/Image_Datasets/Caltech101/

[3] https://www.vision.caltech.edu/Image_Datasets/Caltech101/averages100objects.jpg

distribution, this kind of overfitting can be eliminated if one builds a large enough training set. Dataset bias is different — it consists of systematic biases in a dataset resulting from the way in which the data was collected. These regularities occur in both the training and the test sets, so algorithms which exploit them appear to generalize well on the test set. However, if those regularities aren't present in the situation where one actually wants to use the classifier (e.g. a robot trying to identify objects), the system will perform very poorly in practice. (If an image classifier only recognizes minarets by exploiting rotation artifacts, it's unlikely to perform very well in the real world.)

If dataset bias is strong enough, it encourages the troubling practice of **dataset hacking**, whereby researchers engineer their learning algorithms to be able to exploit the dataset biases in order to make their results seem more impressive. In the case of Caltech101, the dataset biases were strong enough that dataset hacking became essentially the only way to compete. After about 5 years, Caltech101 basically stopped being used for computer vision research. Dozens of other object recognition datasets were created, all using different methodology intended to attenuate dataset bias; see this paper[4] for an interesting discussion. Despite a lot of clever attempts, creating a fully realistic dataset is an elusive goal, and dataset bias will probably always exist to some degree.

An interesting tidbit: both human researchers and learning algorithms are able to determine with surprisingly high accuracy which object recognition dataset a given image was drawn from.

2.3 ImageNet

In 2009, taking into account lessons learned from Caltech101 and other computer vision datasets, researchers built ImageNet, a massive object recognition database consisting of millions of full-resolution images and thousands of object categories. Based on this dataset, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) became one of the most important computer vision benchmarks. Here's how they approached the same questions:

- *Which object categories to consider?* ImageNet was meant to be very comprehensive. The categories were taken from WordNet, a lexical database for English constructed by cognitive scientists at Princeton. WordNet consists of a hierarchy of "synsets", or sets of synonyms which all denote the same concept. ImageNet was intended to include as many synsets as possible; as of 2010, it included almost 22,000 synsets, out of 80,000 noun synsets in WordNet. The categories are very specific, including hundreds of different types of dogs. Out of these categories, 1000 were chosen for the ILSVRC.

- *How many images?* The aim was to come up with hundreds of labeled images for each synset. The ILSVRC categories all have hundreds of associated training examples, for a total of 1.2 million images.

- *Where do the images come from?* Similarly to Caltech101, candidate images were taken from the results of various image search engines,

[4] A. Torralba and A. Efros. An unbiased look at dataset bias. *Computer Vision and Pattern Recognition (CVPR)*, 2011.

and then humans manually labeled them. Labeling millions of images is obviously challenging, so they paid Amazon Mechanical Turk workers to annotate images. Since some of the categories were highly specific or unusual, they had to provide the annotators with additional information (e.g. Wikipedia articles) to help them, and carefully validated the process by measuring inter-annotator agreement.

- *How are the images normalized?* In contrast to Caltech101, the images in the dataset itself are not normalized. (However, the object recognition systems themselves might perform some sort of preprocessing.)

Because the object categories are so diverse and fine-grained, and images can contain multiple objects, there might not be a unique right answer for every image. Therefore, one normally reports **top-5 accuracy**, whereby the algorithm is allowed to make 5 different predictions for each image, and it gets it right if any of the 5 predictions are the correct category.

ImageNet is an extremely challenging dataset to work with because of its scale and the diversity of object categories. The first algorithms to be applied were not neural nets, but in 2012, researchers in Toronto entered this competition using a neural net called **AlexNet** (in honor of its lead creator, Alex Krizhevsky). It achieved top-5 error of 28.5%, which was substantially better than the competitors. This result created a big splash, leading computer vision researchers to switch to using neural nets and prompting some of the world's largest software companies to start up research labs focused on deep learning. Since AlexNet, error rates on ImageNet have fallen dramatically, hitting 4.5% error in 2015 (the last year the competition was run), and all of the leading approaches have been based on conv nets. This even beat human performance, which was measured at 5.1% error (although this can vary significantly depending how one measures).

3 LeNet

Let's look at a particular conv net architecture: LeNet, which was used to classify MNIST digits in 1998. The inputs are grayscale images of size 32×32. One detail I've skipped over so far is the sizes of the outputs of convolution layers. LeNet uses **valid convolutions**, where the values are computed for only those locations whose filters lie entirely within the input. Therefore, if the input is 32×32 and the filters are 5×5, the outputs will be 28×28. (The main alternative is **same convolution**, where the output is the same size as the input, and the input image is padded with zeros in all directions.) The LeNet architecture is shown in Figure 1 and summarized in Table 1.

- *Convolution layer C1.* This layer has 6 feature maps and filters of size 5×5. It has $28 \times 28 \times 6 = 4704$ units, $28 \times 28 \times 5 \times 5 \times 6 = 117,600$ connections, and $5 \times 5 \times 6 = 150$ weights and 6 biases, for a total of 156 trainable parameters.

- *Subsampling layer S2.* In LeNet, the "subsampling layers" are essentially pooling layers, where the pooling function is the mean (rather

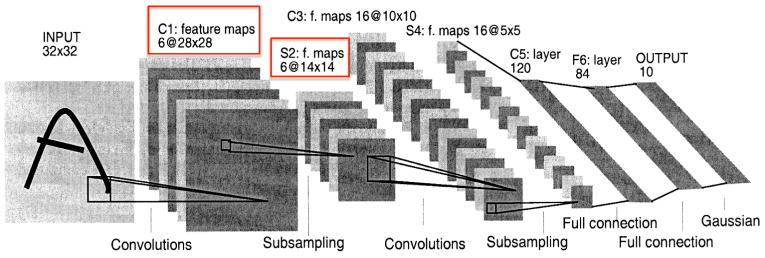


Figure 1: The LeNet architecture from 1998.

than max). They use a stride of 2, so the image size is shrunk by a factor of 2 along each dimension.

- *Convolution layer C3.* This layer has 16 feature maps of size 10×10 and filters of size 5×5. Therefore, it has $10 \times 10 \times 16 = 1600$ units. If all the feature maps were connected to all the feature maps, this layer would have $10 \times 10 \times 5 \times 5 \times 6 \times 16 = 240,000$ connections and $5 \times 5 \times 6 \times 16 = 2400$ weights.[5]

- *Subsampling layer S4.* This is another pooling layer with a stride of 2, so it reduces each dimension by another factor of 2, to 5×5.

- *Fully connected layer F5.* This layer has 120 units with a full set of connections to layer S4. Since S4 has $5 \times 5 \times 16 = 400$ units, this layer has $400 \times 120 = 48,000$ connections, and hence the same number of weights.

- *Fully connected layer F6.* This layer has 84 units, fully connected to F5. Therefore, it has $84 \times 120 = 10,080$ connections and the same number of weights.

- *Output layer.* The original network used something called radial basis functions, but for simplicity we'll pretend it's just a linear function, followed by a softmax over 10 categories. It has $84 \times 10 = 840$ connections and weights.

These calculations are all summarized in Table 1. After sitting through all this tedium, we can draw a number of useful conclusions:

- Most of the units are in the first convolution layer.

- Most of the connections are in the second convolution layer.

- Most of the weights are in the fully connected layers.

These observations correspond to various resource limitations when designing a network architecture. In particular, if we want to make the network as big as possible, here are some of the limitations we run into:

[5] Since this would have been a lot of connections by the standards of 1998, they skimped on connections by connecting only a subset of the feature maps. This brought the number of connections down to 156,000.

Layer	Type	# units	# connections	# weights
C1	convolution	4704	117,600	150
S2	subsampling	1176	4704	0
C3	convolution	1600	240,000	2400
S4	subsampling	400	1600	0
F5	fully connected	120	48,000	48,000
F6	fully connected	84	10,080	10,080
output	fully connected	10	840	840

Table 1: LeNet architecture, with the sizes of layers.

- Running the network to compute predictions (equivalently, the forward pass of backprop) requires approximately one add-multiply operation per connection in the network. As observed in a previous lecture, the backwards pass is about as expensive as two forward passes, so the total computational cost of backprop is proportional to the number of connections. This means the convolution layers are generally the most expensive part of the network in terms of running time.
- Memory is another scarce resource. It's worth distinguishing two situations: **training time**, where we train the network using backprop, and **test time**, the somewhat misleading name for the setting where we use an already-trained network.
  - Backprop requires storing all of the activations in memory.[6] Since the number of activations is the number of units times the mini-batch size, the number of units determines the memory footprint of the activations at training time. The activations don't need to be stored at test time.
  - The weights also need to be stored in memory, both at training time and test time.
- The weights constitute the vast majority of trainable parameters of the model (the number of biases generally being far smaller), so if you're worried about overfitting, you could consider cutting down the number of weights.

LeNet was carefully designed to push the limits of all of these resource constraints using the computing power of 1998. As we'll see, conv nets have grown substantially larger in order to exploit modern computing resources.

Try increasing the sizes of various layers and checking that you're substantially increasing the usage of one or more of these resources.

4 Modern conv nets

As mentioned above, AlexNet was the conv net architecture which started a revolution in computer vision by smashing the ILSVRC benchmark. This

[6] This isn't quite true, actually. There are tricks for storing activations for only a subset of the layers, and recomputing the rest of the activations as needed. Indeed, frameworks like TensorFlow implement this behind the scenes. However, a larger of units generally implies a higher memory footprint.

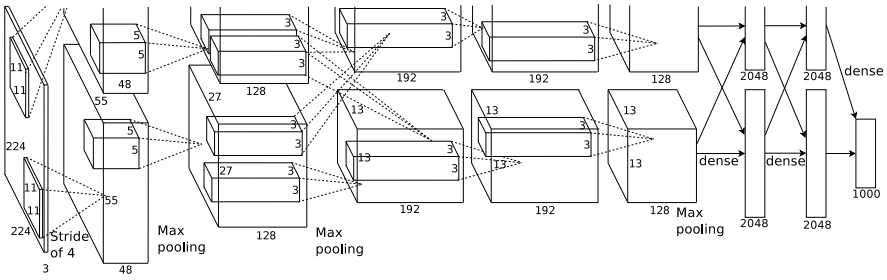


Figure 2: The AlexNet architecture from 2012.

	LeNet (1989)	**LeNet (1998)**	**AlexNet (2012)**
classification task	digits	digits	objects
dataset	USPS	MNIST	ImageNet
# categories	10	10	1,000
image size	16×16	28×28	$256 \times 256 \times 3$
training examples	7,291	60,000	1.2 million
units	1,256	8,084	658,000
parameters	9,760	60,000	60 million
connections	65,000	344,000	652 million
total operations	11 billion	412 billion	200 quadrillion (est.)

Table 2: Comparison of conv net classification architectures.

architecture is shown in Figure 2. Like LeNet, it consists mostly of convolution, pooling, and fully connected layers. It additionally has some "response normalization" layers, which I won't talk about because they're not believed to make a big difference, and have mostly stopped being used.

By most measures, AlexNet is 100 to 1000 times bigger than LeNet, as shown in Table 2. But qualitatively, the structure is very similar to LeNet: it consists of alternating convolution and pooling layers, followed by fully connected layers. Furthermore, like LeNet, most of the units and connections are in the convolution layers, and most of the weights are in the fully connected layers.

Computers have improved a lot since LeNet, but the hardware advance that suddenly made it practical to train large neural nets was **graphics processing units (GPUs)**. GPUs are a kind of processor geared towards highly parallel processing involving relatively simple operations. One of the things they especially excel at is matrix multiplication. Since most of the running time for a neural net consists of matrix multiplication (even convolutions are implemented as matrix products beneath the hood), GPUs gave roughly a 30-fold speedup in practice for training neural nets.

AlexNet set the agenda for object recognition research ever since. In 2013, the ILSVRC winner was based on tweaks to AlexNet. In 2014, the second place entry was **VGGNet**, another conv net based on more or less similar principles.

The winning entry for 2014, **GoogLeNet**, or **Inception**, deserves mention. As the name suggests, it was designed by researchers at Google. The

architecture is shown in Figure 3. Clearly things have gotten more complicated since the days of LeNet. But the main point of interest is that they went out of their way to reduce the number of trainable parameters (weights) from AlexNet's 60 million, to about 2 million. Why? Partly it was to reduce overfitting — amazingly, it's possible to overfit a million images if you have a big enough network like AlexNet.

The other reason has to do with saving memory at "test time", i.e. when the network is being used. Traditionally, networks would be both trained and run on a single PC, so there wasn't much reason to draw a distinction between training and test time. But at Google, the training could be distributed over lots of machines in a datacenter. (The activations and parameters could even be divided up between multiple machines, increasing the amount of available memory at training time.) But the network was also supposed to be runnable on an Android cell phone, so that images wouldn't have to be sent to Google's servers for classification. On a cell phone, it would have been extravagant to spend 240MB to store AlexNet's 60 million parameters, so it was really important to cut down on parameters to make it fit in memory.

They achieved this in two ways. First, they eliminated the fully connected layers, which we already saw contain most of the parameters in LeNet and AlexNet. GoogLeNet is convolutions all the way. It also avoids having large convolutions by breaking them down into a sequence of convolutions involving smaller filters. (Two 3×3 filters have fewer parameters than a 5×5 filter, even though they cover a similar radius of the image.) They call this layer-within-a-layer architecture "Inception", after the movie about dreams-within-dreams.

This is analogous to how linear bottleneck layers can reduce the number of parameters.

Performance on ImageNet improved asonishingly fast during the years the competition was run. Here are the figures:

Year	**Model**	**Top-5 error**
2010	Hand-designed descriptors + SVM	28.2%
2011	Compressed Fisher Vectors + SVM	25.8%
2012	AlexNet	16.4%
2013	a variant of AlexNet	11.7%
2014	GoogLeNet	6.6%
2015	deep residual nets	4.5%

We'll put off the last item, deep residual nets (ResNets), until a later lecture since they depend on some ideas that we won't cover until we talk about RNNs.

It's really unusual for error rates to drop by a factor of 6 over a period of 5 years, especially on a task like object recognition that hundreds of researchers had already worked hard on and where performance had seemed to plateau.

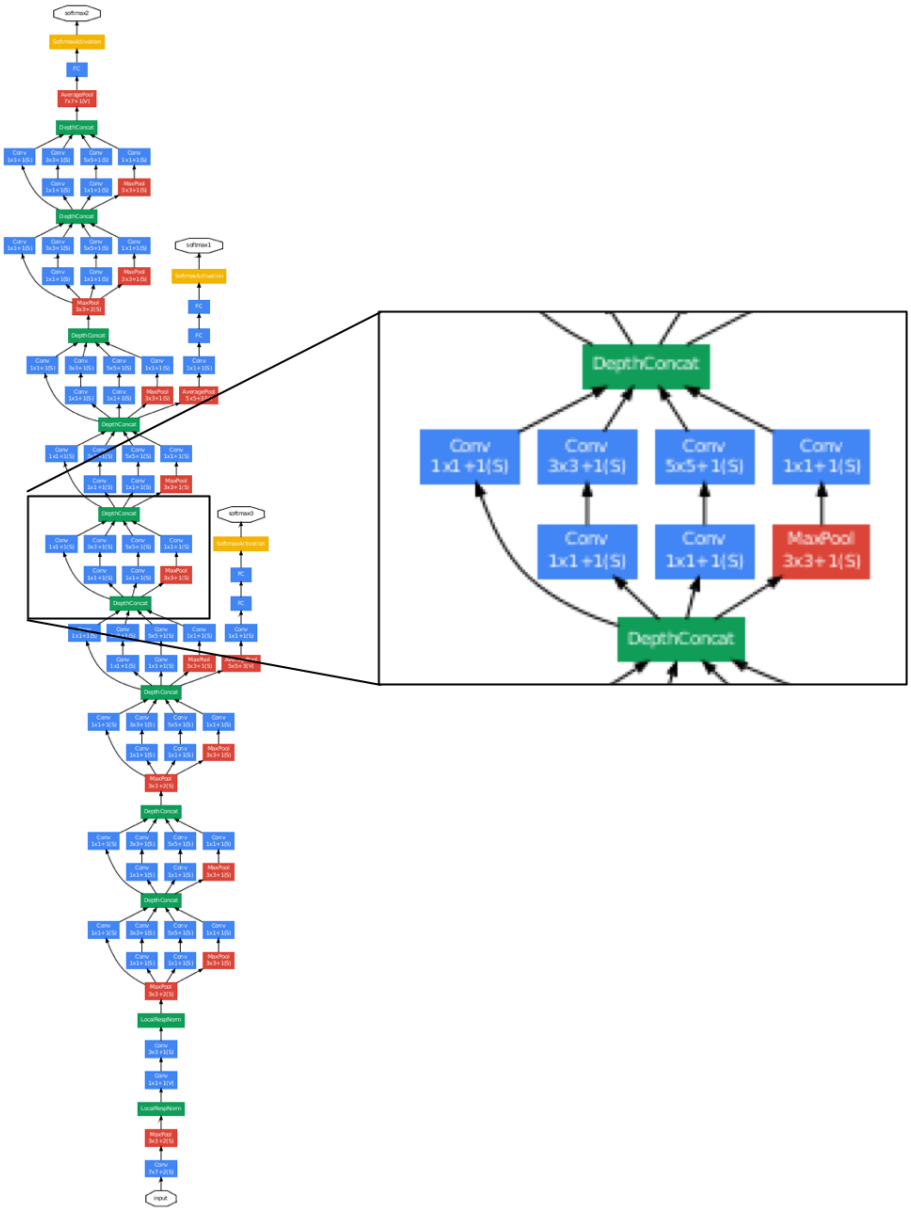


Figure 3: The Inception architecture from 2014.

Lecture 12: Generalization

Roger Grosse

1 Introduction

When we train a machine learning model, we don't just want it to learn to model the training data. We want it to *generalize* to data it hasn't seen before. Fortunately, there's a very convenient way to measure an algorithm's generalization performance: we measure its performance on a held-out test set, consisting of examples it hasn't seen before. If an algorithm works well on the training set but fails to generalize, we say it is *overfitting*. Improving generalization (or preventing overfitting) in neural nets is still somewhat of a dark art, but this lecture will cover a few simple strategies that can often help a lot.

1.1 Learning Goals

- Know the difference between a training set, validation set, and test set.
- Be able to reason qualitatively about how training and test error depend on the size of the model, the number of training examples, and the number of training iterations.
- Understand the motivation behind, and be able to use, several strategies to improve generalization:
  - reducing the capacity
  - early stopping
  - weight decay
  - ensembles
  - input transformations
  - stochastic regularization

2 Measuring generalization

So far in this course, we've focused on training, or optimizing, neural networks. We defined a cost function, the average loss over the training set:

$$\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}), t^{(i)}). \tag{1}$$

But we don't just want the network to get the training examples right; we also want it to generalize to novel instances it hasn't seen before.

Fortunately, there's an easy way to measure a network's generalization performance. We simply partition our data into three subsets:

- A **training set**, a set of training examples the network is trained on.
- A **validation set**, which is used to tune hyperparameters such as the number of hidden units, or the learning rate.
- A **test set**, which is used to measure the generalization performance.

There are lots of variants on this basic strategy, including something called cross-validation. Typically, these alternatives are used in situations with small datasets, i.e. less than a few thousand examples. Most applications of neural nets involve datasets large enough to split into training, validation and test sets.

The losses on these subsets are called **training, validation, and test loss**, respectively. Hopefully it's clear why we need separate training and test sets: if we train on the test data, we have no idea whether the network is correctly generalizing, or whether it's simply memorizing the training examples. It's a more subtle point why we need a separate validation set.

- We can't tune hyperparameters on the training set, because we want to choose values that will generalize. For instance, suppose we're trying to choose the number of hidden units. If we choose a very large value, the network will be able to memorize the training data, but will generalize poorly. Tuning on the training data could lead us to choose such a large value.
- We also can't tune them on the test set, because that would be "cheating." We're only allowed to use the test set once, to report the final performance. If we "peek" at the test data by using it to tune hyperparameters, it will no longer give a realistic estimate of generalization performance.[1]

The most basic strategy for tuning hyperparameters is to do a **grid search**: for each hyperparameter, choose a set of candidate values. Separately train models using all possible combinations of these values, and choose whichever configuration gives the best validation error. A closely related alternative is **random search**: train a bunch of networks using random configurations of the hyperparameters, and pick whichever one has the best validation error. The advantage of random search over grid search is as follows: suppose your model has 10 hyperparameters, but only two of them are actually important. (You don't know which two.) It's infeasible to do a grid search in 10 dimensions, but random search still ought to provide reasonable coverage of the 2-dimensional space of the important hyperparameters. On the other hand, in a scientific setting, grid search has the advantage that it's easy to reproduce the exact experimental setup.

3 Reasoning about generalization

If a network performs well on the training set but generalizes badly, we say it is **overfitting**. A network might overfit if the training set contains **accidental regularities**. For instance, if the task is to classify handwritten digits, it might happen that in the training set, all images of 9's have pixel number 122 on, while all other examples have it off. The network might

[1] Actually, there's some fascinating recent work showing that it's possible to use a test set repeatedly, as long as you add small amounts of noise to the average error. This hasn't yet become a standard technique, but it may sometime in the future. See Dwork et al., 2015, "The reusable holdout: preserving validity in adaptive data analysis."

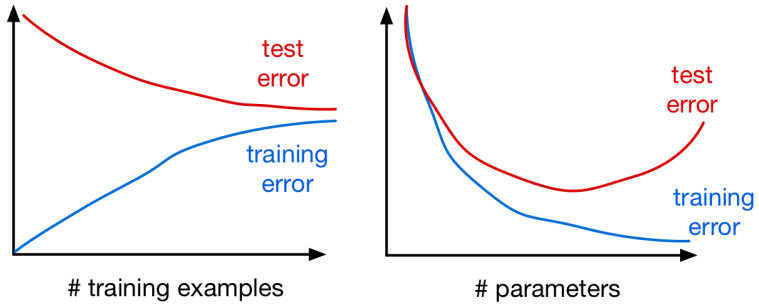


Figure 1: **(left)** Qualitative relationship between the number of training examples and training and test error. **(right)** Qualitative relationship between the number of parameters (or model capacity) and training and test error.

decide to exploit this accidental regularity, thereby correctly classifying all the training examples of 9's, without learning the true regularities. If this property doesn't hold on the test set, the network will generalize badly.

As an extreme case, remember the network we constructed in Lecture 5, which was able to learn arbitrary Boolean functions? It had a separate hidden unit for every possible input configuration. This network architecture is able to **memorize** a training set, i.e. learn the correct answer for every training example, even though it will have no idea how to classify novel instances. The problem is that this network has too large a **capacity**, i.e. ability to remember information about its training data. Capacity isn't a formal term, but corresponds roughly to the number of trainable parameters (i.e. weights). The idea is that information is stored in the network's trainable parameters, so networks with more parameters can store more information.

In order to reason qualitatively about generalization, let's think about how the training and generalization error vary as a function of the number of training examples and the number of parameters. Having more training data should only help generalization: for any particular test example, the larger the training set, the more likely there will be a closely related training example. Also, the larger the training set, the fewer the accidental regularities, so the network will be forced to pick up the true regularities. Therefore, generalization error ought to improve as we add more training examples. On the other hand, small training sets are easier to memorize than large ones, so training error tends to increase as we add more examples. As the training set gets larger, the two will eventually meet. This is shown qualitatively in Figure 1.

If the test error *increases* with the number of training examples, that's a sign that you have a bug in your code or that there's something wrong with your model.

Now let's think about the model capacity. As we add more parameters, it becomes easier to fit both the accidental and the true regularities of the training data. Therefore, training error improves as we add more parameters. The effect on generalization error is a bit more subtle. If the network has too little capacity, it generalizes badly because it fails to pick up the regularities (true or accidental) in the data. If it has too much capacity, it will memorize the training set and fail to generalize. Therefore, the effect

of capacity on test error is non-monotonic: it decreases, and then increases. We would like to design network architectures which have enough capacity to learn the true regularities in the training data, but not enough capacity to simply memorize the training set or exploit accidental regularities. This is shown qualitatively in Figure 1.

3.1 Bias and variance

For now, let's focus on squared error loss. We'd like to mathematically model the generalization error of the classifier, i.e. the expected error on examples it hasn't seen before. To formalize this, we need to introduce the **data generating distribution**, a hypothetical distribution $p_{\mathcal{D}}(\mathbf{x}, t)$ that all the training and test data are assumed to have come from. We don't need to assume anything about the form of the distribution, so the only nontrivial assumption we're making here is that the training and test data are drawn from the same distribution.

Suppose we have a test input $\mathbf{x}$, and we make a prediction y (which, for now, we treat as arbitrary). We're interested in the expected error if the targets are sampled from the conditional distribution $p_{\mathcal{D}}(t \mid \mathbf{x})$. By applying the properties of expectation and variance, we can decompose this expectation into two terms:

This derivation makes use of the formula $\mathrm{Var}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$ for a random variable z.

$$\begin{aligned}
\mathbb{E}[(y-t)^2 \mid \mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2 \mid \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t^2 \mid \mathbf{x}] && \text{by linearity of expectation} \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t \mid \mathbf{x}]^2 + \mathrm{Var}[t \mid \mathbf{x}] && \text{by the formula for variance} \\
&= (y - \mathbb{E}[t \mid \mathbf{x}])^2 + \mathrm{Var}[t \mid \mathbf{x}] \\
&\triangleq (y - y_\star)^2 + \mathrm{Var}[t \mid \mathbf{x}],
\end{aligned}$$

where in the last step we introduce $y_\star = \mathbb{E}[t \mid \mathbf{x}]$, which is the best possible prediction we can make, because the first term is nonnegative and the second term doesn't depend on y. The second term is known as the **Bayes error**, and corresponds to the best possible generalization error we can achieve even if we model the data perfectly.

Now let's treat y as a random variable. Assume we repeat the following experiment: sample a training set randomly from $p_{\mathcal{D}}$, train our network, and compute its predictions on $\mathbf{x}$. If we suppress the dependence on $\mathbf{x}$ for simplicity, the expected squared error decomposes as:

$$\begin{aligned}
\mathbb{E}[(y-t)^2] &= \mathbb{E}[(y - y_\star)^2] + \mathrm{Var}(t) \\
&= \mathbb{E}[y_\star^2 - 2y_\star y + y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y^2] + \mathrm{Var}(t) && \text{by linearity of expectation} \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}(y) + \mathrm{Var}(t) && \text{by the formula for variance} \\
&= \underbrace{(y_\star - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\mathrm{Var}(y)}_{\text{variance}} + \underbrace{\mathrm{Var}(t)}_{\text{Bayes error}}
\end{aligned}$$

The first term is the **bias**, which tells us how far off the model's average prediction is. The second term is the **variance**, which tells us about the variability in its predictions as a result of the choice of training set, i.e. the

amount to which it overfits the idiosyncrasies of the training data. The third term is the Bayes error, which we have no control over. So this decomposition is known as the **bias-variance decomposition**.

To visualize this, suppose we have two test examples, with targets $(t^{(1)}, t^{(2)})$. Figure 2 is a visualization in **output space**, where the axes correspond to the outputs of the network on these two examples. It shows the test error as a function of the predictions on these two test examples; because we're measuring mean squared error, the test error takes the shape of a quadratic bowl. The various quantities computed above can be seen in the diagram:

Understand why output space is different from input space or weight space.

- The generalization error is the average squared length $\|\mathbf{y} - \mathbf{t}\|^2$ of the line segment labeled *residual*.
- The bias term is the average squared length $\|\mathbb{E}[\mathbf{y}] - \mathbf{y}_*\|^2$ of the line segment labeled *bias*.
- The variance term is the spread in the green x's.
- The Bayes error is the spread in the black x's.

4 Reducing overfitting

Now that we've talked about generalization error and how to measure it, let's see how we can improve generalization by reducing overfitting. Notice that I said *reduce*, rather than *eliminate*, overfitting. Good models will probably still overfit at least a little bit, and if we try to eliminate overfitting, i.e. eliminate the gap between training and test error, we'll probably cripple our model so that it doesn't learn anything at all. Improving generalization is somewhat of a dark art, and there are very few techniques which both work well in practice and have rigorous theoretical justifications. In this section, I'll outline a few tricks that seem to help a lot. In practice, most good neural networks combine several of these tricks. Unfortunately, for the most part, these intuitive justifications are hard to translate into rigorous guarantees.

4.1 Reducing capacity

Remember the nonmonotonic relationship between model capacity and generalization error from Figure 1? This immediately suggests a strategy: there are various hyperparameters which affect the capacity of a network, such as the number of layers, or the number of units per layer. We can tune these parameters on a validation set in order to find the sweet spot, which has enough capacity to learn the true regularities, but not enough to overfit. (We can do this tuning with grid search or random search, as described above.)

A network with L layers and H units per layer will have roughly LH^2 weights. Think about why this is.

Besides reducing the number of layers or the number of units per layer, another strategy is to reduce the number of parameters by adding a **bottleneck layer**. This is a layer with fewer units than the layers below or above it. As shown in Figure 3, this can reduce the total number of connections, and hence the number of parameters.

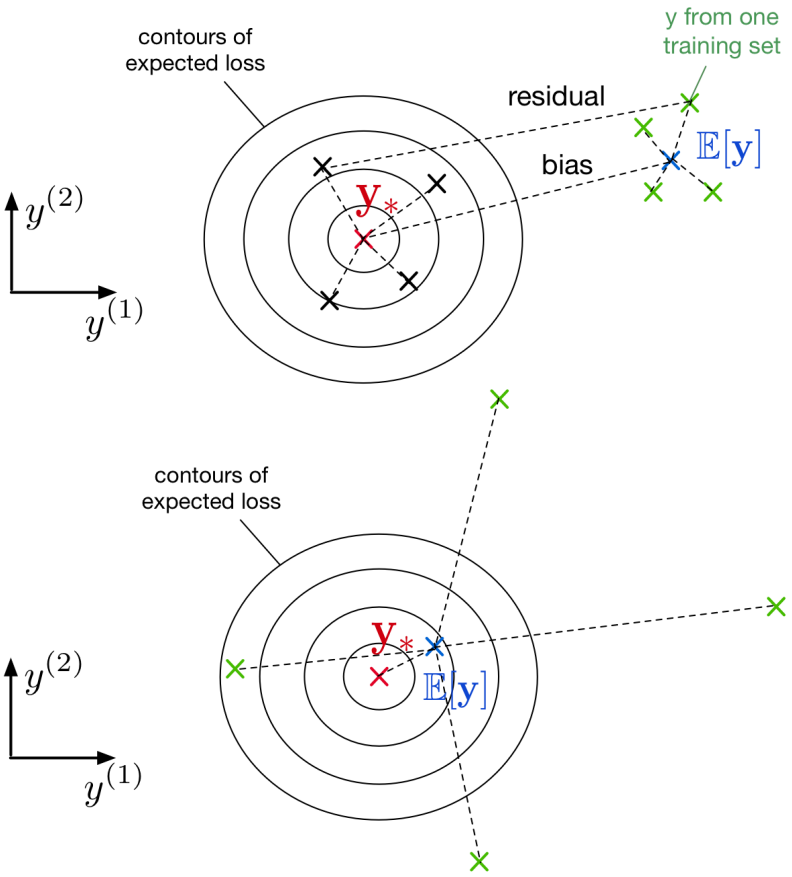


Figure 2: Schematic relating bias, variance, and error. **Top:** If the model is underfitting, the bias will be large, but the variance (spread of the green x's) will be small. **Bottom:** If the model is overfitting, the bias will be small, but the variance will be large.

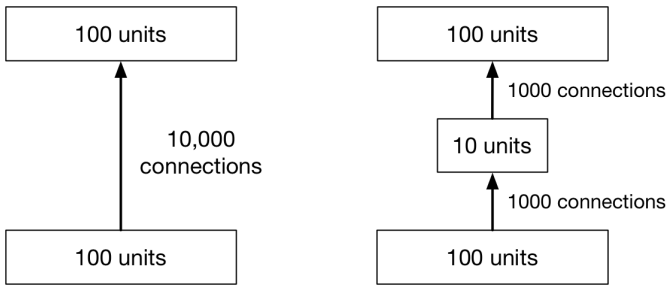


Figure 3: An example of reducing the number of parameters by inserting a linear bottleneck layer.

In general, linear and nonlinear layers have different uses. Recall that adding nonlinear layers can increase the expressive power of a network architecture, i.e. broaden the set of functions it's able to represent. By contrast, adding linear layers can't increase the expressivity, because the same function can be represented by a single layer. For instance, in Figure 3, the left-hand network can represent all the same functions as the right-hand one, since one can set $\tilde{\mathbf{W}} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}$; it can also represent some functions that the right-hand one can't. The main use of linear layers, therefore, is for bottlenecks. One benefit is to reduce the number of parameters, as described above. Bottlenecks are also useful for another reason which we'll talk about later on, when we discuss autoencoders.

Reducing capacity has an important drawback: it might make the network too simple to learn the true regularities in the data. Therefore, it's often preferable to keep the capacity high, but prevent it from overfitting in other ways. We'll discuss some such alternatives now.

4.2 Early stopping

Think about how the training and test error change over the course of training. Clearly, the training error ought to continue improving, since we're optimizing the training error. (If you find the training error going up, there may be something wrong with your optimizer.) The test error generally improves at first, but it may eventually start to increase as the network starts to overfit. Such a pattern is shown in Figure 4. (Curves such as these are referred to as **training curves**.) This suggests an obvious strategy: stop the training at the point where the generalization error starts to increase. This strategy is known as **early stopping**. Of course, we can't do early stopping using the test set, because that would be cheating. Instead, we would determine when to stop by monitoring the validation error during training.

Unfortunately, implementing early stopping is a bit harder than it looks from this cartoon picture. The reason is that the training and validation error fluctuate during training (because of stochasticity in the gradients), so it can be hard to tell whether an increase is simply due to these fluctuations. One common heuristic is to space the validation error measurements far apart, e.g. once per epoch. If the validation error fails to improve after one epoch (or perhaps after several consecutive epochs), then we stop training. This heuristic isn't perfect, and if we're not careful, we might stop training too early.

4.3 Regularization and weight decay

So far, all of the cost functions we've discussed have consisted of the average of some loss function over the training set. Often, we want to add another term, called a **regularization term**, or **regularizer**, which penalizes hypotheses we think are somehow pathological and unlikely to generalize well.

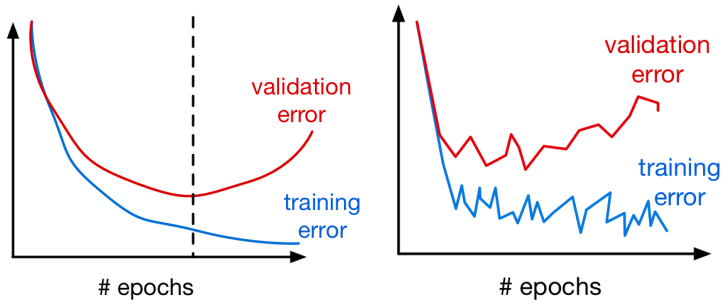


Figure 4: Training curves, showing the relationship between the number of training iterations and the training and test error. **(left)** Idealized version. **(right)** Accounting for fluctuations in the error, caused by stochasticity in the SGD updates.

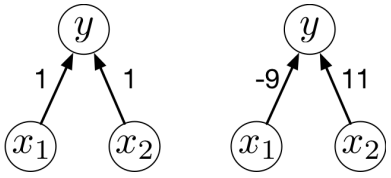


Figure 5: Two sets of weights which make the same predictions assuming inputs x_1 and x_2 are identical.

The total cost, then, is

$$\mathcal{J}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x},\boldsymbol{\theta}),t)}_{\text{training loss}} + \underbrace{\mathcal{R}(\boldsymbol{\theta})}_{\text{regularizer}} \tag{2}$$

For instance, suppose we are training a linear regression model with two inputs, x_1 and x_2, and these inputs are identical in the training set. The two sets of weights shown in Figure 5 will make identical predictions on the training set, so they are equivalent from the standpoint of minimizing the loss. However, Hypothesis A is somehow better, because we would expect it to be more stable if the data distribution changes. E.g., suppose we observe the input $(x_1 = 1, x_2 = 0)$ on the test set; in this case, Hypothesis A will predict 1, while Hypothesis B will predict -8. The former is probably more sensible. We would like a regularizer to favor Hypothesis A by assigning it a smaller penalty.

One such regularizer which achieves this is L_2 **regularization**; for a linear model, it is defined as follows:

This is an abuse of terminology; mathematically speaking, this really corresponds to the *squared* L_2 norm.

$$\mathcal{R}_{L_2}(\mathbf{w}) = \frac{\lambda}{2}\sum_{j=1}^{D} w_j^2. \tag{3}$$

(The hyperparameter λ is sometimes called the **weight cost**.) L_2 regularization tends to favor hypotheses where the norms of the weights are

smaller. For instance, in the above example, with $\lambda = 1$, it assigns a penalty of $\frac{1}{2}(1^2+1^2) = 1$ to Hypothesis A and $\frac{1}{2}((-8)^2+10^2) = 82$ to Hypothesis B, so it strongly prefers Hypothesis A. Because the cost function includes both the training loss and the regularizer, the training algorithm is encouraged to find a compromise between the fit to the training data and the norms of the weights. L_2 regularization can be generalized to neural nets in the obvious way: penalize the sum of squares of all the weights in all layers of the network.

It's pretty straightforward to incorporate regularizers into the stochastic gradient descent computations. In particular, by linearity of derivatives,

$$\frac{\partial \mathcal{J}}{\partial \theta_j} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j}. \tag{4}$$

If we derive the SGD update in the case of L_2 regularization, we get an interesting interpretation.

Observe that in SGD, the regularizer derivatives do not need to be estimated stochastically.

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial \mathcal{J}^{(i)}}{\partial \theta_j} \tag{5}$$

$$= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j} \right) \tag{6}$$

$$= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \lambda \theta_j \right) \tag{7}$$

$$= (1 - \alpha\lambda)\theta_j - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}. \tag{8}$$

In each iteration, we shrink the weights by a factor of $1 - \alpha\lambda$. For this reason, L_2 regularization is also known as **weight decay**.

Regularization is one of the most fundamental concepts in machine learning, and tons of theoretical justifications have been proposed. Regularizers are sometimes viewed as penalizing the "complexity" of a network, or favoring explanations which are "more likely." One can formalize these viewpoints in some idealized settings. However, these explanations are very difficult to make precise in the setting of neural nets, and they don't explain a lot of the phenomena we observe in practice. For these reasons, I won't attempt to justify weight decay beyond the explanation I just provided.

4.4 Ensembles

Think back to Figure 2. If you average the predictions of multiple networks trained independently on separate training sets, this reduces the variance of the predictions, which can lead to lower loss. Of course, we can't actually carry out the hypothetical procedure of sampling training sets independently (otherwise we're probably better off combining them into one big training set). We could try to train a bunch of networks on the *same* training set starting from different initializations, but their predictions might be too similar to get much benefit from averaging. However, we can try to simulate the effect of independent training sets by somehow injecting variability into the training procedure. Here some ways of injecting variability:

- Train on random subsets of the full training data. This procedure is known as **bagging**.
- Train networks with different architectures (e.g. different numbers of layers or units, or different choice of activation function).
- Use entirely different models or learning algorithms.

The set of trained models whose predictions we're combining is known as an **ensemble**. Ensembles of networks often generalize quite a bit better than single networks. This benefit is significant enough that the winning entries for most of the major machine learning competitions (e.g. ImageNet, Netflix, etc.) used ensembles.

It's possible to prove that ensembles outperform individual networks in the case of convex loss functions. In particular, suppose the loss function $\mathcal{L}$ is convex as a function of the outputs $\mathbf{y}$. Then, by the definition of convexity,

This isn't the same as the cost being convex as a function of $\boldsymbol{\theta}$, which we saw can't happen for MLPs. Lots of loss functions are convex with respect to $\mathbf{y}$, such as squared error or cross-entropy.

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1. \tag{9}$$

Hence, the average of the predictions must beat the average losses of the individual predictions. Note that this is true regardless of where the $\mathbf{y}$s came from. They could be outputs of different neural networks, or completely different learning algorithms, or even numbers you pulled out of a hat. The guarantee doesn't hold for non-convex cost functions (such as error rate), but ensembles still tend to be very effective in practice.

This result is closely related to the Rao-Blackwell theorem from statistics.

4.5 Data augmentation

Another trick is to artificially augment the training set by introducing distortions into the inputs, a procedure known as **data augmentation**. This is most commonly used in vision applications. Suppose we're trying to classify images of objects, or of handwritten digits. Each time we visit a training example, we can randomly distort it, for instance by shifting it by a few pixels, adding noise, rotating it slightly, or applying some sort of warping. This can increase the effective size of the training set, and make it more likely that any given test example has a closely related training example. Note that the class of useful transformations will depend on the task; for instance, in object recognition, it might be advantageous to flip images horizontally, whereas this wouldn't make sense in the case of handwritten digit classification.

4.6 Stochastic regularization

One of the biggest advances in neural networks in the past few years is the use of stochasticity to improve generalization. So far, all of the network architectures we've looked at compute functions deterministically. But by injecting some stochasticity into the computations, we can sometimes prevent certain pathological behaviors and make it hard for the network to overfit. We tend to call this **stochastic regularization**, even though it doesn't correspond to adding a regularization term to the cost function.

The most popular form of stochastic regularization is **dropout**. The algorithm itself is simple: we drop out each individual unit with some probability ρ (usually $\rho = 1/2$) by setting its activation to zero. We can represent this in terms of multiplying the activations by a **mask variable** m_i, which randomly takes the values 0 or 1:

$$h_i = m_i \cdot \phi(z^{(i)}). \tag{10}$$

We derive the backprop equations in the usual way:

$$\overline{z^{(i)}} = \overline{h_i} \cdot \frac{\mathrm{d}h_i}{\mathrm{d}z^{(i)}} \tag{11}$$

$$= \overline{h_i} \cdot m_i \cdot \phi'(z^{(i)}) \tag{12}$$

Why does dropout help? Think back to Figure 5, where we had two different sets of weights which make the same predictions if inputs x_1 and x_2 are always identical. We saw that L_2 regularization strongly prefers A over B. Dropout has the same preference. Suppose we drop out each of the inputs with 1/2 probability. B's predictions will vary wildly, causing it to get much higher error on the training set. Thus, it can achieve some of the same benefits that L_2 regularization is intended to achieve.

One important point: while stochasticity is helpful in preventing overfitting, we don't want to make predictions stochastically at test time. One naïve approach would be to simply not use dropout at test time. Unfortunately, this would mean that all the units receive twice as many incoming signals as they do during training time, so their responses will be very different. Therefore, at test time, we compensate for this by multiplying the values of the weights by $1 - \rho$. You'll see an interesting interpretation of this in Homework 4.

In a few short years, dropout has become part of the standard toolbox for neural net training, and can give a significant performance boost, even if one is already using the other techniques described above. Other stochastic regularizers have also been proposed; notably batch normalization, a method we already mentioned in the context of optimization, but which has also been shown to have some regularization benefits. It's also been observed that the stochasticity in stochastic gradient descent (which is normally considered a drawback) can itself serve as a regularizer. The details of stochastic regularization are still poorly understood, but it seems likely that it will continue to be a useful technique.

Lecture 13: Recurrent Neural Nets

Roger Grosse

1 Introduction

Most of the prediction tasks we've looked at have involved pretty simple kinds of outputs, such as real values or discrete categories. But much of the time, we're interested in predicting more complex structures, such as images or sequences. The next three lectures are about producing sequences; we'll get to producing images later in the course. If the inputs and outputs are both sequences, we refer to this as **sequence-to-sequence prediction**. Here are a few examples of sequence prediction tasks:

- As we discussed in Lecture 10, language modeling is the task of modeling the distribution over English text. This isn't really a prediction task, since the model receives no input. But the output is a document, which is a sequence of words or characters.
- In speech-to-text, we'd like take an audio waveform of human speech and output the text that was spoken. In text-to-speech, we'd like to do the reverse.
- In caption generation, we get an image as input, and would like to produce a natural language description of the image.
- Machine translation is an especially important example of sequence-to-sequence prediction. We receive a sentence in one language (e.g. English) and would like to produce an equivalent sentence in another language (e.g. French).

We've already seen one architecture which generate sequences: the neural language model. Recall that we used the chain rule of conditional probability to decompose the probability of a sentence:

$$p(w_1, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \,|\, w_1, \ldots, w_{t-1}), \tag{1}$$

and then made a Markov assumption so that we could focus on a short time window:

$$p(w_t \,|\, w_1, \ldots, w_{t-1}) = p(w_t \,|\, w_{t-K}, \ldots, w_{t-1}), \tag{2}$$

where K is the context length. This means the neural language model is **memoryless**: its predictions don't depend on anything before the context window. But sometimes long-term dependencies can be important.

Figure 1 shows a neural language model with context length 1 being used to generate a sentence. Let's say we modify the architecture slightly by adding connections between the hidden units. This gives it a long-term

For a neural language model, each set of hidden units would usually receive connections from the last K inputs, for $K > 1$. For RNNs, usually it only has connections from the current input. Why?

Figure 1: **Left:** A neural language model with context length of 1. **Right:** Turning this into a recurrent neural net by adding connections between the hidden units. Note that information can pass through the hidden units, allowing it to model long-distance dependencies.

memory: information about the first word can flow through the hidden units to affect the predictions about later words in the sentence. Such an architecture is called a **recurrent neural network (RNN)**. This seems like a simple change, but actually it makes the architecture much more powerful. RNNs are widely used today both in academia and in the technology industry; the state-of-the-art systems for all of the sequence prediction tasks listed above use RNNs.

1.1 Learning Goals

- Know what differentiates RNNs from multilayer perceptrons and memoryless models.
- Be able to design RNNs by hand to perform simple computations.
- Know how to compute the loss derivatives for an RNN using backprop through time.
- Know how RNN architectures can be applied to sequence prediction problems such as language modeling and machine translation.

2 Recurrent Neural Nets

We've already talked about RNNs as a kind of architecture which has a set of hidden units replicated at each time step, and connections between them. But we can alternatively look at RNNs as dynamical systems, i.e. systems which change over time. In this view, there's just a single set of input units, hidden units, and output units, and the hidden units feed into themselves. This means the graph of an RNN may have **self-loops**; this is in contrast to the graphs for feed-forward neural nets, which must be directed acyclic graphs (DAGs). What these self-loops really mean is that the values of the hidden units at one time step depend on their values at the previous time step.

We can understand more precisely the computation the RNN is performing by **unrolling** the network, i.e. explicitly representing the various units at all time steps, as well as the connections between them. For a given sequence length, the unrolled network is essentially just a feed-forward neural net, although the weights are shared between all time steps. See Figure 2 for an example.

The trainable parameters for an RNN include the weights and biases for all of the layers; these are replicated at every time step. In addition, we

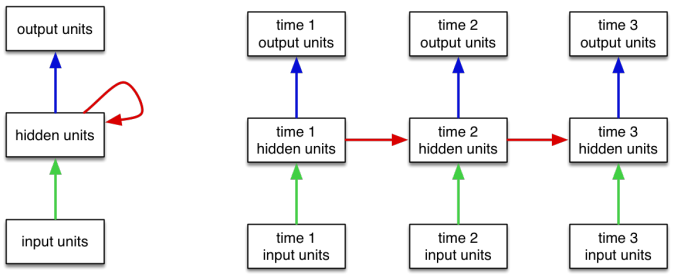


Figure 2: An example of an RNN and its unrolled representation. Note that each color corresponds to a weight matrix which is replicated at all time steps.

need some extra parameters to get the whole thing started, i.e. determine the values of the hidden units at the first time step. We can do this one of two ways:

- We can learn a separate set of biases for the hidden units in the first time step.
- We can start with a dummy time step which receives no inputs. We would then learn the initial values of the hidden units, i.e. their values during the dummy time step.

Really, these two approaches aren't very different. The signal from the $t = 0$ hiddens to the $t = 1$ hiddens is always the same, so we can just learn a set of biases which do the same thing.

Let's look at some simple examples of RNNs.

Example 1. *Figure 3 shows an example of an RNN which sums its inputs over time. All of the units are linear. Let's look at each of the three weights:*

- *The hidden-to-output weight is 1, which means the output unit just copies the hidden activation.*
- *The hidden-to-hidden weight is 1, which means that in the absence of any input, the hidden unit just remembers its previous value.*
- *The input-to-hidden weight is 1, which means the input gets added to the hidden activation in every time step.*

Example 2. *Figure 3 shows a slightly different RNN which receives two inputs at each time step, and which determines which of the two inputs has a larger sum over time steps. The hidden unit is linear, and the output unit is logistic. Let's look at what it's doing:*

- *The output unit is a logistic unit with a weight of 5. Recall that large weights squash the function, effectively making it a hard threshold at 0.*
- *The hidden-to-hidden weight is 1, so by default it remembers its previous value.*

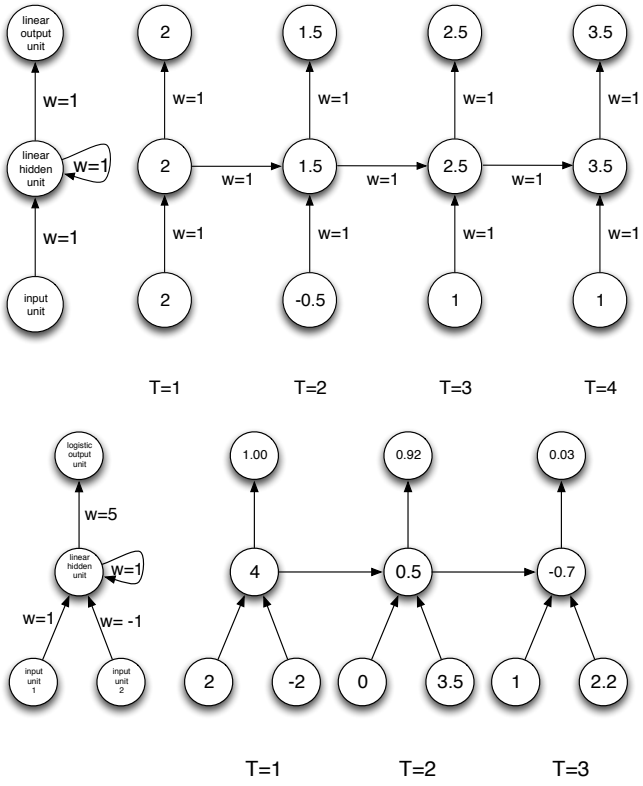


Figure 3: **Top:** the RNN for Example 1. **Bottom:** the RNN for Example 2.

- *The input-to-hidden weights rae 1 and -1, which means it adds one of the inputs and subtracts the other.*

Example 3. *Now let's consider how to get an RNN to perform a slightly more complex computation: the* **parity function**. *This function takes in a sequence of binary inputs, and returns 1 if the number of 1's is odd, and 0 if it is even. It can be computed sequentially by computing the parity of the initial subsequences. In particular, each parity bit is the XOR of the current input with the previous parity bit:*

$$\begin{aligned} \textit{Input:} &\quad 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1 \\ \textit{Parity bits:} &\quad 0\ 1\ 1\ 0\ 1\ 1 \longrightarrow \end{aligned}$$

This suggests a strategy: the output unit $y^{(t)}$ represents the parity bit, and it feeds into the computation at the next time step. In other words, we'd like to achieve the following relationship:

$y^{(t-1)}$	$x^{(t)}$	$y^{(t)}$
0	0	0
0	1	1
1	0	1
1	1	0

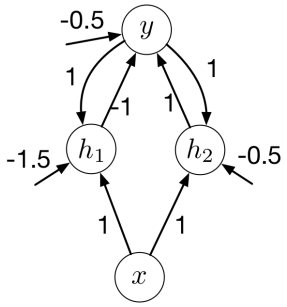


Figure 4: RNN which computes the parity function (Example 3).

But remember that a linear model can't compute XOR, so we're going to need hidden units to help us. Just like in Lecture 5, we can let one hidden unit represent the AND of its inputs and the other hidden unit represent the OR. This gives us the following relationship:

$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*

Based on these computations, the hidden units should receive connections from the input units and the previous time step's output, and the output unit should receive connections from the hidden units. Such an architecture is shown in Figure 4. This is a bit unusual, since output units don't usually feed back into the hiddens, but it's perfectly legal.

The activation functions will all be hard thresholds at 0. Based on this table of relationships, we can pick weights and biases using the same techniques from Lecture 5. This is shown in Figure 4.

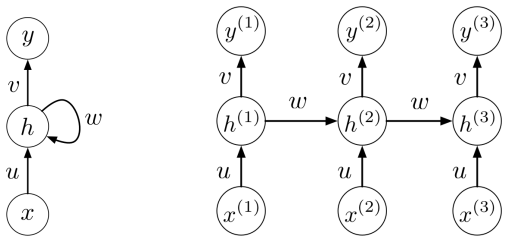
How would you modify this solution to use logistics instead of hard thresholds?

For the first time step, the parity bit should equal the input. This can be achieved by postulating a dummy output $y^{(0)} = 0$.

3 Backprop Through Time

As you can guess, we don't normally set the weights and biases by hand; instead, we learn them using backprop. There actually isn't much new here. All we need to do is run ordinary backprop on the unrolled graph, and account for the weight sharing. Despite the conceptual simplicity, the algorithm gets a special name: **backprop through time**.

Consider the following RNN:



It performs the following computations in the forward pass:

All of these equations are basically like the feed-forward case except $z^{(t)}$.

$$z^{(t)} = ux^{(t)} + wh^{(t-1)} \tag{3}$$
$$h^{(t)} = \phi(z^{(t)}) \tag{4}$$
$$r^{(t)} = vh^{(t)} \tag{5}$$
$$y^{(t)} = \phi(r^{(t)}). \tag{6}$$

Figure 5 shows the unrolled computation graph. Note the weight sharing. Now we just need to do backprop on this graph, which is hopefully a completely mechanical procedure by now:

Pay attention to the rules for $\overline{h^{(t)}}$, $\overline{u}$, $\overline{v}$, and $\overline{w}$.

$$\overline{\mathcal{L}} = 1 \tag{7}$$
$$\overline{y^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial y^{(t)}} \tag{8}$$
$$\overline{r^{(t)}} = \overline{y^{(t)}}\, \phi'(r^{(t)}) \tag{9}$$
$$\overline{h^{(t)}} = \overline{r^{(t)}}\, v + \overline{z^{(t+1)}}\, w \tag{10}$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)}) \tag{11}$$
$$\overline{u} = \sum_{t=1}^{T} \overline{z^{(t)}}\, x^{(t)} \tag{12}$$
$$\overline{v} = \sum_{t=1}^{T} \overline{r^{(t)}}\, h^{(t)} \tag{13}$$
$$\overline{w} = \sum_{t=1}^{T-1} \overline{z^{(t+1)}}\, h^{(t)} \tag{14}$$

These update rules are basically like the ones for an MLP, except that the weight updates are summed over all time steps.

Why are the bounds different in the summations over t?

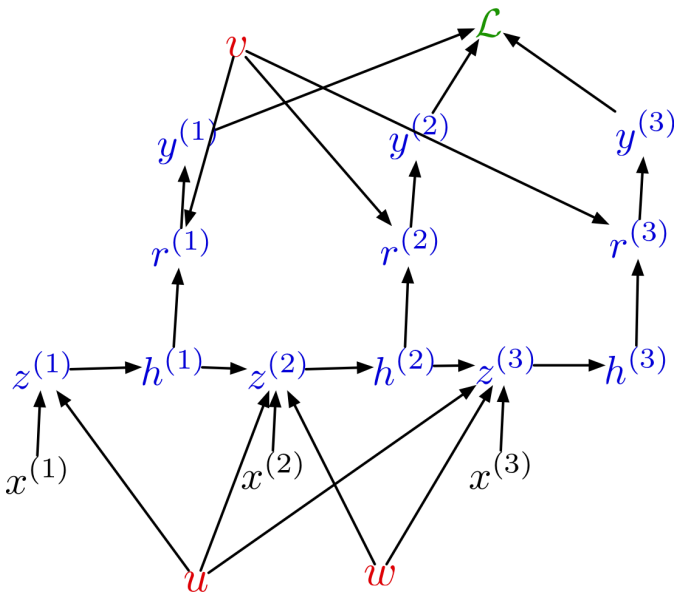


Figure 5: The unrolled computation graph.

The vectorized backprop rules are analogous:

Remember that for all the activation matrices, rows correspond to training examples and columns corresopnd to units, and N is the number of data points (or mini-batch size).

$$\overline{\mathcal{L}} = 1 \tag{15}$$

$$\overline{\mathbf{Y}^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathbf{Y}^{(t)}} \tag{16}$$

$$\overline{\mathbf{R}^{(t)}} = \overline{\mathbf{Y}^{(t)}} \circ \phi'(\mathbf{R}^{(t)}) \tag{17}$$

$$\overline{\mathbf{H}^{(t)}} = \overline{\mathbf{R}^{(t)}} \, \mathbf{V}^\top + \overline{\mathbf{Z}^{(t+1)}} \, \mathbf{W}^\top \tag{18}$$

$$\overline{\mathbf{Z}^{(t)}} = \overline{\mathbf{H}^{(t)}} \circ \phi'(\mathbf{Z}^{(t)}) \tag{19}$$

$$\overline{\mathbf{U}} = \frac{1}{N} \sum_{t=1}^{T} \overline{\mathbf{Z}^{(t)}}^\top \mathbf{X}^{(t)} \tag{20}$$

$$\overline{\mathbf{V}} = \frac{1}{N} \sum_{t=1}^{T} \overline{\mathbf{R}^{(t)}}^\top \mathbf{H}^{(t)} \tag{21}$$

$$\overline{\mathbf{W}} = \frac{1}{N} \sum_{t=1}^{T-1} \overline{\mathbf{Z}^{(t+1)}}^\top \mathbf{H}^{(t)} \tag{22}$$

When implementing RNNs, we generally do an explicit summation over time steps, since there's no easy way to vectorize over time. However, we still vectorize over training examples and units, just as with MLPs.

That's all there is to it. Now you know how to compute cost derivatives for an RNN. The tricky part is how to use these derivatives in optimization. Unless you design the architecture carefully, the gradient descent updates will be unstable because the derivatives explode or vanish over time. Dealing with exploding and vanishing gradients will take us all of next lecture.

4 Sequence Modeling

Now let's look at some ways RNNs can be applied to sequence modeling.

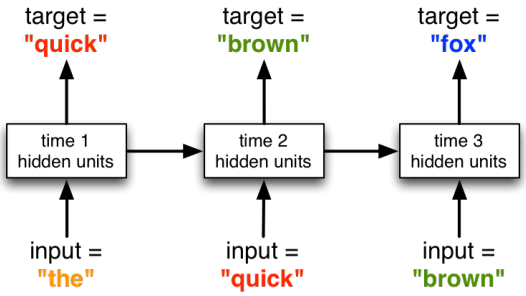
4.1 Language Modeling

We can use RNNs to do language modeling, i.e. model the distribution over English sentences. Just like with n-gram models and the neural language model, we'll use the Chain Rule of Conditional Probability to decompose the distribution into a sequence of conditional probabilities:

$$p(w_1, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_1, \ldots, w_{t-1}), \tag{23}$$

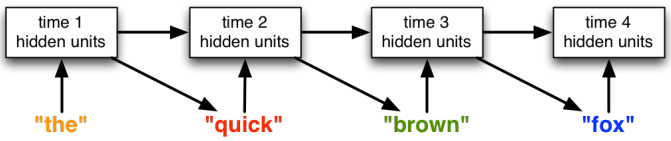
However, unlike with the other two models, we *won't* make a Markov assumption. In other words, the distribution over each word depends on *all* the previous words. We'll make the predictions using an RNN; each of the conditional distributions will be predicted using the output units at a given time step. As usual, we'll use a softmax activation function for the output units, and cross-entropy loss.

At training time, the words of a training sentence are used as both the inputs and the targets to the network, as follows:



It may seem funny to use the sentence as both input and output — isn't it easy to predict a sentence from itself? But each word appears as a target *before* it appears as an input, so there's no way for information to flow from the word-as-input to the word-as-target. That means the network can't cheat by just copying its input.

To generate from the RNN, we sample each of the words in sequence from its predictive distribution. This means we compute the output units for a given time step, sample the word from the corresponding distribution, and then feed the sampled word back in as an input in the next time step. We can represent this as follows:



Remember that vocabularies can get very large, especially once you include proper nouns. As we saw in Lecture 10, it's computationally difficult to predict distributions over millions of words. In the context of a

neural language model, one has to deal with this by changing the scheme for predicting the distribution (e.g. using hierarchical softmax or negative sampling). But RNNs have memory, which gives us another option: we can model text one *character* at a time! In addition to the computational problems of large vocabularies, there are additional advantages to modling text as sequences of characters:

- Any words that don't appear in the vocabulary are implicitly assigned probability 0. But with a character-based language model, there's only a finite set of ASCII characters to consider.
- In some languages, it's hard to define what counts as a word. It's not always as simple as "a contiguous sequence of alphabetical symbols." E.g., in German, words can be built compositionally in terms of simpler parts, so you can create perfectly meaningful words which haven't been said before.

Here's an example from Geoffrey Hinton's Coursera course of a paragraph generated by a character-level RNN which was trained on Wikipedia back in 2011.[1] (Things have improved significantly since then.)

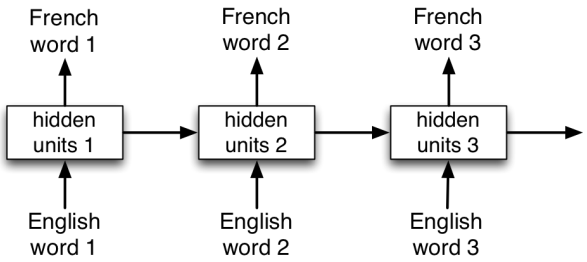
> He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the ephemerable street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

The first thing to notice is that the text isn't globally coherent, so it's clearly not just memorized in its entirety from Wikipedia. But the model produces mostly English words and some grammatical sentences, which is a nontrivial achievement given that it works at the character level. It even produces a plausible non-word, "ephemerable", meaning it has picked up some morphological structure. The text is also locally coherent, in that it starts by talking about politics, and then transportation.

4.2 Neural Machine Translation

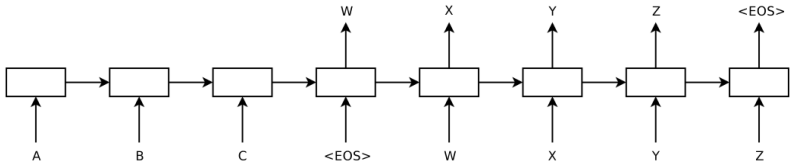
Machine translation is probably the canonical sequence-to-sequence prediction task. Here, the input is a sentence in one language (e.g. English), and the output is a sentence in another language (e.g. French). One could try applying an RNN to this task as follows:

[1] J. Martens. Generating text with recurrent neural networks. ICML, 2011



But this has some clear problems: the sentences might not even be the same length, and even if they were, the words wouldn't be perfectly aligned because different languages have different grammar. There might also be some ambiguous words early on in the sentence which can only be resolved using information from later in the sentence.

Another approach, which was done successfully in 2014[2], is to have the RNN first read the English sentence, remember as much as it can in its hidden units, and then generate the French sentence. This is represented as follows:



The special end-of-sentence token `<EOS>` marks the end of the input. The part of the network which reads the English sentence is known as the **encoder**, and the part that reads the French sentence is known as the **decoder**, and they don't share parameters.

Interestingly, remembering the English sentence is a nontrivial subproblem in itself. We can defined a simplified task called **memorization**, where the network gets an English sentence as input, and has to output the same sentence. Memorization can be a useful testbed for experimenting with RNN algorithms, just as MNIST is a useful testbed for experimenting with classification algorithms.

Before RNNs took over, most machine translation was done by algorithms which tried to transform one sentence into another. The RNN approach described above is pretty different, in that it converts the whole sentence into an abstract semantic representation, and then uses that to generate the French sentence. This is a powerful approach, because the encoders and decoders can be *shared* between different languages. Inputs of any language would be mapped to a common semantic space (which ought to capture the "meaning"), and then any other langage could be generated from that semantic representation. This has actually been made to work, and RNNs are able to perform machine translation on pairs of languages for which there were no aligned pairs in the training set!

[2] I. Sutskever. Sequence to sequence learning with neural networks. 2014

Input:

```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```

Target: 25011.

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else
    5308)
```

Target: 1218.

Input:

```
vqppkn
sqdvfljmnc
y2vxdddsepnimcbvubkomhrpliibtwztbljipcc
```

Target: hkhpg

Figure 6: **Left:** Example inputs for the "learning to execute" task. **Right:** An input with scrambled characters, to highlight the difficulty of the task.

Input:

```
print(6652).
```

Target:	6652.
"Baseline" prediction:	6652.
"Naive" prediction:	6652.
"Mix" prediction:	6652.
"Combined" prediction:	6652.

Input:

```
d=5446
for x in range(8):d+=(2678 if 4803<2829 else 9848)
print((d if 5935<4845 else 3043)).
```

Target:	3043.
"Baseline" prediction:	3043.
"Naive" prediction:	3043.
"Mix" prediction:	3043.
"Combined" prediction:	3043.

```
print((5997-738)).
```

Target:	5259.
"Baseline" prediction:	5101.
"Naive" prediction:	5101.
"Mix" prediction:	5249.
"Combined" prediction:	5229.

Input:

```
print(((1090-3305)+9466)).
```

Target:	7251.
"Baseline" prediction:	7111.
"Naive" prediction:	7099.
"Mix" prediction:	7595.
"Combined" prediction:	7699.

Figure 7: Examples of outputs of the RNNs from the "Learning to execute" paper.

4.3 Learning to Execute Programs

A particularly impressive example of the capabilities of RNNs is that they are able to learn to execute simple programs. This was demonstrated by Wojciech Zaremba and Ilya Sutskever, then at Google.[3] Here, the input to the RNN was a simple Python program consisting of simple arithmetic and control flow, and the target was the result of executing the program. Both the inputs and the targets were fed to the RNN one *character* at a time. Examples are shown in Figure 6.

Their RNN architecture was able to learn to do this fairly well. Some examples of outputs of various versions of their system are shown in Figure 7. It's interesting to look at the pattern of mistakes and try to guess what the networks do or don't understand. For instance, the networks don't really seem to understand carrying: they know that something unusual needs to be done, but it appears they're probably just making a guess.

[3] W. Zaremba and I. Sutskever. Learning to Execute. ICLR, 2015.

Lecture 14: Learning Long-Term Dependencies

Roger Grosse

1 Introduction

Last lecture, we introduced RNNs and saw how to derive the gradients using backprop through time. In principle, this lets us train them using gradient descent. But in practice, gradient descent doesn't work very well unless we're careful. The problem is that we need to learn dependencies over long time windows, and the gradients can explode or vanish.

We'll first look at the problem itself, i.e. why gradients explode or vanish. Then we'll look at some techniques for dealing with the problem — most significantly, changing the architecture to one where the gradients are stable.

1.1 Learning Goals

- Understand why gradients explode or vanish, both
  - in terms of the mechanics of computing the gradients
  - the functional relationship between the hidden units at different time steps
- Be able to analyze simple examples of iterated functions, including identifying fixed points and qualitatively determining the long-term behavior from a given initialization.
- Know about various methods for dealing with the problem, and why they help:
  - Gradient clipping
  - Reversing the input sequence
  - Identity initialization
- Be familiar with the long short-term memory (LSTM) architecture
  - Reason about how the memory cell behaves for a given setting of the input, output, and forget gates
  - Understand how this architecture helps keep the gradients stable

2 Why Gradients Explode or Vanish

Recall the encoder-decoder architecture for machine translation, shown again in Figure 1. It has to read an English sentence, store as much information as possible in its hidden activations, and output a French sentence. The information about the first word in the sentence doesn't get used in the

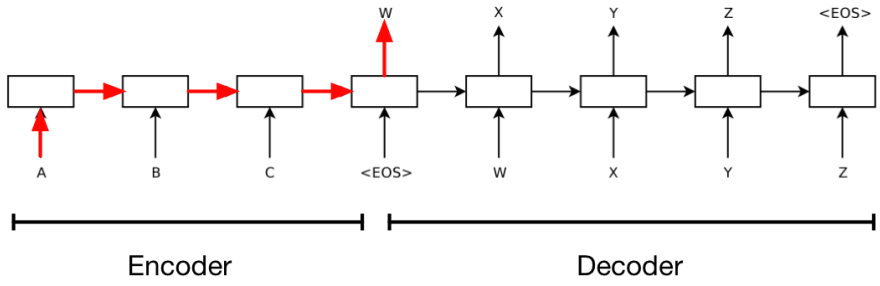
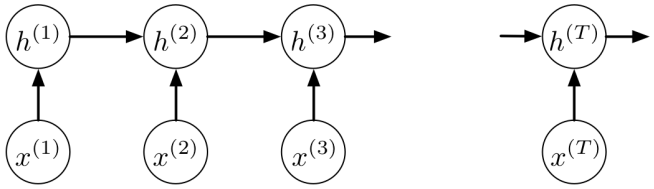


Figure 1: Encoder-decoder model for machine translation (see 14.4.2 for full description). Note that adjusting the weights based on the first input requires the error signal to travel backwards through the entire path highlighted in red.

predictions until it starts generating. Since a typical sentence might be about 20 words long, this means there's a long temporal gap from when it sees an input to when it uses that to make a prediction. It can be hard to learn long-distance dependencies, for reasons we'll see shortly. In order to adjust the input-to-hidden weights based on the first input, the error signal needs to travel backwards through this entire pathway (shown in red in Figure 1).

2.1 The mechanics of backprop

Now consider a univariate version of the encoder:



Assume we've already backpropped through the decoder network, so we already have the error signal $\overline{h^{(T)}}$. We then alternate between the following two backprop rules:[1]

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

If we iterate the rules, we get the following formula:

$$\begin{aligned}\overline{h^{(1)}} &= w^{T-1}\phi'(z^{(2)}) \cdots \phi'(z^{(T)})\overline{h^{(T)}} \\ &= \frac{\partial h^{(T)}}{\partial h^{(1)}}\overline{h^{(T)}}\end{aligned}$$

Hence, $\overline{h^{(1)}}$ is a linear function of $\overline{h^{(T)}}$. The coefficient is the partial derivative $\frac{\partial h^{(T)}}{h^{(1)}}$. If we make the simplifying assumption that the activation func-

[1] Review Section 14.3 if you're hazy on backprop through time.

tions are linear, we get

$$\frac{\partial h^{(T)}}{\partial h^{(1)}} = w^{T-1},$$

which can clearly explode or vanish unless w is very close to 1. For instance, if $w - 1.1$ and $T = 50$, we get $\partial h^{(T)}/\partial h^{(1)} = 117.4$, whereas if $w = 0.9$ and $T = 50$, we get $\partial h^{(T)}/\partial h^{(1)} = 0.00515$. In general, with nonlinear activation functions, there's nothing special about $w = 1$; the boundary between exploding and vanishing will depend on the values $h^{(t)}$.

More generally, in the multivariate case,

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}.$$

This quantity is called the **Jacobian**. It can explode or vanish just like in the univariate case, but this is slightly more complicated to make precise. In the case of linear activation functions, $\partial \mathbf{h}^{(t+1)}/\partial \mathbf{h}^{(t)} = \mathbf{W}$, so

"Jacobian" is a general mathematical term for the matrix of partial derivatives of a vector-valued function.

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \mathbf{W}^{T-1}.$$

This will explode if the largest eigenvalue of $\mathbf{W}$ is larger than 1, and vanish if the largest eigenvalue is smaller than 1.

Contrast this with the behavior of the forward pass. In the forward pass, the activations at each step are put through a nonlinear activation function, which typically squashes the values, preventing them from blowing up. Since the backwards pass is entirely linear, there's nothing to prevent the derivatives from blowing up.

2.2 Iterated functions

We just talked about why gradients explode or vanish, in terms of the mechanics of backprop. But whenever you're trying to reason about a phenomenon, don't go straight to the equations. Instead, try to think qualitatively about what's going on. In this case, there's actually a nice interpretation of the problem in terms of the function the RNN computes. In particular, each layer computes a function of the current input and the previous hidden activations, i.e. $\mathbf{h}^{(t)} = f(\mathbf{x}^{(t)}, \mathbf{h}^{(t-1)})$. If we expand this recursively, we get:

$$\mathbf{h}^{(4)} = f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)}). \tag{1}$$

This looks a bit like repeatedly applying the function f. Therefore, we can gain some intuition for how RNNs behave by studying **iterated functions**, i.e. functions which we iterate many times.

Iterated functions can be complicated. Consider the innocuous-looking quadratic function

$$f(x) = 3.5\, x\, (1 - x). \tag{2}$$

If we iterate this function multiple times (i.e. $f(f(f(x)))$, etc.), we get some complicated behavior, as shown in Figure 2. Another famous example of the complexity of iterated functions is the Mandelbrot set:

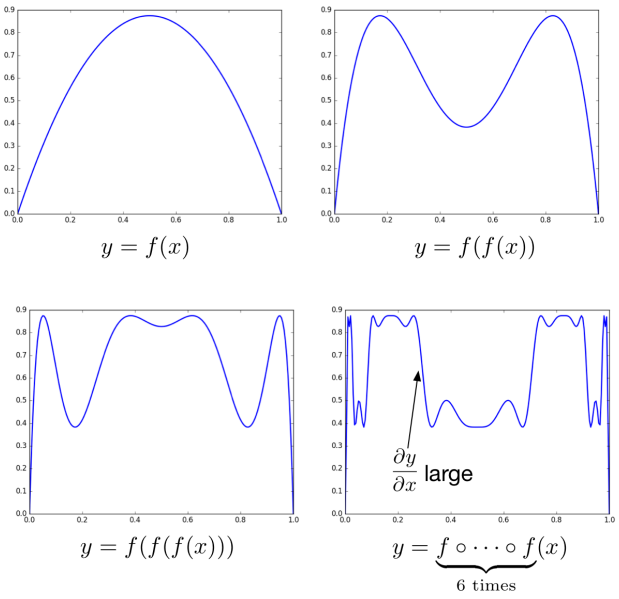
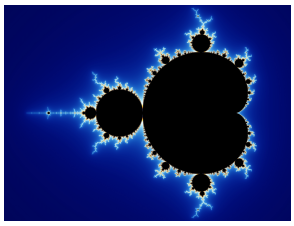


Figure 2: Iterations of the function $f(x) = 3.5\,x\,(1-x)$.



This is defined in terms of a simple mapping over the complex plane:

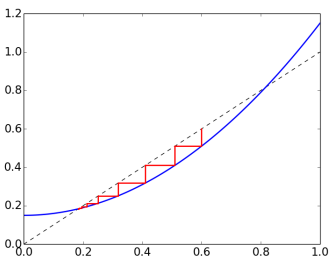
$$z_n = z_{n-1}^2 + c \tag{3}$$

If you initialize at $z_0 = 0$ and iterate this mapping, it will either stay within some bounded region or shoot off to infinity, and the behavior depends on the value of c. The Mandelbrot set is the set of values of c where it stays bounded; as you can see, this is an incredibly complex fractal.

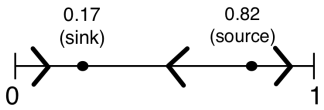
It's a bit easier to analyze iterated functions if they're monotonic. Consider the function

$$f(x) = x^2 + 0.15.$$

This is monotonic over $[0, 1]$. We can determine the behavior of repeated iterations visually:

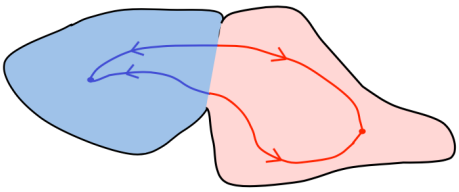


Here, the red line shows the trajectory of the iterates. If the initial value is $x_0 = 0.6$, start with your pencil at $x = y = 0.6$, which lies on the dashed line. Set $y = f(x)$ by moving your pencil vertically to the graph of the function, and then set $x = y$ by moving it horizontally to the dashed line. Repeat this procedure, and you should notice a pattern. There are some regions where the iterates move to the left, and other regions where the move to the right. Eventually, the iterates either shoot off to infinity or wind up at a **fixed point**, i.e. a point where $x = f(x)$. Fixed points are represented graphically as points where the graph of x intersects the dashed line. Some fixed points (such as 0.82 in this example) repel the iterates; these are called **sources**. Other fixed points (such as 0.17) attract the iterates; these are called **sinks**, or **attractors**. The behavior of the system can be summarized with a **phase plot**:



Observe that fixed points with derivatives $f'(x) < 1$ are sinks and fixed points with $f'(x) > 1$ are sources.

Even though the computations of an RNN are discrete, we can think of them as a sort of dynamical system, which has various attractors:



– Geoffrey Hinton, Coursera

This figure is a cartoon of the space of hidden activations. If you start out in the blue region, you wind up in one attractor, whereas if you start out in the red region, you wind up in the other attractor. If you evaluate the Jacobian $\partial \mathbf{h}^{(T)} / \partial \mathbf{h}^{(1)}$ in the interior of one of these regions, it will be close to 0, since if you change the initial conditions slightly, you still wind up at exactly the same place. But the Jacobian right on the boundary will be large, since shifting the initial condition slightly moves us from one attractor to the other.

To make this story more concrete, consider the following RNN, which uses the tanh activation function:

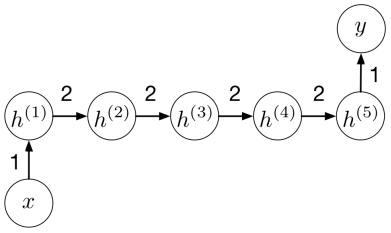


Figure 3 shows the function computed at each time step, as well as the function computed by the network as a whole. From this figure. you can see which regions have exploding or vanishing gradients.

Think about how we can derive the right-hand figure from the left-hand one using the analysis given above.

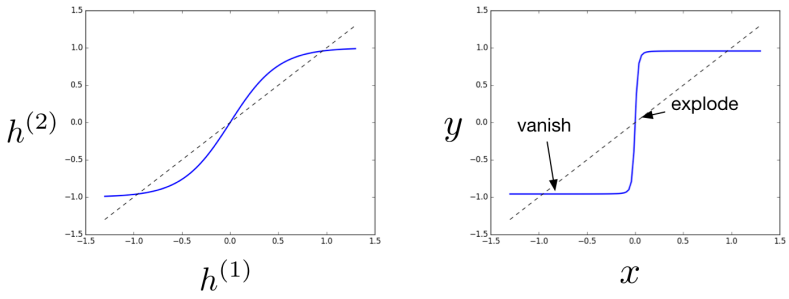


Figure 3: **(left)** The function computed by the RNN at each time step, **(right)** the function computed by the network.

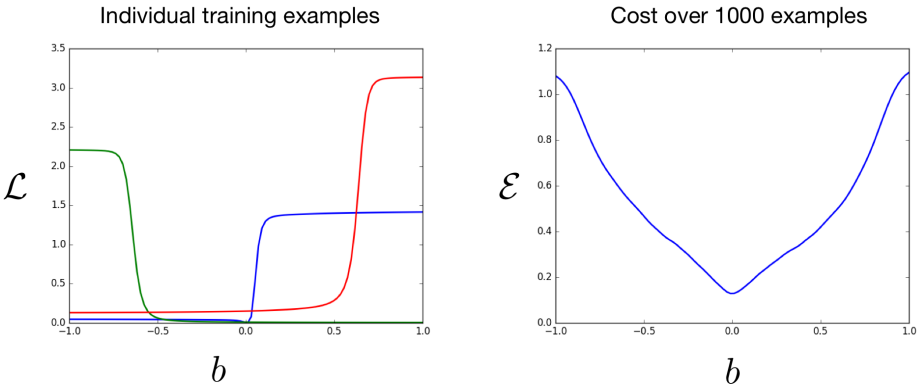


Figure 4: **(left)** Loss function for individual training examples, **(right)** cost function averaged over 1000 training examples.

This behavior shows up even if we look at the gradients with respect to parameters of the network. Suppose we define an input distribution and loss function for this network; we'll use squared error loss, but the details aren't important. Figure 4 shows the loss function for individual training examples, as well as the cost function averaged over 1000 training examples. Recall our discussion of features of the optimization landscape (plateaux, ridges, etc.). This figure shows a new one, namely **cliffs**. In this case, cliffs are a problem only for individual training examples; the cost function averaged over 1000 examples is fairly smooth. Whether or not this happens depends on the specifics of the problem.

3 Keeping Things Stable

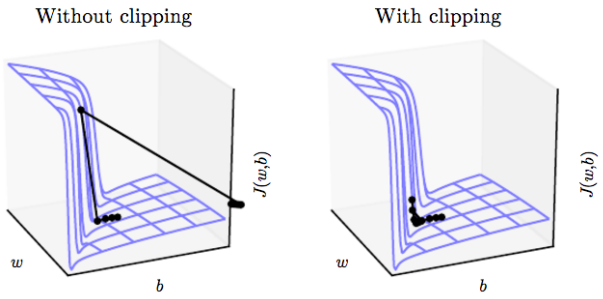
Now that we've introduced the problem of exploding and vanishing gradients, let's see what we can do about it. We'll start with some simple tricks, and then consider a fundamental change to the network architecture.

3.1 Gradient Clipping

First, there's a simple trick which sometimes helps a lot: **gradient clipping**. Basically, we prevent gradients from blowing up by rescaling them so that their norm is at most a particular value η. I.e., if $\|\mathbf{g}\| > \eta$, where $\mathbf{g}$ is the gradient, we set

$$\mathbf{g} \leftarrow \frac{\eta \mathbf{g}}{\|\mathbf{g}\|}. \tag{4}$$

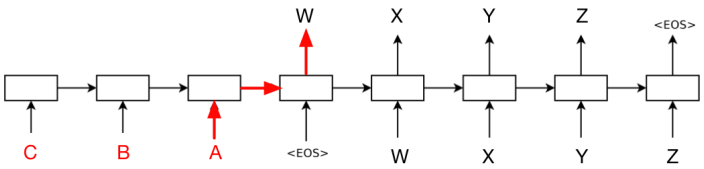
This biases the training procedure, since the resulting values won't actually be the gradient of the cost function. However, this bias can be worth it if it keeps things stable. The following figure shows an example with a cliff and a narrow valley; if you happen to land on the face of the cliff, you take a huge step which propels you outside the good region. With gradient clipping, you can stay within the valley.



— Goodfellow et al., *Deep Learning*

3.2 Input Reversal

Recall that we motivated this whole discussion in terms of the difficulty of learning long-distance dependencies, such as between the first word of the input sentence and the first word of the output sentence. What makes it especially tricky in translation is that *all* of the dependencies are long; this happens because for similar languages like English and French, the corresponding words appear in roughly the same order in both sentences, so the gaps between input and output are all roughly the sentence length. We can fix this by reversing the order of the words in the input sentence:



There's a gap of only one time step between when the first word is read and when it's needed. This means that the network can easily learn the relationships between the first words; this could allow it to learn good word representations, for instance. Once it's learned this, it can go on to the more difficult dependencies between words later in the sentences.

3.3 Identity Initialization

In general, iterated functions can have complex and chaotic behavior. But there's one particular function you can iterate as many times as you like: the identity function $f(x) = x$. If your network computes the identity function, the gradient computation will be perfectly stable, since the Jacobian is simply the identity matrix. Of course, the identity function isn't a very interesting thing to compute, but it still suggests we can keep things stable by encouraging the computations to stay close to the identity function.

The **identity RNN** architecture[2] is a kind of RNN where the activation functions are all ReLU, and the recurrent weights are initialized to the identity matrix. The ReLU activation function clips the activations to be nonnegative, but for nonnegative activations, it's equivalent to the identity function. This simple initialization trick achieved some neat results; for instance, it was able to classify MNIST digits which were fed to the network one pixel at a time, as a length-784 sequence.

3.4 Long-Term Short Term Memory

We've just talked about three tricks for training RNNs, and they are all pretty widely used. But the identity initialization trick actually gets at something much more fundamental. That is, why is it a good idea for the RNN to compute something close to the identity function? Think about how a computer works. It has a very large memory, but each instruction accesses only a handful of memory locations. All the other memory locations simply keep their previous value. In other words, if the computer's entire memory is represented as one big vector, the mapping from one time step to the next is very close to the identity function. This behavior is the most basic thing we'd desire from a memory system: the ability to preserve information over time until it's needed.

Unfortunately, the basic RNN architectures we've talked about so far aren't very good at remembering things. All of the units we've covered so far in the course consist of linear functions followed by a nonlinear activation function:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{5}$$

For some activation functions, such as logistic or tanh, this can't even represent the identity mapping; e.g., in the network shown in Figure 3, each time step computes a function fairly close to the identity, but after just 5 steps, you get a step function.

The **Long-Term Short-Term Memory (LSTM)** architecture was designed to make it easy to remember information over long time periods until it's needed. The name refers to the idea that the activations of a network correspond to short-term memory, while the weights correspond to long-term memory. If the activations can preserve information over long distances, that makes them long-term short-term memory.

The basic LSTM unit (called a **block**) has much more internal structure than the units we've covered so far in this course. The architecture is shown in Figure 5. Each hidden layer of the RNN will be composed of many (e.g. hundreds or thousands) of these blocks.

[2] Le et al., 2015. A simple way to initialize recurrent networks of rectified linear units.

- At the center is a **memory cell**, which is the thing that's able to remember information over time. It has a linear activation function, and a self-loop which is **modulated** by a **forget gate**, which takes values between 0 and 1; this means that the weight of the self-loop is equal to the value of the forget gate.
- The forget gate is a unit similar to the ones we've covered previously; it computes a linear function of its inputs, followed by a logistic activation function (which means its output is between 0 and 1). The forget gate would probably be better called a "remember gate", since if it is on (takes the value 1), the memory cell remembers its previous value, whereas if the forget gate is off, the cell forgets it.
- The block also receives inputs from other blocks in the network; these are summed together and passed through a tanh activation function (which squashes the values to be between -1 and 1). The connection from the input unit to the memory cell is gated by an **input gate**, which has the same functional form as the forget gate (i.e., linear-then-logistic).
- The block produces an output, which is the value of the memory cell, passed through a tanh activation function. It may or may not pass this on to the rest of the network; this connection is modulated by the **output gate**, which has the same form as the input and forget gates.

It's useful to summarize various behaviors the memory cell can achieve depending on the values of the input and forget gates:

input gate	forget gate	behavior
0	1	remember the previous value
1	1	add to the previous value
0	0	erase the value
1	0	overwrite the value

If the forget gate is on and the input gate is off, the block simply computes the identity function, which is a useful default behavior. But the ability to read and write from it lets it implement more sophisticated computations. The ability to add to the previous value means these units can simulate a counter; this can be useful, for instance, when training a language model, if sentences tend to be of a particular length.

When we implement an LSTM, we have a bunch of vectors at each time step, representing the values of all the memory cells and each of the gates. Mathematically, the computations are as follows:

$$\begin{pmatrix} \mathbf{i}^{(t)} \\ \mathbf{f}^{(t)} \\ \mathbf{o}^{(t)} \\ \mathbf{g}^{(t)} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{pmatrix} \tag{6}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \mathbf{g}^{(t)} \tag{7}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh(\mathbf{c}^{(t)}). \tag{8}$$

Here, (6) uses a shorthand for applying different activation functions to different parts of the vector. Observe that the blocks receive signals from

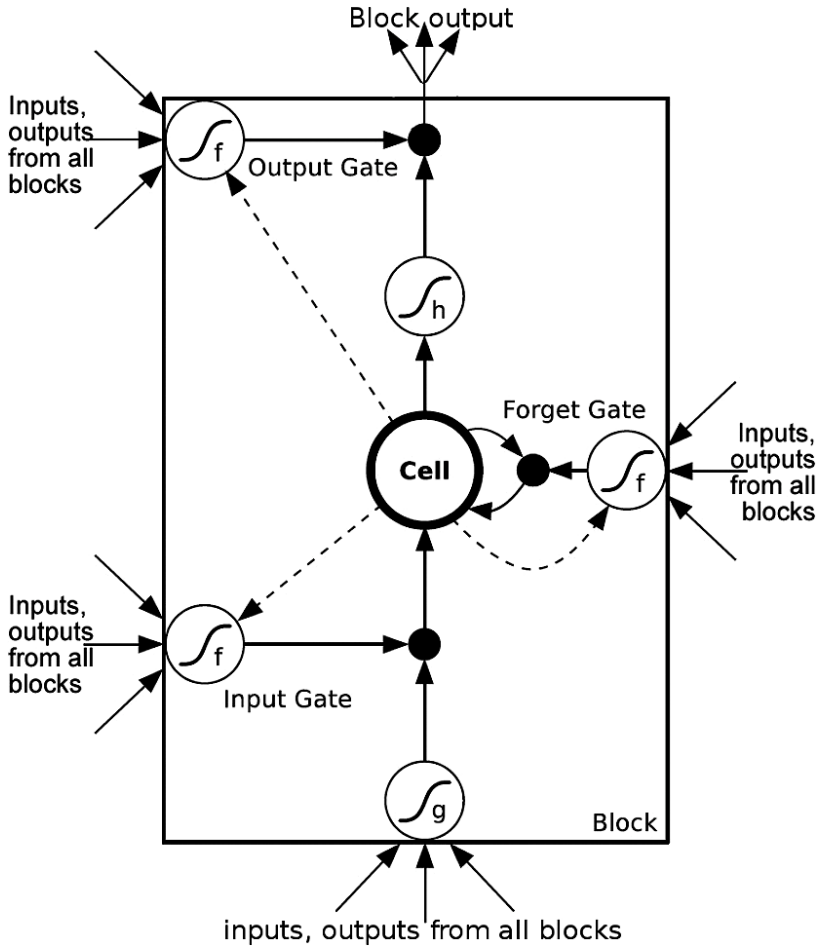


Figure 5: The LSTM unit.

the current inputs and the previous time step's hidden units, just like in standard RNNs. But the network's input $\mathbf{g}$ and the three gates $\mathbf{i}$, $\mathbf{o}$, and $\mathbf{f}$ have independent sets of incoming weights. Then (7) gives the update rule for the memory cell (think about how this relates to the verbal description above), and (8) defines the output of the block.

For homework, you are asked to show that if the forget gate is on and the input and output gates are off, it just passes the memory cell gradients through unmodified at each time step. Therefore, the LSTM architecture is resistant to exploding and vanishing gradients, although mathematically both phenomena are still possible.

If the LSTM architecture sounds complicated, that was the reaction of machine learning researchers when it was first proposed. It wasn't used much until 2013 and 2014, when resesarchers achieved impressive results on two challenging and important sequence prediction problems: speech-to-text and machine translation. Since then, they've become one of the most widely used RNN architectures; if someone tells you they're using an RNN, there's a good chance they're actually using an LSTM. There have been many attempts to simplify the architecture, and one particular variant called the gated recurrent unit (GRU) is fairly widely used, but so far nobody has found anything that's both simpler and at least as effective across the board. It appears that most of the complexity is probably required. Fortunately, you hardly ever have to think about it, since LSTMs are implemented as a black box in all of the major neural net frameworks.

4 ResNets

Before 2015, the GoogLeNet (Inception) architecture set the standard for a deep conv net. It was about 20 layers deep, not counting pooling. In 2015, the new state-of-the-art on ImageNet was the deep residual network (ResNet), which had the distinction that that it was 150 layers deep. When we discussed image classification, I promised we'd come back to ResNets once we covered a key conceptual idea. That idea was exploding and vanishing gradients.

Recall that the Jacobian $\partial \mathbf{h}^{(T)}/\partial \mathbf{h}^{(1)}$ for an RNN is the product of the Jacobians of individual layers:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

Multiplying together lots of matrices causes the Jacobian to explode or vanish unless we're careful about keeping all of them close to the identity. But notice that this same formula applies to the Jacobian for a feed-forward network (e.g. MLP or conv net). How come we never talked about exploding and vanishing gradients until we got to RNNs? The reason is that until recently, feed-forward nets were at most tens of layers deep, whereas RNNs would often be unrolled for hundreds of time steps. Hence, we'd be doing lots more steps of backprop (i.e. multiplying lots of Jacobians together), making things more likely to explode or vanish. This means if we want to train feed-forward nets with hundreds of layers, we need to figure out how to keep the backprop computations stable.

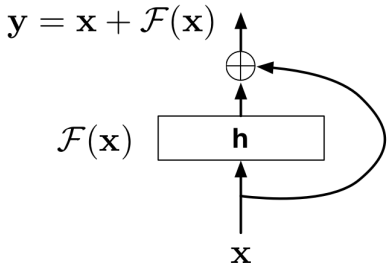
In Homework 3, you derived the backprop equations for the following architecture, where the inputs get added to the outputs:

$$\begin{aligned} \mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{h} &= \phi(\mathbf{z}) \\ \mathbf{y} &= \mathbf{x} + \mathbf{W}^{(2)}\mathbf{h} \end{aligned} \tag{9}$$

This is a special case of a more general architectural primitive called the **residual block**:

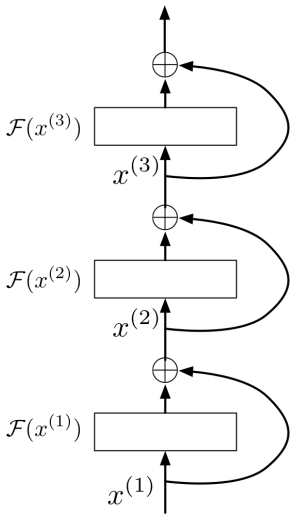
$$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x}), \tag{10}$$

where $\mathcal{F}$ is a function called the **residual function**. In the above example, $\mathcal{F}$ is an MLP with one hidden layer. In general, it's typically a shallow neural net, with 1–3 hidden layers. We can represent the residual block graphically as follows:



Here, $\oplus$ denotes addition of the values.

We can string together multiple residual blocks in series to get a **deep residual network**, or **ResNet**:



(Each layer computes a separate residual function, with separate trainable parameters.) Last lecture, we noted two architectures that make it easy to represent the identity function: identity RNNs and LSTMs. The ResNet is a third such architecture. Observe that if each $\mathcal{F}$ returns zero (e.g. because all the weights are 0), then this architecture simply passes the input $\mathbf{x}$ through unmodified. I.e., it computes the identity function.

We can also see this algebraically in terms of the backprop equation for a residual block:

$$\begin{aligned}\overline{\mathbf{x}^{(\ell)}} &= \overline{\mathbf{x}^{(\ell+1)}} + \overline{\mathbf{x}^{(\ell+1)}}\,\frac{\partial \mathcal{F}}{\partial \mathbf{x}} \\ &= \overline{\mathbf{x}^{(\ell+1)}}\left(\mathbf{I} + \frac{\partial \mathcal{F}}{\partial \mathbf{x}}\right)\end{aligned} \tag{11}$$

Hence, if $\partial \mathcal{F}/\partial \mathbf{x} = 0$, the error signals are simply passed through unmodified. As long as $\partial \mathcal{F}/\partial \mathbf{x}$ is small, the Jacobian for the residual block will be close to the identity, and the error signals won't explode or vanish.

So that's the one big idea behind ResNets. If people say they are using ResNets for a vision task, they're probably referring to particular architectures based on the ones in this paper[3]. This paper achieved state-of-the-art on ImageNet in 2015, and since then, the state-of-the-art on many computer vision tasks has consisted of variants of these ResNet architectures. There's one important detail that needs to be mentioned: the input and output to a residual block clearly need to be the same size, because the output is the sum of the input and the residual function. But for conv nets, it's important to shrink the images (e.g. using pooling) in order to expand the number of feature maps. ResNets typically achieve this by having a few convolution layers with a stride of 2, so that the dimension of the image is reduced by a factor of 2 along each dimension.

The benefit of the ResNet architecture is that it's possible to train absurdly large numbers of layers. The state-of-the-art ImageNet classifier from the above paper had 50 residual blocks, and the residual function for each was a 3-layer conv net, so the network as a whole had about 150 layers. Hardly anybody had been expecting it to be useful to train 150 layers. On a smaller object recognition benchmark called CIFAR, they were actually able to train a ResNet with 1000 layers, though it didn't work any better than their 100-layer network.

What on earth are all these layers doing? When we visualized the Inception activations, we found pretty good evidence that higher layers were learning more abstract and high-level features. But the idea that there are 150 meaningfully different levels of abstraction seems pretty fishy. We actually don't have a good explanation for why 150 layers works better than 50.

[3] K. He, X. Zhang, S. Ren, and J. Sun, 2016. Deep residual learning for image recognition

Lecture 15: Autoregressive and Reversible Models

Roger Grosse

In this lecture, we'll cover two kinds deep generative model architectures which can be trained using maximum likelihood. The first kind is reversible architectures, where the network's computations can be inverted in order to recover the input which maps to a given output. We'll see that this makes the likelihood computation tractable.

The second kind of architecture is autoregressive models. This isn't new: we've already covered neural language models and RNN language models, both of which are examples of autoregressive models. In this lecture, we'll introduce two tricks for making them much more scalable, so that we can apply them to high-dimensional data modalities like high-resolution images and audio waveforms.

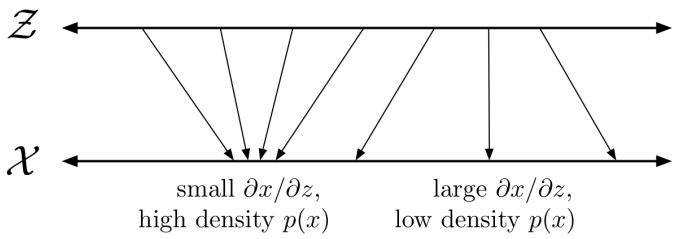
1 Reversible Models

Mathematically, reversible models are based on the **change-of-variables formula** for probability density functions. Suppose we have a bijective, differentiable mapping $f : \mathcal{Z} \to \mathcal{X}$. ("Bijective" means the mapping must be 1–1 and cover all of $\mathcal{X}$.) Since f is bijective, we can think of it as representing a change-of-variables transformation. For instance, $\mathbf{x} = f(\mathbf{z}) = 12\mathbf{z}$ could represent a conversion of units from feet to inches. If we have a density $p_Z(\mathbf{z})$, the change-of-variables formula gives us the density $p_X(\mathbf{x})$:

$$p_X(\mathbf{x}) = p_Z(\mathbf{z}) \left| \det \left(\frac{\partial \mathbf{x}}{\partial \mathbf{z}} \right) \right|^{-1}, \tag{1}$$

where $\mathbf{z} = f^{-1}(\mathbf{x})$. Let's unpack this. First, $\partial \mathbf{x}/\partial \mathbf{z}$ is the Jacobian of f, which is the linearization of f around $\mathbf{z}$. Then we take the absolute value of the matrix determinant. Recall that the absolute value of the determinant of a matrix gives the factor by which the associated linear transformation expands or contracts the volume of a set. So the determinant of the Jacobian determines how much f is expanding or contracting the volume locally around $\mathbf{z}$. We then take the inverse of the determinant, which means if f expands the volume, then the density $p_X(\mathbf{x})$ shrinks, and vice versa. Heuristically, this is justified by the following picture:

If we consider the linear transformation $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ for some matrix $\mathbf{A}$, and apply it to a set with volume V, we'll get a set with volume $V|\det \mathbf{A}|$.



Now suppose the mapping f is the function computed by a generator network (i.e. its outputs as a function of its inputs). It's tempting to apply the change-of-variables formula in order to compute $p_X(\mathbf{x})$. But in order for this to work, three things need to be true:

1. The mapping f needs to be differentiable, so that the Jacobian $\partial \mathbf{x}/\partial \mathbf{z}$ is defined.

2. We need to be able to compute $\mathbf{z} = f^{-1}(\mathbf{x})$, which means f needs to be invertible, with an easy-to-compute inverse.

3. We need to be able to compute the (log) determinant of the Jacobian.

With regards to (1), networks with ReLU nonlinearities technically aren't differentiable because ReLU is nondifferentiable at 0. In practice, we can ignore this issue because the inputs to the activation function are very unlikely to be exactly zero, so with high probability, the Jacobian will be defined. Or, if we're still worried, we could just pick a differentiable activation function. But the other two points are much harder to deal with.

Fortunately, there's a simple and elegant kind of network architecture called a **reversible architecture** which is efficiently invertible and for which we can compute the log determinant efficiently. (In fact, the determinant turns out to be 1.) This architecture is based on the **reversible block**, which is very similar to the residual block from Lecture 17. Recall that residual blocks implement the following equation:

$$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x}), \tag{2}$$

where $\mathcal{F}$ is some function, such as a shallow network. Reversible blocks are similar, except that we divide the units into two groups; the residual function for the first group depends only on the other group, and the second group is left unchanged. Mathematically,

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_2) \\ \mathbf{y}_2 &= \mathbf{x}_2 \end{aligned} \tag{3}$$

This is shown schematically in Figure 1. The reversible block is easily inverted, i.e. if we're given $\mathbf{y}_1$ and $\mathbf{y}_2$, we can recover $\mathbf{x}_1$ and $\mathbf{x}_2$:

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{y}_2 \\ \mathbf{x}_1 &= \mathbf{y}_1 - \mathcal{F}(\mathbf{x}_2) \end{aligned} \tag{4}$$

Here's what happens when we compose two residual blocks, with the roles of $\mathbf{x}_1$ and $\mathbf{x}_2$ swapped:

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_2) \\ \mathbf{y}_2 &= \mathbf{x}_2 + \mathcal{G}(\mathbf{y}_1) \end{aligned} \tag{5}$$

This is shown schematically in Figure 1. To invert the composition of two blocks:

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{y}_2 - \mathcal{G}(\mathbf{y}_1) \\ \mathbf{x}_1 &= \mathbf{y}_1 - \mathcal{F}(\mathbf{x}_2) \end{aligned} \tag{6}$$

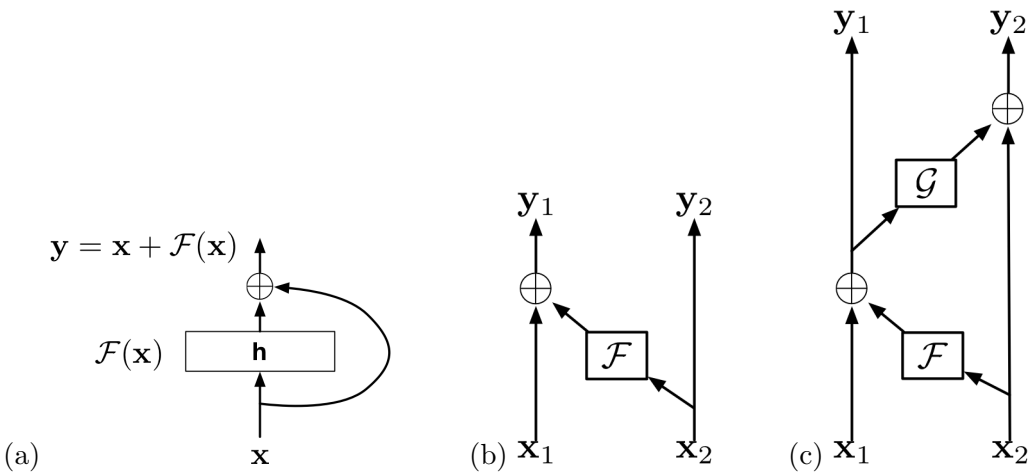
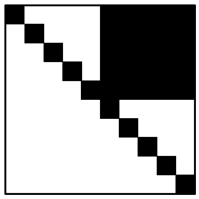


Figure 1: **(a)** A residual block. **(b)** A reversible block. **(c)** A composition of two reversible blocks.

So we've shown how to invert a reversible block. What about the determinant of the Jacobian? Here is the formula for the Jacobian, which we get by differentiating Eqn. 3 and putting the result into the form of a block matrix:

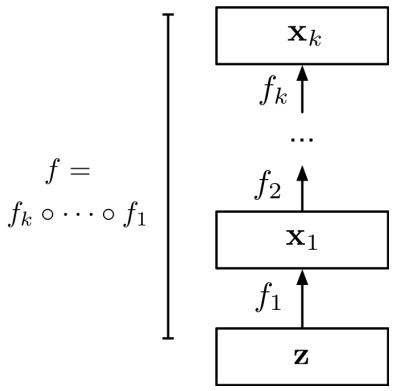
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \mathbf{I} & \frac{\partial \mathcal{F}}{\partial \mathbf{x}_2} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

(Recall that $\mathbf{I}$ denotes the identity matrix.) Here's the pattern of nonzero entries of this matrix:



This is an upper triangular matrix. Think back to linear algebra class: the determinant of an upper triangular matrix is simply the product of the diagonal entries. In this case, the diagonal entries are all 1's, so the determinant is 1. How convenient! Since the determinant is 1, the mapping is **volume preserving**, i.e. it maps any given set to another set of the same volume. In our context, this just means the determinant term disappears from the change-of-variables formula (Eqn. 1).

All this analysis so far was for a single reversible block. What if we build a reversible network by chaining together lots of reversible blocks?



Fortunately, inversion of the whole network is still easy, since we just invert each block from top to bottom. Mathematically,

$$f^{-1} = f_1^{-1} \circ \cdots \circ f_k^{-1}. \tag{7}$$

For the determinant, we can apply the chain rule for derivatives, followed by the product rule for determinants:

$$\begin{aligned} \left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{z}}\right| &= \left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}} \cdots \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \mathbf{z}}\right| \\ &= \left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}\right| \cdots \left|\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}\right| \left|\frac{\partial \mathbf{x}_1}{\partial \mathbf{z}}\right| \\ &= 1 \cdot 1 \cdots 1 \\ &= 1 \end{aligned} \tag{8}$$

Hence, the full reversible network is also volume preserving.

Because we can compute inverses and determinants, we can train a reversible generative model using maximum likelihood using the change-of-variables formula. This is the idea behind **nonlinear independent components estimation (NICE)**[1]. (This paper introduced the idea of training reversible architectures with maximum likelihood.) The change-of-variables formula gives us:

$$\begin{aligned} p_X(\mathbf{x}) &= p_Z(\mathbf{z}) \left|\det\left(\frac{\partial \mathbf{x}}{\partial \mathbf{z}}\right)\right|^{-1} \\ &= p_Z(\mathbf{z}) \end{aligned} \tag{9}$$

Hence, the maximum likelihood objective over the whole dataset is:

$$\prod_{i=1}^{N} p_X(\mathbf{x}^{(i)}) = \prod_{i=1}^{N} p_Z(f^{-1}(\mathbf{x}^{(i)})) \tag{10}$$

Remember, p_Z is a simple, fixed distribution (e.g. independent Gaussians), so $p_Z(\mathbf{z})$ is easy to evaluate. Note that this objective only makes sense because of the volume constraint.

If f weren't constrained to be volume preserving, then f^{-1} could map every training example very close to $\mathbf{0}$, and hence $p_Z(f^{-1}(\mathbf{x}^{(i)}))$ would be large for every training example. The volume preservation constraint prevents this trivial solution.

2 Autoregressive Models

Autoregressive models are another kind of deep generative model with tractable likelihoods. We've already seen two examples in this course: the

[1] Dinh et al., 2014. NICE: Non-linear independent components estimation.

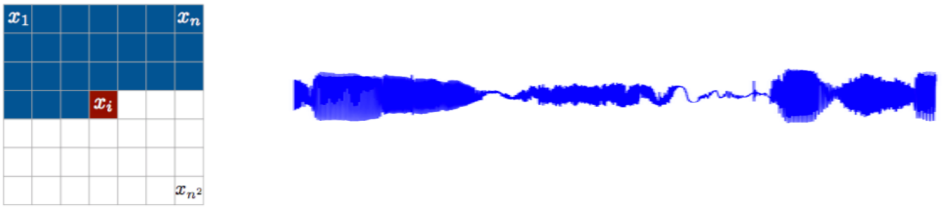


Figure 2: Examples of sequence modeling tasks with very long contexts. **Left:** Modeling images as sequences using raster scan order. **Right:** Modeling an audio waveform (e.g. speech signal).

neural language model (Lecture 5) and RNNs (Lectures 13-14). Here, the observations were given as sequences $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)})$, and we decomposed the likelihood into a product of conditional distributions:

$$p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}) = \prod_{t=1}^{T} p(\mathbf{x}^{(t)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t-1)}). \tag{11}$$

So the maximum likelihood objective decomposes as a sequence of prediction problems for each term in the sequence given the previous terms. Assuming the observations were discrete (as they are in all the autoregressive models considered in this course), the prediction at each time step can be made using a neural network which outputs a probability distribution using a softmax activation function.

So far, we've mostly considered using short sequences or short context lengths. The neural language model from Assignment 1 used context windows of length 3, though the architecture could work better with contexts of length 10 or so. Machine translation at the word level involves outputting sequences of length 20 or so (the typical number of words in a sentence).

But what if accurately modeling the distribution requires much longer term dependencies? One example is autoregressive models of images. A grayscale image is typically represented with pixel values which are integers from 0 to 255 (i.e. one byte each). We can treat this as a sequence using the raster scan order (Figure 2). But even a fairly small image would correspond to a very long sequence, e.g. a 100×100 image would correspond to a sequence of length 10,000. Clearly, images have a lot of global structure, so the predictions would need to take into account all of the pixels which were already generated. As another example, consider learning a generative model of audio waveforms. An audio waveform is stored as a sequence of integer-valued samples, with a sampling rate of at least 16,000 Hz (cycles/second) in order to have reasonably good sound quality. This means that to predict the next term in the sequence, if we want to account for even 1 second of context, this requires a context of length 16,000.

One way to account for such a long context is to use an RNN, which (through its hidden units) accounts for the entire sequence that was generated so far. The problem is that computing the hidden units for each time step depends on the hidden units from the previous time step, so the forward pass of backprop requires a `for`-loop over time steps. (The backward

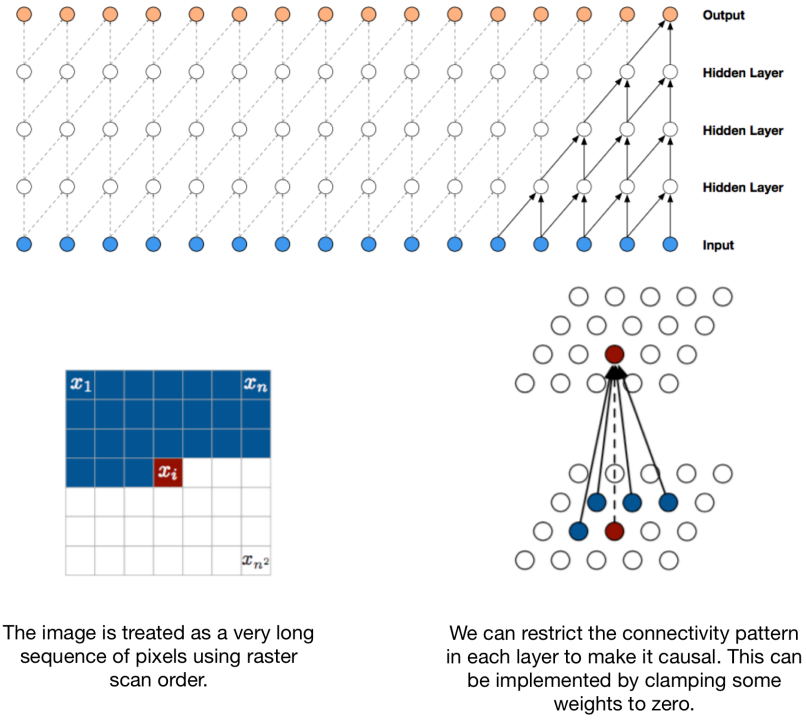


Figure 3: **Top:** a causal CNN applied to sequential data (such as an audio waveform). Source: van den Oord et al., 2016, "WaveNet: a generative model for raw audio". **Bottom:** applying causal convolution to modeling images. Source: van den Oord et al., 2016, "Pixel recurrent neural networks".

pass requires a `for`-loop as well.) With thousands of time steps, this can get very expensive. But think about the neural language model architecture from Lecture 7. At training time, the predictions at each time step are done independently of each other, so all the time steps can be processed simultaneously with vectorized computations. This implies that training with very long sequences could be done much more efficiently if we could somehow get rid of the recurrent connections.

Causal convolution is an elegant solution to this problem. Observe that, in order to apply the chain rule for conditional probability (Eqn. 11), it's important that information never leak backwards in time, i.e. that each prediction be made only using observations from earlier in the sequence. A model with this property is called **causal**. We can design convolutional neural nets (CNNs) to have a causal structure by masking their connections, i.e. constraining certain of their weights to be zero, as shown in Figure 3. At training time, the predictions can be computed for the entire sequence with a single forward pass through the CNN. Causal convolution is a particularly elegant architecture in that it allows computations to be shared between the predictions for different time steps, e.g. a given unit in the first layer will affect the predictions at multiple different time steps.

It's interesting to contrast a causal convolution architecture with an RNN. We could turn the causal CNN into an RNN by adding recurrent

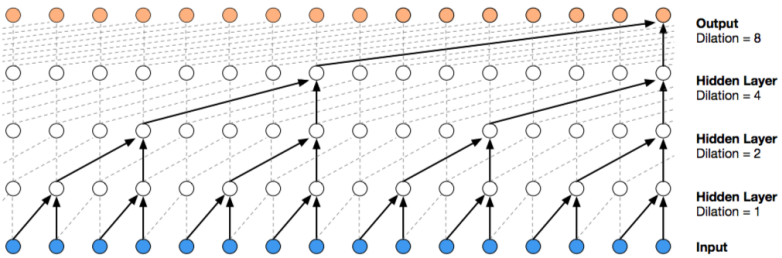


Figure 4: The dilated convolution architecture used in WaveNet. Source: van den Oord et al., 2016, "WaveNet: a generative model for raw audio".

connections between the hidden units. This would have the advantage that, because of its memory, the model could use information from all previous time steps to make its predictions. But training would be very slow, since it would require a `for`-loop over time steps. A very influential recent paper[2] showed that both strategies are actually highly effective for modeling images. Take a moment to look at the examples in that paper.

The problem with a straightforward CNN architecture is that the predictions are made using a relatively short context because the output units have a small receptive field. Fortunately, there's a clever fix for this problem, which you've already seen in Programming Assignment 2: **dilated convolution**. Recall that this means that each unit receives connections from units in the previous layer with a spacing larger than 1. Figure 4 shows part of the dilated convolution architecture for WaveNet[3], an autoregressive model for audio. The first layer has a dilation of 1, so each unit has a receptive field of size 1. The next layer has a dilation of 2, so each unit has a receptive field of size 2. The dilation factors are spaced by factors of 2, i.e., $\{1, 2, \ldots, 512\}$, so that the 10th layer has receptive fields of size 1024. Hence, it gets exponentially large receptive fields with only a linear number of connections. This 10-layer architecture is repeated 5 times, so that the receptive fields are approximately of size 5000, or about 300 milliseconds. This is a large enough context to generate impressively good audio. You can find some neat examples here:

`https://deepmind.com/blog/wavenet-generative-model-raw-audio/`

Compared with other autoregressive models, causal, dilated CNNs are quite efficient at training time, despite their large context. However, all autoregressive models, including both CNNs and RNNs, share a common disadvantage: they are very slow to generate from, since the model's own samples need to be fed in as inputs, which means it requires a `for`-loop over time steps. So if efficiency of generation is a big concern, then GANs or reversible models would be much preferred. Learning a generative model of audio which is of similar quality to WaveNet, yet also efficient to generate from, is an active area of research.

[2] van den Oord et al., 2016, "Pixel recurrent neural networks". `https://arxiv.org/abs/1601.06759`

[3] van den Oord et al., 2016, "WaveNet: a generative model for raw audio". `https://arxiv.org/abs/1609.03499`