6.036: Intro to Machine Learning

Lecturer: Professor Leslie Kaelbling

Notes by: Andrew Lin

Fall 2019

Introduction

Lectures are posted online as well, but a good reason to go to lecture is to ask questions that come up. As discussed in the lab sections, there's a lot of parts to this class. Lectures and notes are meant to give a high-level conceptual view of the new material for the week, and the exercises and homework will be more thoroughly worked examples. Labs are intended to engage us on conceptual questions and ideas that are hard to teach in the homework style: it's for us to engage and talk about the material.

Fact 1
This document will only contain notes from the lectures, not the recitation/lab sections.

1 September 10, 2019

There are many aspects of machine learning, and this class will mostly focus on learning models and taking models of data to make predictions about similar examples. This is super vague, but one of the hardest jobs is to turn a model into a math problem. The way machine learning people work is to turn vague problems into mathematics, and then turn that into computer science. Then those steps need to be tested: hopefully each step of that process is doing the correct things.

Many people have a "romantic spark" when tying to do machine learning: the hope is that they can toss a bunch of data into a model and get things essentially for free. But usually this won't happen: some amount of problem framing and harder work needs to be done first.

Example 2
We can be given a bunch of $\triangle$ and $\square$ shapes, and some of them are positive and some are negative. How can we decide if a new $\triangle$ shape is positive or negative?

This is an **estimation** problem: we basically look at statistics and see how likely it is for the next shape to follow some specific pattern.

Example 3
Let's say $\triangle$ and $\square$ are positive, and $\bigcirc$ is negative. What would an octagon be?

It's reasonable to say it could be positive because it's a polygon, and it's also reasonable to say it's negative because it looks closer to a circle than to the other shapes. This is a **generalization** problem, and it's hard: there are rarely definitely correct answers.

The first kind of problem that we'll be talking about in this class is **classification**, which is a kind of **supervised learning**. Generally, supervised learning is a kind of problem where we are given an initial **data set**

$$D = \left\{ (x^{(1)}, y^{(1)}), \cdots, (x^{(n)}, y^{(n)}) \right\},$$

where the $x^{(i)} \in \mathbb{R}^d$ are vectors of real numbers. In a classification problem, the possible values for $y^{(i)}$ are discrete, and in a **binary classification problem**, we restrict ourselves to $y^{(i)} \in \{+1, -1\}$. Our job in such a problem is to create some kind of **classifier**: we want a function h (stands for hypothesis) which inputs vectors x and outputs a classification y.

If we are trying to classify our data set, how do we decide whether we did a good job? We can try testing it on new data, but we need to be a bit more specific here.

Example 4

If we are trying to predict the height of a group of students based on things like their zip code, we should be careful: the rule h may depend on things like the students' age. So if we train our classifier on a bunch of elementary school kids, we shouldn't expect it to do so well on college students too.

So we often assume that our data set is **iid**, or **independent and identically distributed**. So our rules are supposed to let us perform well on the specific probability distribution we're pulling from! It would be asking too much to train students on calculus problems and then give something else on the exam.

We often use a **proxy**, which is not exactly the problem we're trying to solve, to help us through this process. These perform well on the training data, but it can't be the only thing we do: we can't just memorize our data set, because this won't necessarily generalize. So we can avoid the degenerate answer of "remember data" by **picking our hypotheses from a restricted class**. (For example, hypotheses like "polygon" or "looks like a circle" in Example 3 seem more reasonable than just memorizing the shape.) So we'll restrict our hypothesis $h \in \mathcal{H}$ to be in a specific **hypothesis class**, and the goal is that this will help us generalize.

Definition 5

The **training error** for a hypothesis is defined to be

$$\varepsilon_n(h) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & h^{x^{(i)}} \neq y^{(i)} \\ 0 & \text{otherwise.} \end{cases}$$

Basically, we add an error term each time we misclassify: this tells us how often we get something wrong. Dividing by n is to normalize.

Definition 6

The **testing error** for a hypothesis is the error on a new data set $\left\{ (x^{(n+1)}, y^{(n+1)}), \cdots, (x^{(n+n')}, y^{(n+n')}) \right\}$:

$$\varepsilon(h) = \frac{1}{n'} \sum_{i=n+1}^{n+n'} \begin{cases} 1 & h^{x^{(i)}} \neq y^{(i)} \\ 0 & \text{otherwise.} \end{cases}$$

This is basically equivalent to the training error, but on a new data set. We should expect the testing error to be **bigger** than the training error: generally speaking, our hypothesis is tailor-made for the training data, but it has never seen the new test data before.

We'll mostly be discussing a **supervised learning algorithm** throughout this class: given training data, this algorithm (which we can think of as part of our hypothesis class $\mathcal{H}$), outputs the "best" hypothesis h. For now, "best" means "lowest training error." Usually, this learning algorithm can be made tangible because our hypotheses $h \in \mathcal{H}$ can be described with some fixed-size set of parameters.

Let's try to make this more concrete. Today, we will begin by looking at a very simple class of hypotheses:

Definition 7

The hypothesis class of **linear classifiers** for data points $x^{(i)} \in \mathbb{R}^d$ is dictated by the parameters

$$\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}.$$

(That is, it is specified by $d + 1$ real numbers.) Hypotheses take the form

$$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} 1 & \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise.} \end{cases}$$

The notation $h(x; \theta, \theta_0)$ means that we take the specific function $h(x)$ with the parameters θ and θ_0. Notice that θ and x are both $d \times 1$ vectors, so $\theta^T x$ is essentially a dot product, which spits out a scalar.

So our goal is to take a data set and spit out a θ and θ_0 that fit our data well: what linear classifiers do here is **draw a hyperplane that divides $\mathbb{R}^d$ into two halves.** For example, when $d = 2$,

$$\theta^T x + \theta_0 = \theta_1 x_1 + \theta_2 x_2 + \theta_0.$$

Notice that the boundary (where this expression is equal to 0) is some line. In d dimensions, we have a hyperplane and a normal vector: everything in the half-space of the normal vector is predicted to be positive, and everything in the other half-space is predicted to be negative. This is definitely restricted: we can't make very fancy separators between our positive and negative points, but this is our first step.

So we've basically made this into a math problem: given a dataset D, we want to find θ and θ_0 that minimize $\varepsilon(\theta, \theta_0)$. Our **proxy** here is that we'll try to minimize $\varepsilon_n(\theta, \theta_0)$, the training error.

When we have a new problem, it's nice to think about stupid algorithms: it helps us think about the form of the answer that we want. Here's what we can call a **random linear classifier**:

Algorithm 1: Random linear classifier

```
input  : $D_n$, $k$
output: $\theta, \theta_0$
for $j = 1$ to $k$ do
    sample $(\theta^{(j)}, \theta_0^{(j)})$ from $[-1, 1]^d, [-1, 1]$
end
$j^* = \text{argmin}_{j \in \{1, \cdots, k\}} \varepsilon_n(\theta^{(j)}, \theta_0^{(j)})$;
return $(\theta^{(j*)}, \theta_0^{(j*)})$
```

Basically, randomly pick a bunch of parameters. The **argmin** gives us the index of the best training error, and we just return the θ and θ_0 that correspond to it. k here is called a **hyperparameter**: as it gets larger, the training error is likely to get smaller, but this isn't necessarily the best

Here's a less stupid algorithm: the **perceptron** algorithm, which came in 1962 when people weren't even thinking too much about machine learning:

Algorithm 2: Perceptron
```
input  : $D_n$, $\tau$
output: $\theta, \theta_0$
$\theta = (0, \cdots, 0)^T$;
$\theta_0 = 0$;
for $t = 1$ to $\tau$ do
    for $i = 1$ to $n$ do
        if $y^{(i)}(\theta^T x^{(i)} + \theta_0) \leq 0$ then
            $\theta = \theta + y^{(i)} x^{(i)}$;
            $\theta_0 = \theta_0 + y^{(i)}$ ;
        end
    end
end
return $(\theta, \theta_0)$
```

Basically, if $y^{(i)}$ and $\theta^T x^{(i)} + \theta_0$ have opposite signs, we made a mistake, so we want to edit our θ and θ_0 as indicated. To explain what's going on here, if $y^{(i)} = 1$, we want the dot product of θ and $x^{(i)}$ to be more positive.

Notably, if we go through one iteration of the inner loop, and we don't make any mistakes, we should stop early (because the training error is 0 in this case).

It's not immediately obvious that this converges or does a particularly good job, but it turns out that this algorithm will always find a separator if the data is separable! Meanwhile, if our data isn't linearly separable, our classifier keeps moving around – it turns out taking some kind of "average perceptron" does a pretty good job.

Fact 8
Just because a data set has few points doesn't mean it is easily classifiable! We saw an example where three points took many more steps than a larger set to finish classification.

Next time, we'll start by studying "how long" it'll take for the bouncing around to terminate.

2 September 18, 2019

Lecture notes for this week were transcribed from the Spring 2019 lecture video rather than the Fall 2019 lecture.

Last lecture, we introduced the perceptron algorithm, a way for us to find a linear separator. What's interesting about it is that rather than formulating an objective and trying to come up with an algorithm to reach that objective, the algorithm came first. It turns out that we can prove some helpful properties of how this algorithm works, particularly regarding convergence! This will also give us some additional insight into what's going on: as the semester progresses and we consider more complicated hypothesis classes, our guarantees unfortunately get weaker and weaker.

We'll consider the simple case where we have a perceptron through the origin: in other words, there is no offset θ_0, just a vector parameter θ. To review, here's our algorithm in this simple case: every time we make a **mistake**, we update the value of θ:

```
input  : $D_n$, $\tau$
output: $\theta, \theta_0$
$\theta = (0, \cdots, 0)^T$;
for $t = 1$ to $\tau$ do
    for $i = 1$ to $n$ do
        if $y^{(i)} \cdot \theta^T x^{(i)} \leq 0$ then
            $\theta = \theta + y^{(i)} x^{(i)}$;
        end
    end
end
return $\theta$
```

This algorithm can therefore only distinguish between positive and negative data points with a separator **through the origin**. (We'll come back to what to do in the other cases later).

Definition 9

A dataset D is **linearly separable** through the origin if there exists a θ such that

$$y^{(i)}(\theta^T x^{(i)}) > 0 \quad \text{for all } i.$$

(Linear separability in general also includes the offset term θ_0 inside the parenthetical term.)

In other words, there exists a θ that correctly classifies all points in our dataset D.

Definition 10

The **margin** of a **labeled data point** (x, y) **with respect to** a separator θ is

$$y \cdot \frac{\theta^T x}{||\theta||}.$$

(The numerator also includes a θ_0 term if we don't want to consider only hyperplanes through the origin.)

This often causes a lot of confusion, but the basic idea is that $\frac{\theta^T x}{||\theta||}$ is the signed distance of the data point from the separator, so the margin tells us **how right we are**. If it's negative, we're pretty wrong, and if it's very large, then we're pretty correct.

Definition 11

The **margin** of a **dataset** D with respect to a separator θ is

$$\min_i y^{(i)} \cdot \frac{\theta^T x^{(i)}}{||\theta||}.$$

If all points are classified correctly, then the margin is positive, and it's equal to the margin of the "closest" data point to the separator.

Theorem 12 (Perceptron convergence theorem)
Given a **linearly separable** dataset D, if there exists a θ^* such that

$$y^{(i)} \cdot \frac{\theta^{*T} x^{(i)}}{||\theta||} \geq \gamma \quad \text{for all } i,$$

(the margin of D is at least γ), and all $||x^{(i)}|| \leq R$ (the norms of our vectors are a bounded distance from our origin), then the perceptron algorithm will make at most $\left(\frac{R}{\gamma}\right)^2$ mistakes before converging.

Fact 13
It may feel like changing the units of our data set may change how our algorithm works, but because R and γ both have units of length, they'll scale together and $\left(\frac{R}{\gamma}\right)^2$ will stay constant.

In other words, this tells us that the perceptron does really well if we have a large margin relative to the size of the ball that our vectors are contained in! This is the easiest proof we can do in this class, so let's walk through the main points.

Proof sketch. Since there exists at least one separator with margin at least γ by assumption, let this be θ^*. Let $\theta^{(0)}$ be the zero vector, and let $\theta^{(k)}$ denote θ after k mistakes for all $k \geq 1$.

Our θ will "dance around" θ^*, but we claim that the angle between θ^* and $\theta^(k)$ will get smaller on each iteration. More precisely, we show that

$$\cos(\theta^*, \theta^{(k)}) = \frac{\theta^* \cdot \theta^{(k)}}{||\theta^*||||\theta^{(k)}||}$$

gets bigger as k gets large (and eventually reaches 1). To show this, we'll break it up into two pieces: first of all, by the iterative definition of $\theta^{(k)}$,we can write

$$\frac{\theta^* \cdot \theta^{(k)}}{||\theta^*||} = \frac{\theta^* \cdot (\theta^{(k-1)} + y^{(i)} x^{(i)})}{||\theta^*||} = \frac{\theta^* \cdot \theta^{(k)}}{||\theta^*||} + \frac{y^{(i)} \theta^* x^{(i)}}{||\theta^*||}.$$

But notice that the second term on the right side is the margin of (x, y), and it's at least γ by definition. Thus, by induction,

$$\frac{\theta^* \cdot \theta^{(k)}}{||\theta^*||} \geq k\gamma$$

for all k. So plugging this back in,

$$\cos(\theta^*, \theta^{(k)}) \geq \frac{k\gamma}{||\theta||^{(k)}}.$$

Now we just want to bound the remaining unknown term (it should not necessarily be clear why we're squaring it, because we're going through the proof forward even if the logic would have initially been deduced backwards):

$$||\theta^{(k)}||^2 = ||\theta^{(k-1)} + y^{(i)} x^{(i)}||^2,$$

and now expanding this out, we have

$$= ||\theta^{k-1}||^2 + 2y^{(i)}\theta^{(k-1)} \cdot x^{(i)} + ||x^{(i)}||^2 {y^{(i)}}^2 \leq ||\theta^{k-1}||^2 + 2y^{(i)} x^{(i)} ||\theta^{(k-1)} + R^2.$$

Notably, the middle term is negative, because the algorithm made a mistake on data point $(x^{(i)}, y^{(i)})$. So we have the bound

$$||\theta^{(k)}||^2 = ||{\theta^{(k-1)}}^2|| + R^2 \implies ||\theta^{(k)}||^2 \leq kR^2.$$

Now we're almost done: plugging this back in yields

$$\cos(\theta^*, \theta^{(k)}) \geq \frac{k\gamma}{R\sqrt{k}}.$$

But the cosine of an angle must be at most 1, so

$$\frac{k\gamma}{R\sqrt{k}} \leq 1 \implies k \leq \left(\frac{R}{\gamma}\right)^2,$$

as desired.

□

This theorem is nice, because it even applies when we have an infinite stream of data! As long as we have a sufficient margin, our perceptron will stop making mistakes, which is pretty cool.

So it might seem now like we've been going through a pretty limited case: why should we care about the perceptron-through-origin if we have been working with a more general problem so far?

Proposition 14
We can **reduce** the perceptron problem to a perceptron through the origin by transforming the data.

How can we do this? We'll do something called a **feature transformation**: if our original data set is in $\mathbb{R}^d$, we'll map our vectors via $\phi : \mathbb{R}^d \to \mathbb{R}^D$, where D is usually larger than d. Specifically, consider

$$\phi((v_1, v_2, \cdots, v_d)) = (v_1, v_2, \cdots, v_d, 1).$$

Then if our hypothesis class was originally of the form

$$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0),$$

we can construct an equivalent hypothesis class

$$h(x'; \theta') = \text{sign}(\theta'^T x'),$$

where $x' = (x_1, \cdots, x_d, 1)$ and $\theta' = (\theta_1, \cdots, \theta_d, \theta_0)$.

Example 15
Consider a dataset in 1 dimension where (1) and (2) are classified as positive, while (3) and (4) are negative.

We can separate these via $\theta = (-1), \theta_0 = 2.5$, but we can also separate through the origin by replacing every point (x) with $(x, 1)$. Now $(-1, 2.5)$ corresponds to the line $y = \frac{1}{2.5}x$, which does indeed linearly separate our points.

Remark 16. *The margin γ does change in our* **augmented space**, *and that's something we do have to check. The proof is really a tool of understanding here!*

So now let's generalize some more. The previous example was linearly separable, but what if we don't have this?

Example 17
The data set in 1-dimension where (1) and (-1) are classified as negative and (2) and (-2) are classified as positive is **not linearly separable** (through or not through the origin).

So we want to think of ways to put our data in a different space so that it **becomes** linearly separable. (In fact, we can think of all the layers of a neural network except the last one as just "finding a good transformation.") One possible way to create a new feature space is via the map

$$\phi(x) = (x, x^2).$$

In this new world, we have a 2-dimensional dataset, where $(-1, 1)$ and $(1, 1)$ are classified as negative and $(-2, 4)$ and $(2, 4)$ are classified as positive. This is easy to linearly separate: just take $y = 2$.

In general, there are systematic ways to transform our data into higher-dimensional spaces. Here's one example:

Example 18

A **polynomial basis** of order k consists of all monomials of order at most k. For example, a polynomial basis in 3 dimensions of order 2 is

$$[1, x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1x_2, x_2x_3, x_1x_3].$$

So if we have a problem where the data is a bit complicated, we can try to find a linear separator in a more general space. In the polynomial basis example, any separator will be an equation of the form $p(x_1, \cdots, x_d) \geq 0$, where p is a polynomial in the variables $x_1, \cdots, x_d$ of order at most k. It may not be easy to give a description of this, but we can use this to classify new points by just plugging in the value of $x_1, \cdots, x_d$.

Notice that it can be a lot harder to classify our data points in these spaces, because there are more dimensions for the perceptron algorithm to traverse. In particular, γ gets larger by a bit, but R gets bigger a lot faster! Thus, our perceptron convergence theorem tells us a much weaker result.

Question 19. *Is there a difference between transforming classifiers into higher dimensions and our data points into higher dimensions?*

The answer is no: as soon as the data points are put into a higher-dimensional space, the classifier must also be put in a higher-dimensional space (just because we're working with linear separators here). However, as the shapes are allowed to get funkier, we are at risk of **overfitting**, which will be a theme that comes up often in this class. (In particular, the algorithm may do badly with extra noise thrown in.)

There's a flip side to all of this, though. We've been working with vectors and numbers throughout our discussion of this algorithm, but how exactly do we **encode our real-world data** into a vector to feed into our algorithm?

Example 20

We could be working with the number of cylinders of a car, or the manufacturer of a phone, or the third digit of a Social Security Number. Our question: **what are some ways that we can encode those features?**

Here's some options:

- Treat our feature as being a numerical value in $\mathbb{R}$. This makes sense for something like the number of cylinders in a car (saying something like $2 > 1$ and that the distance from 3 to 2 is the same as the distance from 2 to 1 makes sense), but not for something like a car manufacturer (because there's no inherent number scale).
- Use a **one-hot** encoding. If there are m possible values for our discrete variable, encode the first value as $(1, 0, \cdots, 0)$, the second as $(0, 1, \cdots, 0)$, and so on. This is good because it can indicate **presence** of a feature.
- Potentially, represent with binary numbers to save space (because we can use less bits). **This is a very bad idea!** Coding our variable in binary means that our algorithm needs to decode the binary, which just makes things more

difficult. So compression makes things smaller but harder to interpret, and that's exactly what we don't want our algorithm to work with.

Another key idea is that we can break up a feature by **factoring**.

Example 21

Consider a list of blood types like A+, A-, B+, B-, AB+, AB-, O+, and O-. Out of these, we could do a one-hot encoding, but it may be helpful to separate some aspects here: doing a one-hot coding on (A, B, AB, O) and another one on (+, -) may be better. We can even factor this into (A, no A), (B, no B), and (+, -).

So if we know something about what our input data means, **interpreting that for our learning algorithm can make a difference in how well it performs.**

Finally, one note about numbers: if different variables are of different magnitudes, we often like to **standardize numerical features** so that some dimensions don't have way larger values than others. This means we replace

$$[\tilde{x}_j \to \frac{x_j - \overline{x}}{\sigma_x},$$

which means that each variable has mean 0 and standard deviation 1. This is nice because it puts all of our variables on equal footing, making the learning algorithm potentially less biased.

3 September 24, 2019

In today's class, we're going to keep talking about classification, but we'll introduce a way of thinking about algorithms with the following basic underlying idea: come up with machinery that lets us derive machine learning algorithms for arbitrarily propagated problems. This will be the foundation of much more complicated learning methods later on in this class!

Basically, we'll turn machine learning problems into optimization problems: taking a function and using computational methods to find its minimum or maximum.

Definition 22

The **objective function**, often denoted $J(\Theta)$, is a measurement of "how well we're doing:" it's the quantity we're trying to optimize with respect to Θ.

Here, Θ refers to all parameters at once: for example, in linear separators, it refers to both θ and θ_0. Our goal is then to find

$$\Theta^* = \text{argmin}_\Theta J(\theta) :$$

basically, find the best possible parameters to minimize $J(\theta)$.

Most of our functions we study in this class will look like the following:

$$J(\Theta) = \left(\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(h(x^{(i)}, \Theta), y^{(i)})\right) + \lambda R(\Theta).$$

Here, $\mathcal{L}$ is a loss function: it tells us how sad we are that we made the guess $h(x^{(i)}, \Theta)$ when the actual value was $y^{(i)}$. So the whole first term here is the **training error**, because it judges our performance on the training data set. Meanwhile, the second term $\lambda R(\Theta)$ is called the **regularizer**: this basically helps us constrain the values of Θ so that the function doesn't try "super hard" to fit the data. (This avoids overfitting, among other things.) $\lambda > 0$ here is a constant: it's a knob that we can turn, also known as a **hyperparameter**. Having control over this number lets us decide how heavily we care about fitting the data versus the properties of Θ.

Today, we'll study **linear logistic classifiers**, also known as **logistic regression**. Recall that we've been working so far with **0–1 loss**: the loss function

$$\mathcal{L}(y, a) = \begin{cases} 1 & y \neq a \\ 0 & y = a \end{cases}$$

just counts the number of points that we classify wrong. Recall that the perceptron algorithm does minimize this loss when our data is linearly separable, but it doesn't promise anything in general.

So perhaps we should come up with an algorithm that minimizes 0–1 loss. Unfortunately, this is actually very computationally difficult to find! (In fact, we currently don't know any fast methods for doing this at all, because the problem is NP-hard.)

If we want to solve this problem, the idea is that we want to make our optimization problem less difficult, without changing the actual core ideas of our algorithm too much.

Example 23

Let's consider 1-dimensional space now, so each data point is a single $+$ or a $-$ on the number line. Then our linear separator is just a single point, and our loss function $\mathcal{L}(x)$ is basically a step function: it jumps by 1 (or -1) when x crosses one of our data points.

The motivation is that we want to make our curve more smooth! Instead of a **sign function** $\text{sgn}(\theta^T x + \theta_0)$, we'll use the **sigmoid curve**

$$\sigma(\Theta^T x + \theta_0), \quad \text{where } \sigma(z) = \frac{1}{1 + e^{-z}}.$$

This is "kind of like the sign function," but it's smooth: notice that it is $\frac{1}{2}$ at $z = 0$, close to 1 for large z, and close to 0 for small z. This means that we have a way to see "how wrong" our prediction is on each data point. Soon, we'll interpret this value as a probability: it gives us our predicted probability that a data point x is supposed to be positive.

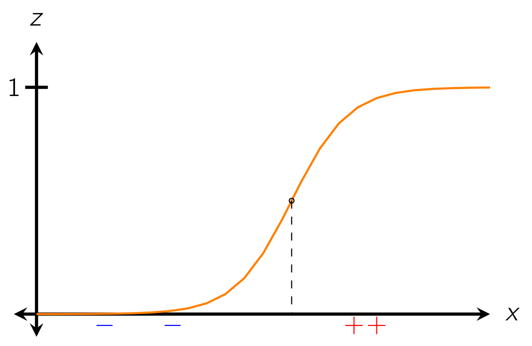
To make this into a classifier, we need to specify a **threshold**: for instance, we can say that we predict 1 if

$$\sigma(\theta^T x + \theta_0) > 0.5,$$

which exactly corresponds to $\theta^T x + \theta_0 > 0$.

Remark 24. *This threshold may be different: for example, if false positives are very bad, we may have to set the threshold higher.*

So this is indeed the same hypothesis class, but we're just setting things up differently to make the optimization easier and application more useful. Here's an example of what this may look like:



In one dimension, notice that the θ-values can shift our sigmoid curve, flip it around, make it steeper, and so on. It's a bit harder to visualize this function in two dimensions, but we can think of σ as intersecting our linear separator at 0.5.

So how do we derive a loss function here? We're going to think of it probabilistically: our classifier should assign "high probability" to our specific data set if it is good. Let's label our points with 1 and 0 (± 1 works too). If we define our **guess**

$$g^{(i)} = \sigma(\theta^T x^{(i)} + \theta_0),$$

then consider the quantity

$$P = \prod_{i=1}^{n} \begin{cases} g^{(i)} & y^{(i)} = 1 \\ 1 - g^{(i)} & y^{(i)} = 0 \end{cases}.$$

Recall that $g^{(i)}$ gives us the probability that we think $y^{(i)}$ should be positive, so if $y^{(i)} = 1$, we're happy with a factor of $g^{(i)}$. However, if $y^{(i)} = 0$, then we actually want the complement of that probability. Then we multiply all of those together, because the probability that we're correct on all of our points is the product (since we have independent events).

So P is a reasonable thing to try to maximize. But products are hard to deal with in optimization (for example, multiplying a lot of small numbers can do bad things, and also taking derivatives of products is disgusting). So we'll take the logarithm $\log(P)$ of this product, which is monotonic in P, so the argmax will be the same as the original objective function.

A few more details: notice that we can write the quantity above as

$$P = \prod_{i=1}^{n} {g^{(i)}}^{y^{(i)}} \cdot (1 - g^{(i)})^{(1-y^{(i)})},$$

since the two exponents are going to be 0 and 1 in some order. (It's good for us to convince ourselves that this is true.) Taking the logarithm of both sides,

$$\log P = \sum_{i=1}^{n} \left(y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log(1 - g^{(i)}) \right),$$

Notice that this term is indeed $\log g^{(i)}$ when $y = 1$ and $\log(1 - g^{(i)}$ when $y = 0$, so the if statement is still hidden in here. So because we want our loss function to be minimized instead of maximized, **we'll negate this quantity**. This gives us a final expression of

$$\mathcal{L} = \sum_{i=1}^{n} \mathcal{L}_{\text{NLL}}(g^{(i)}, y^{(i)})$$

(NLL here stands for **negative log likelihood**), where the negative log likelihood between a guess g and a value y

$$\mathcal{L}_{\text{NLL}}(g, y) = -(y \log g + (1 - y) \log(1 - g)).$$

(This may also be referred to as a **log loss** or **cross entropy**.) It's a good exercise to think about how this looks similar to our 0–1 loss!

So we have our objective function now (let's ignore the regularizer for now). We need to be smart about how to optimize, and we're going to use one of the simplest possible optimization algorithms: **gradient descent**.

The main idea here is that if we're trying to optimize a function f, we compute the derivative at our current point x, and move in the direction that makes the function smaller. Here's the **one-dimensional gradient descent algorithm**: basically, at each step, we move along the direction of "descent" with a step size of η, until our guess is sufficiently good.

Algorithm 3: One-dimensional gradient descent

```
input  : $\theta_{\text{init}}$, $\varepsilon$, $\eta$, $f$, $f'$
output: $\theta$
$\theta^{(0)} = \theta_{\text{init}}$;
$t = 0$;
while $t = 0$ or $|f(\theta^{(t)}) - f(\theta^{(t-1)})| < \varepsilon$ do
    $t = t + 1$ ;
    $\theta^{(t)} = \theta^{(t-1)} - \eta f'(\theta^{(t-1)})$ ;
end
return $\theta$
```

So how well does this function work? If we have a bumpy function with many local minima, we might terminate at a local instead of global minimum! But we can guarantee at least some results:

Theorem 25

If f is convex, then for any desired accuracy ε, **there exists some** η such that gradient descent converges to θ within ε of the optimal f.

The fundamental idea is that if we pick η to be very small, we take very small steps, and it takes a long time to finish the algorithm. But if η is very large, we take enormous steps, and our algorith may not converge. So we should play around with this and get a feel for what's going on here, and it's not always clear what η should be.

Gradient descent in general (for more than one dimension) looks very similar, but the idea is that our derivative behaves a little differently.

Definition 26

The **gradient** of a function $f(\theta)$, where $\theta = (\theta_1, \cdots, \theta_m)$, takes on the form

$$\nabla_\theta f = \begin{bmatrix} \partial f/\partial\theta_1 \\ \partial f/\partial\theta_2 \\ \vdots \\ \partial f/\partial\theta_m \end{bmatrix}.$$

We can think of this as being the "steepest direction up the hill:" the direction points in the direction of largest positive change. Then we just keep trying to go down the hill, and everything else looks the same:

Algorithm 4: Gradient descent

```
input  : $\theta_{\text{init}}$, $\varepsilon$, $\eta$, $f$, $\nabla_\theta f$
output: $\theta$
$\theta^{(0)} = \theta_{\text{init}}$;
$t = 0$;
while $t = 0$ or $|f(\theta^{(t)}) - f(\theta^{(t-1)})| < \varepsilon$ do
    $t = t + 1$ ;
    $\theta^{(t)} = \theta^{(t-1)} - \eta\nabla_\theta f(\theta^{(t-1)})$ ;
end
return $\theta$
```

Notice here that even if we have our objective function f, we're still going to need to find the gradient of f, and this is usually done by hand.

As a separate note, it's generally good to make sure that the algorithm terminates for a reason other than the ε condition, in case our step size is very bad. As shown in class, if we use the one-dimensional example in the graph above, the logistic curve becomes steeper and steeper, because this allows us to minimize loss even after we've correctly classified all of our points.

In general, we can see that the logistic regression tends to give us better (more representative) separators, even though both this and the previous example are linear separators. In some sense, the logistic regression is better at optimizing our margin.

One final note about the regularizer: we often set $R(\Theta) = ||\Theta||^2$. This generally keeps our optimization from going nuts: we can't let the weights get too large, and this may also help us deal with outliers. We'll cover more about this later on.

4 October 1, 2019

So far, we've looked at linear separators of different kinds: we've been doing this by using the perceptron algorithm, or by using gradient descent to find a logistic linear classifier. We'll still be doing supervised learning now, but we'll move from classification to a problem called **regression** (which doesn't mean moving backwards). The central idea is that we're going to look at hypotheses of the form $\mathbb{R}^d \to \mathbb{R}$ (our target space is now a real number rather than a discrete set like $\{\pm 1\}$).

As a reminder, the idea behind supervised learning is that we're given a set of data points

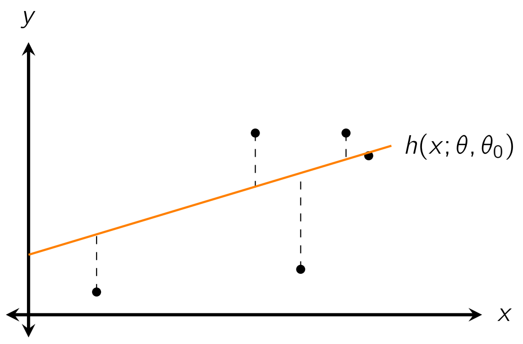
$$\{(x^{(i)}, y^{(i)}) : x^{(i)} \in \mathbb{R}^d, y^{(i)} \in \mathbb{R}\}.$$

Our goal will be to pick a hypothesis class and a loss function to go with it. Let's start by making this hypothesis class $\mathcal{H}$ reasonably simple:

$$\boxed{h(x; \theta, \theta_0) = \theta^T x + \theta_0}.$$

(Note that the difference here is that we're not taking the sign or sigmoid of this value: we are legitimately using it as our output). This is called a **linear regressor**: the hypothesis will look like a hyperplane.

Then what will our loss function be? Here's a picture of what our linear regressor might look like, where the dashed lines represent the errors from our prediction.



Notice that this linear regressor gives us a guess for y given any value of x. So if we predict a guess g when the answer is actually a, what can our loss function be? Something that looks like

$$L(g, a) = (g - a)^2$$

might make sense. This is called **squared loss**: a reasonable alternative is to consider $|g - a|$, which is called the **L^1 loss**, but the justification for using that is a bit subtle. Plus, squaring the deviations means that we'll penalize extremely wrong predictions more heavily! So now we have our loss function:

$$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n}(\theta^T x^{(i)} + \theta_0 - y^{(i)})^2.$$

(Remember that the first two terms in the parentheses are our guess, and $y^{(i)}$ is the actual answer.) This is called the **ordinary least squares** regression model. It turns out that this is a particularly nice model because we can solve it analytically.

Before we do that, though, let's write down the problem as a matrix problem: this makes our derivations shorter, and it makes the code easy to think about.

Definition 27

Assume that the θ_0 corresponds to the final row of our θ vector, just like we did with the perceptron algorithm. Define the d by n matrix X and the 1 by n matrix Y via

$$X = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \cdots & x_d^{(n)} \end{bmatrix}, \quad y = \begin{bmatrix} y^{(1)} & \cdots & y^{(n)} \end{bmatrix}.$$

We're going to take some transposes of these matrices so that they agree with what we might see in the literature: defining

$$W = X^T, T = Y^T,$$

we get an $n \times d$ and $n \times 1$ matrix, respectively. Given these matrices, we can rewrite our objective function J:

$$\boxed{J(\theta) = \frac{1}{n}(W\theta - T)^T(W\theta - T)}.$$

(This is a dot product of the vector $(W\theta - T)$ with itself.) We can check that all of the dimensions make sense here: we're dotting an $n \times 1$ vector by itself. What's going on here is that the kth element of $W\theta - T$ corresponds to the error of our kth guess, and then dotting with itself gives us the squared loss that we're looking for.

So if we want to minimize this, we take the derivative and set it equal to zero. One way to approach this is to take the derivative with respect to each θ_i, which gives d equations and d unknowns.

But instead, we can take the **gradient of the loss function with respect to** θ. This should have dimensions $d \times 1$, the same as θ: doing out the calculations, we find that

$$\nabla_\theta \cdot J = \frac{2}{n}W^T(W\theta - T).$$

If we forget that this is a derivative, this kind of looks like a plausible answer: we just go through and use the chain rule. The only problems are having to know what order multiplication works in, and also how to do transposes. Unfortunately, the best way is probably just to match up the dimensions.

How can we check to make sure it's correct? The kth element of the gradient is $\frac{\partial J}{\partial \theta_k}$, and we can work through the vector partial derivatives with a pencil.

What's nice is that this turns out to be a convex function (which we can check on our own). Setting this gradient equal to zero,

$$W^T W\theta = W^T T.$$

Multiplying by the relevant inverses, we get

$$\theta = (W^T W)^{-1} W^T T.$$

So if we can just invert a matrix, we can find the optimal θ directly! There aren't really any other machine learning problems where we can just directly write down the answer.

Note, though, that regression doesn't do such a great job with nonlinear data. We can use a fancier hypothesis class in general, or we can use a feature transformation to send $\phi(x)$ to y (where the xs might be put in a polynomial basis, or something else like that). For example, a second order polynomial feature representation means we're taking

$$\text{Span}(1, x, x^2) \to y,$$

which is really just fitting a quadratic curve to our data.

Remark 28. *We need to be careful not to overfit, though. If we have n data points in one dimension, an $(n-1)$-degree polynomial will be able to go through all of them (as long as we don't have two points with the same x-coordinate).*

Remark 29. *When we're coding, we might also get some messages like "**singular matrix**," which likely means that the matrix $W^T W$ is not invertible. This often happens when $d > n$: that basically means we don't have enough data to define our parameters in a good way. It also may happen that we have extra data, or "only barely" invertible matrices. Then numerically, the computer may not be able to represent the situation we're trying to work with!*

So to fix some of the issues that may come up, we're going to add a regularization term. The vocabulary used here is **ridge regression**: we're adding a penalty on the magnitude of θ. That usually looks like

$$J_{\text{ridge}}(\theta) = \frac{1}{n}(W\theta - T)^T (W\theta - T) + \lambda ||\theta||^2.$$

(Remember that we can always write $||\theta||^2 = \theta^T \theta$.) We've made our problem harder here, but luckily there's still an analytic solution: it simplifies to

$$\theta_{\text{ridge}} = (W^T W + n\lambda I)^{-1} W^T T.$$

Remember that this θ_{ridge} is the θ that minimizes the loss function. I here is the identity matrix: notice that if $\lambda = 0$, this reduces to the normal case. What's nice about this form is that if we perturb λ by a little bit, we get some extra "positive stuff" all along the matrix: **that contribution will often make $W^T W$ invertible**. (This is because the row and column vectors become no longer linearly dependent.)

Fact 30

To pick the correct value of λ, we might want to do something like cross-validation. Intuitively, a small λ is good if we have dense enough data to know what the shape of our curve looks like: we can't overfit as easily. So we basically just try small and big λ: our goal is to have good generalization in applications, rather than fitting super well to our training data. Notice that a very big λ will make our coefficients get smaller, since $\theta \approx 0$ is the right answer in that case.

So why are we penalizing large θs here? One way to think about this is that when θs are big, changing a data point by a little bit can change the prediction by a lot. This makes our prediction θ unstable, which is not good; **stability implies good prediction.**

So let's say we've run our algorithm and need to use it on test data. There's two main reasons why we might have large test error:

- **structural error**: our hypothesis class doesn't actually represent the data well. For example, maybe we're trying to find linear or polynomial regressors for data that's actually tracing out a circle: then we're basically doomed to fail. Also in this category, if we make our regularization term too strong, then we overconstrain the values of θ that we can use.
- **estimation error**: we just don't have enough data to reliably estimate our parameters (in other words, pick a good hypothesis out of our hypothesis class).

These are also called "bias" and "variance" respectively, but these terms also have definitions in other areas of machine learning. One term that we may see is the **bias-variance tradeoff**, though (which is what it sounds like).

We'll finish this class by talking a bit more about optimization. We can always give numpy our data matrix directly and have it calculate that matrix $(W^T W + n\lambda I)^{-1} W^T T$, but this fails at a point (when the matrix gets too big). Luckily, we also have the other tool in our toolkit: gradient descent. We're lucky here that both objective functions we talked about today are convex, so there **exists a step size** that will help us converge to the correct answer.

What we'll be doing is often called **batch gradient descent**. Remember that the gradient here is the sum of a bunch of terms (corresponding to the errors of the individual data points). If our objective function can be written as

$$f(\theta) = \sum_i f_i(\theta),$$

as it has been for most of our applications, we can do a **stochastic gradient descent**:

Algorithm 5: Stochastic gradient descent

```
input  : $\theta_{\text{init}}$, $\nabla_\theta f$, $\eta$, $T$
output: $\theta$
$\theta^{(0)} = \theta_{\text{init}}$;
for $t = 1$ to $T$ do
    randomly pick $i \in \{1, 2, \cdots, n\}$;
    $\theta^{(t)} = \theta^{(t-1)} - \eta(t) \cdot \nabla_\theta f_i(\theta^{(t-1)})$;
end
return $\theta$
```

Basically, we randomly pick any data point and move in the direction that the gradient tells us to move in. This may seem objectively worse than our ordinary gradient descent, but let's think about this a bit: one idea is that if we're stuck in a very shallow optimum for the whole function, we might jump out of it if we only need to look at one point at a time. Also, if we have a ton of data, each iteration can be super cheap! We don't need to look at every point every single time to get a pretty good answer.

This isn't going to optimize perfectly, but in a way, this is another way to "regularize" our function. We don't allow ourselves to over-optimize if we have this extra element of randomness.

Notice that in the above algorithm, η is now a function of t. The reason we like this is because of the following:

Theorem 31

If J is a convex function and $\eta(t)$ satisfies

$$\sum_{t=1}^{\infty} \eta(t) = \infty \text{ but } \sum_{t=1}^{\infty} \eta(t)^2 < \infty,$$

then stochastic gradient descent will converge to the minimum with probability 1 (not necessarily always, but with very high likelihood).

Basically, there are two main reasons we might not converge: we might flail around too much, or we might stop before we are able to get to the minimum. These conditions that we place on our step size η prevent either of those cases from happening.

Example 32

Notice that $\eta(t) = \frac{c}{t}$ satisfies the conditions that we want, because $\sum \frac{1}{n}$ diverges but $\sum \frac{1}{n^2}$ does not. There's not really a lot of science behind picking a good starting step size: that's mostly dependent on the specific situation!

So does our stochastic gradient descent work for ridge regression? We have to rewrite it a bit:

$$\frac{2}{n}\left[\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)} x^{(i)}\right] + 2\lambda\theta,$$

which we can then rewrite as an sum over i:

$$= \sum_{i=1}^{n}\left[\frac{2}{n}\theta^T x^{(i)} - y^{(i)} x^{(i)} + \frac{2}{n}\lambda\theta.\right]$$

Then we have an objective function $f(\theta) = \sum_i f_i(\theta)$, so then we can go ahead and apply stochastic gradient.

5 October 8, 2019

It's time for neural networks! We've been looking at simple models, and we know how to train a model like a linear separator for classification. Now, we'll do a more complicated model: there will be more parameters, and we'll just do a slightly more complicated version of gradient descent. Basically, we're scaling the whole problem up.

Neural networks are often made up of **neurons** or **units** which look something like this:

$$\begin{tikzcd}
x_1 \arrow[dr] & & & \\
x_2 \arrow[r] & \Sigma \arrow[r, "z"] & f \arrow[r, "a"] & {} \\
x_3 \arrow[ur] & & & \\
& w_0 \arrow[uu] & &
\end{tikzcd}$$

The idea is that we have inputs that come in, each multiplied by some weight w_i, and then we add an additional weight w_0. Neurons take in a vector $x \in \mathbb{R}^m$ (unfortunately, we do have to change notation here), and a weight vector $w \in \mathbb{R}^m$: this outputs the **pre-activation**

$$z = w^T x + w_0$$

(should look fairly familiar at this point). This value of z then gets plugged into an **activation function** f, outputting some **activation** $a = f(z)$. (The point of this last part is usually to add

Example 33

Linear and logistic regression are both specialized examples of this neuron (logistic regression has a sigmoid function for f, for example).

The idea is that in biology, neurons will output some kind of pulse (corresponding to a) if the input is large enough (corresponding to z). This model we have is definitely not biologically accurate. But it's a nice metaphor, and it works well at the computational level.

Well, a is our guess, so if we have a training data set with values of y, we know how to minimize the loss function with gradient descent, and that's what we've already seen. What's new is when we start adding multiple neurons!

Definition 34

A **fully-connected layer** is a set of n parallel units: we have m input values $x_1, \cdots, x_m$, each connected to the n Σs (though with different weights). Each of the n units outputs a value, which means that this layer outputs an n-dimensional vector when we give it an m-dimensional input.

Layers are way easier to deal with because it gives a better system for computation. We'll be dealing first with **feed-forward** neural networks: basically, our graph does not have any loops, so inputs early on cannot depend on later outputs.

We can represent the weights of a fully-connected layer with matrices: W, an $m \times n$ matrix, represents the weights going into each unit, and W_0, an n-dimensional vector, represents the w_0 offsets for each unit. Then the outputs follow the same form:

$$Z = W^T X + W_0.$$

Indeed, this is an $n \times 1$ column vector, and the units work out. And then if we apply f component-wise to our pre-activation vector z, we get our output

$$A = f(Z).$$

So here's what a many-layered neural network might look like:

$$\begin{tikzcd} {} \arrow[r, "x", "A^0"'] & \boxed{\begin{matrix}W^1\\ W_0^1\end{matrix}} \arrow[r, "z^1"] & \boxed{f^{(1)}} \arrow[r, "A^1"] & \boxed{\begin{matrix}W^{(2)}\\ W_0^{(2)}\end{matrix}} \arrow[r, "z^2"] & \boxed{f^{(2)}} \arrow[r, "A^2"] & \cdots \end{tikzcd}$$

So a neural network is a parametric function that takes in a vector of inputs and gives a vector of outputs. It's differentiable with respect to those parameters, too: what exactly are our parameters here? If we have L layers, then

$$\Theta = \{(W^1, W_0^1, \cdots, W^L, W_0^L)\} :$$

every element of W^i and W_0^i is something that we get to fine-tune when training our model. Do note, though, that we get to pick (by design) the number of layers L, the number of units in each layer n^i, and the kind of nonlinearity f that we use at each layer.

This model, ultimately, is still like everything else we do: it is possible to write down our hypothesis

$$h(x) = NN(x : \theta),$$

and then we could take the gradient and do things as we have been told to do so far. But computing the gradient is hard in a way: let's try to make it nice algorithmically! In particular, let's do it in a generic way so that it works for any number of layers and units per layer and nonlinearities.

The idea is that we'll think about computing the gradient ∇ for each individual component in our W matrices and W_0 vectors. If we can compute the gradient for one data point x, then we can do stochastic gradient descent (or we can just sum up the gradient over all points x and do it in batches).

Let's start at the last layer. How can we find the gradient of the loss with respect to W^L? Let $\mathcal{L}$ be the loss function for a particular data point $x^{(i)}$ if we were supposed to have $y^{(i)}$ as our answer:

$$\mathcal{L} = \text{loss}(NN(x^{(i)}; \theta), y^{(i)}).$$

So now we use the chain rule: informally, it looks like

$$\frac{\partial \mathcal{L}}{\partial W^L} = \frac{\partial Z^L}{\partial W^L} \cdot \frac{\partial \mathcal{L}}{\partial Z^L}.$$

(Yes, these are matrices, but the form of the chain rule looks the same: we'll fix up the details later. Yes, the order does matter, but we'll think about that too.) Note that this works for any layer, so we can also replace L with any ℓ.

To deal with the first term, note that

$$Z^\ell = {W^\ell}^T A^{\ell-1} + W_0^\ell,$$

where $A^\ell = f(Z^\ell)$. So if we just relax a bit, the derivative of Z with respect to W is just the previous A: this does work out if we're careful. On the other hand, computing the second term $\frac{\partial \mathcal{L}}{\partial Z^\ell}$ is a recursive computation: we just go back one layer at a time by repeatedly using the chain rule!

$$\frac{\partial \mathcal{L}}{\partial Z^\ell} = \frac{\partial \mathcal{L}}{\partial A^L} \frac{\partial A^L}{\partial Z^L} \frac{\partial Z^L}{\partial A^{L-1}} \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdots \frac{\partial Z^{\ell+1}}{\partial A^\ell} \cdot \frac{\partial A^\ell}{\partial Z^\ell}.$$

So generally speaking, taking a product of these partials as we go along will give us the loss function with respect to any variable that we want! So now let's just go back and make all of the transposes work out.

Okay, so what are these partial derivatives? Starting from the first term, $\frac{\partial \mathcal{L}}{\partial A^L}$ just depends on the specific loss function we're trying to do, and it's an $n^L \times 1$ vector.

Next, **we can verify that** the terms of the form $\frac{\partial Z^\ell}{\partial A^{\ell-1}}$ correspond to $m^\ell \times n^\ell$ matrices:

$$Z^\ell = {W^\ell}^T A^{\ell-1} + W_0^\ell \implies \frac{\partial Z^\ell}{\partial A^{\ell-1}} = W^\ell.$$

Finally, A^ℓ depends on Z^ℓ via our activation function f, and the function f is applied component wise. So the derivative is just an $n^\ell \times n^\ell$ square diagonal matrix, where the ith entry is just $f'(z_i^\ell)$.

And now we can finally write out what we're looking for: to get the dimensions to all work out, we write it as

$$\boxed{\frac{\partial \mathcal{L}}{\partial Z^\ell} = \frac{\partial A^\ell}{\partial Z^\ell} \cdot W^{\ell+1} \cdot \frac{\partial A^{\ell+1}}{\partial Z^{\ell+1}} \cdots W^L \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \mathcal{L}}{\partial A^L}}.$$

Z^ℓ is an n^ℓ by 1 vector, and the loss is a scalar, so this corresponds to the gradient of our loss. We can check that the dimensions on the right hand side work out, and if we just multiply this with $\frac{\partial Z^\ell}{\partial W^\ell}$, this gives us the gradient of the loss with respect to the weights, as desired.

Remark 35. *This is known as* **error backpropagation**, *because we start by computing the Zs and the As from front to back. Then to calculate the gradient, we go backwards, finding the effect of each Z^L and A^L in turn on our loss to figure out where the "blame" lies.*

So let's write out our algorithm:

Algorithm 6: Neural network stochastic gradient descent

```
input  : $D_n, T, L, (n^1, \cdots, n^L), (f^1, \cdots, f^L), (f^{1\prime}, \cdots, f^{L\prime})$
output: $W^1, \cdots, W^L, W_0^1, \cdots, W_0^L$
$W_{ij}^\ell \sim$ Gaussian$(0, 1/n^{\ell-1})$;
$W_0^\ell \sim$ Gaussian$(0, 1)$;
for $t = 1$ to $T$ do
    pick $i$ at random and let $A^0 = x^{(i)}$;
    for $\ell = 1$ to $L$ do
        $Z^\ell = {W^L}^T A^{\ell-1} + W_0^\ell$;
        $A^\ell = f(Z^\ell)$;
    end
    for $\ell = L$ to $1$ do
        $\frac{\partial \mathcal{L}}{\partial A^\ell} = \frac{\partial \mathcal{L}}{\partial Z^{\ell+1}} \frac{\partial Z^{\ell+1}}{\partial A^\ell}$ (except when $\ell = L$, in which case it's $\frac{\partial \mathcal{L}}{\partial A^L}$);
        $\frac{\partial \mathcal{L}}{\partial Z^\ell} = \frac{\partial \mathcal{L}}{\partial A^\ell} \cdot \frac{\partial A^\ell}{\partial Z^\ell}$;
        $\frac{\partial \mathcal{L}}{\partial W^\ell} = \frac{\partial \mathcal{L}}{\partial Z^\ell} \cdot \frac{\partial Z^\ell}{\partial W^\ell}$;
        $\boxed{W^\ell = W^\ell - \eta(t) \frac{\partial \mathcal{L}}{\partial W^\ell}}$;
    end
end
return $W^1, \cdots, W^L, W_0^1, \cdots, W_0^L$;
```

The first step here is that we need to start off at a non-standard place in our space: **initialize everything randomly**. Remember that if we have really large magnitudes in something like our sigmoid function, we have derivative 0 (which is not so good). So we need to normalize in a way so that our weights aren't quite so big: that explains the $\frac{1}{n^{\ell-1}}$ term.

Then the loop does two things: first, we go forward and find our values of Z^ℓ and A^ℓ. Then we go backwards, compute the necessary derivatives, and finally do an update step at the end (the boxed term).

So what should we know about the function f? One key idea: **if f is the identity or any linear function, then the whole neural network is no better than just having one layer.** This is because the composition of two linear functions is linear, so we're essentially just using one matrix.

Also, the sigmoid function was the standard choice of nonlinearity (for f) for a long time, but this has now shifted to the **rectified linear unit**

$$\text{relu}(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$

It's not exactly differentiable at $z = 0$, but this is not too worried about: we can just define it to either be 0 or 1 and this won't be too bad.

Remark 36. *Note, though, that our last activation function f^L does need to take a special form: for example, if we're doing regression, we want it to be the identity, and if we want it to be a classification, we use something like the sigmoid function.*

What this basically does is chop off the negative part. This is really nice because the derivatives are really nice to calculate, and they're not so sensitive to the weights (except that we still need to initialize them randomly).

6 October 22, 2019

We're now in the middle of the "basic section" of neural networks: last time, we worked through the idea of defining a neural network in terms of a bunch of layers, and we saw that this is basically just a giant parametric function between our inputs and outputs. We found a general method (backpropagation) to compute gradients: as a warning, this week's homework is pretty long, and it has to do with working through the coding of backpropagation.

But it turns out that getting neural networks to work is a bit of an art, so we'll talk about some of the extra strategies and tricks! (In fact, there's a whole area of research which does an automated search over the structure to find the best one.) Here, we'll focus on two main ideas: training and regularization.

Remember that given a dataset, we want to minimize our objective function

$$J(w) = \sum_{i=1}^{n} \mathcal{L}(h(X^{(i)}, W), y^{(i)})$$

and the traditional **batch gradient descent** tells us to change our weights via (for some η)

$$W = W - \eta \sum_{i=1}^{n} \nabla_W \mathcal{L}(h(x^{(i)}, W), y^{(i)}),$$

where we are summing over our entire data set. We've also seen the other extreme: in **stochastic gradient descent**, we replace the sum from 1 to n with a random index i (that is, we update based on one data point at a time). Batch gradient descent is good, because it tells us exactly which direction we want to go in each time. Stochastic gradient descent, on the other hand, takes us in a slightly crazy direction, but we know that this works out on average (and often takes much less time than batch gradient descent).

We can make a compromise between these two:

Definition 37
In **mini-batch**, we select k of our points at random and just sum the gradients of the loss over those points.

The value of k depends on the curve that we're using it on: there's no exact formula for this. Mini-batch is usually set up in any libraries we happen to be using in code.

Remark 38. *It also turns out that our hardware can parallelize multiple data points pretty well, so we don't really have to put in one point at a time: we can still have our data in matrix form.*

Remark 39. *Also, there really should be a $\frac{1}{n}$ in front of the objective function and gradient sum term, but people are often lazy because the constant factor can be absorbed into the η anyway.*

We played around with some simple values of η in one dimension, but picking η is very difficult in general. In fact, there really isn't any good single value: remember the expression

$$\frac{\partial mcL}{\partial W^1} = W^1 \frac{\partial A^1}{\partial Z^1} W^2 \frac{\partial A^2}{\partial Z^2} W^3 \cdots W^L \frac{\partial A^L}{\partial Z^L} \frac{\partial \mathcal{L}}{\partial A^L}.$$

If our weights are generally larger than 1, the derivatives will get very big! This is essentially exponential in the depth of our network, so the magnitude of our derivatives differ very much. Finding a single value of η that works for the whole network is thus quite hard. Having a human do this is annoying, so let's think about algorithms that can do it: this is the idea of **adaptive step sizes**. We'll also have a step size for each weight, so we need to define η_{tw} for all t and **all weights** w (not just per layer, either.)

Notice here that if we have a per-weight step size, we won't be going in the direction of the gradient anymore! But we'll argue that this might be a good idea.

This next idea is something that will come up in reinforcement learning too:

Definition 40

Suppose we have some input values $a_1, \cdots, a_T$. The **running averages** are defined as

$$\begin{cases} A_0 = 0 \\ A_t = \gamma_t A_{t-1} + (1-\gamma_t)a_t \end{cases}$$

for some $\gamma_t \in (0, 1]$.

In other words, we take some convex combination of our previous average and our new value a_t. If γ_t is close to 1, then A_t is pretty close to A_{t-1}, so our running average has a "long memory." On the other hand, if γ_t is close to 0, then A_t is close to a_t.

Example 41

The **moving average** is a simple case of this, where all γ_ts are equal to some constant.

Doing out the calculations,

$$A_T + \gamma(A_{T-1}) + (1-\gamma)a_T = \gamma^2 A_{T-2} + a(1-\gamma)a_{T-1} + (1-\gamma)a_T \cdots$$

and the coefficients form a geometric sequence

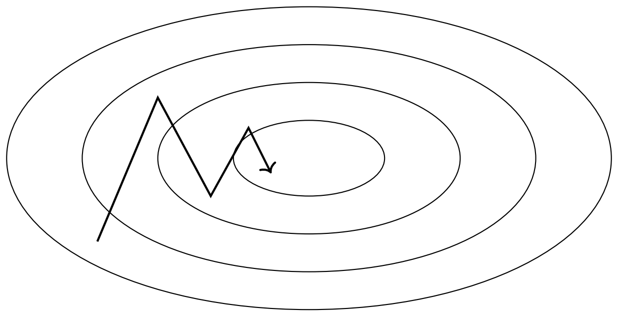
$$= \sum_{t=0}^{T} \gamma^{T-t}(1-\gamma)a_t.$$

. The first data value, corresponding to $t = 0$, has a very small contribution, because γ^T is close to 0. So this is a way to be sensitive to the history of our values, without keeping everything forever.

Fact 42

Setting $\gamma_t = 1 - \frac{1}{t}$ just gives us back the usual average, and we can check this for ourselves.

One thing we need to worry about when we have an elliptical shaped surface for our gradients, so we take a path like this (the issue is that we're updating a lot, but not actually moving too much in the correct direction):



So the point is that we somehow want to average out the direction over time, and the idea here is the **momentum**

rule: it's usually defined as having a **velocity** which defines our weight updates via

$$\begin{cases} V_0 = 0 \\ V_t = \gamma V_{t-1} + \eta \nabla_W J(W_{t-1}) \\ W_t = W_{t-1} - V_t. \end{cases}$$

But here's a better way which is probably more clear:

Definition 43

Define $M_0 = 0$, and define our **momentum**

$$M_t = \gamma M_{t-1} + (1-\gamma)\nabla_W J(W_{t-1})$$

to be the moving average of our gradients. Then we do a weight update in the moving average direction:

$$W_t = W_{t-1} - \eta' M_t,$$

This way, if we initially keep bouncing between two different values, we'll eventually smooth out that problem. It's nice here that we still have one knob η', but it's much easier to tune than the ones before.

There's another problem we might have, where our function essentially drops down by a lot at certain points but not others: so $|\nabla|$ is small at certain points, meaning we get very small updates, and large at others, meaning we might jump by too much. So the intuition here is to take small steps with $|\nabla|$ is big, and vice versa, and this is called the **ada-delta** or **ada-grad**:

Definition 44

Define constants

$$g_{tj} = [\nabla_W J(W_{t-1})]_j$$

(take the jth component). Also define the moving average of the gs via

$$G_{tj} = \gamma T_{t-1,j} + (1-\gamma) g_{tj}^2,$$

which is the moving average of the **squared gradient magnitude**. Then update via

$$W_{tj} = W_{t-1,j} - \frac{\eta}{\sqrt{G_{ij} + \varepsilon}} g_{tj}.$$

This makes sense except for that denominator: we're updating the gradient by multiplying by some step size. ε is here to keep us out of trouble if our Gs are too small. So if we suddenly hit a place with large g_{tj}, we'll update with a smaller effective η. (Notice that the g_{tj}s need to be squared or at least absolute-valued so that the moving average G actually tracks the magnitude of what's going on.)

Fact 45

We often use these in stochastic or minibatch gradient descent to help regulate or smooth things out.

So in practice, people basically do a combination of these things. We'll call this method "Adam" for "adaptive momentum," and literally everyone uses this. Here's the equations that govern what we do:

$$\begin{cases} m_{tj} = B_1 m_{t-1,j} + (1 - B_1) g_{tj} \\ V_{tj} = B_2 V_{t-1,j} + (1 - B_2) g_{tj}^2 \end{cases}$$

are a moving average of our gradient and squared gradient, respectively. Also do a little correction

$$\hat{m}_{tj} = \frac{m_{tj}}{1 - B_1^t}, \quad \hat{V}_{tj} = \frac{V_{tj}}{1 - B_2^t}$$

to correct for the fact that we start off by underestimating (this is basically having to do with only having the first t terms of the geometric series sum). Then we'll just update

$$\boxed{W_{tj} = W_{t-1,j} - \frac{\eta}{\sqrt{\hat{V}_{ij} + \varepsilon}} \hat{m}_{tj}}$$

in the momentum direction with a step size that depends on the magnitude of V.

So we have four magic parameters $B_1, B_2, \eta, \varepsilon$: the original Adam paper suggests

$$B_1 = 0.9, \quad B_2 = 0.999, \quad \varepsilon = 10^{-8}.$$

Basically everyone uses these values, and we're just left with η as our own knob to turn.

So this is a way to help our optimization work better: it's just about minimizing J in an efficient way. But we've talked about how this simple J (just the empirical loss) might not be the thing that we want to optimize, because it might overfit. So the question becomes: **what kind of regularizer should we use for neural networks?**

It turns out that neural networks trained with stochastic gradient descent don't actually overfit as much as we may expect, and this is an open area of research! Regularization is still useful, though, so we'll still make sure to mention it. One thing that seems to make sense is to look at

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(h(x^{(i)}, W), y^{(i)}) + \lambda ||W||^2$$

(so we take the sum of the squares of the weights again). This is still not a crazy thing to do, and if we compute out the gradient update, we end up with

$$W_t = W_{t-1}(1 - \lambda) - \eta \sum_{i=1}^{n} \nabla_W \mathcal{L}(h(x^{(i)}, W), y^{(i)}).$$

This can be thought of as **weight decay**: we basically force ourselves to avoid making the weights too big by reducing everything by a factor of $1 - \lambda$ each time.

An alternative method is to do **early stopping**: we can usually plot a curve of training loss as a function of time (**epochs** in neural network words, which measures how hard or how long we've been trying to optimize), and simultaneously also plot **validation loss**. What tends to happen is that validation loss starts to get larger after a while: at that point, overfitting is starting to happen. So we basically just deliver the weights at the minimum validation loss, and it's possible mathematically to show that this is actually very related to the first idea.

And as an addendum, something related to both of these is that we can add small Gaussian noise to our inputs. Overfitting means that we get too committed to our data points, so adding Gaussian noise avoids that as well!

The next idea, which is **batch normalization**, is not actually so well justified right now, but it works pretty well with mini-batches (of some size k). Basically, take our neural network, and between our W^ℓ linear layer and our f^ℓ

activation function, add a new layer N for normalization.

This layer N takes in an $n^\ell \times k$ matrix (k because we're feeding in k data points all at once). Each column corresponds to a data point, but we're actually going to do our normalization **row-wise**. In other words, N does a **separate processing** for each output unit of our W^ℓ.

Basically, for of these n^ℓ rows, compute the average and standard deviation

$$\mu_i = \frac{1}{k}\sum_{j=1}^{k} Z_{ij}, \quad \sigma_i^2 = \frac{1}{k}\sum_{j=1}^{k}(Z_{ij} - \mu_i)^2.$$

This gives us a z-score

$$\overline{z_{ij}} = \frac{z_{ij} - \mu_i}{\sigma_i + \varepsilon}$$

(again the ε to avoid bad denominators). This normalizes per output unit, per batch. If we made this our output, we'd be a bit too restrictive, so we also add additional weights

$$\hat{z}_{ij} = G_i \overline{z_{ij}} + B_i,$$

one per unit per layer. So in summary, each normalization unit in each layer has a G_i and B_i: we take the z-values that come out of the W^ℓ linear transformation, normalize the values, and then do an additional transformation to them.

So if we want to be able to do gradient descent, we have to also be able to calculate things like $\frac{\partial \mathcal{L}}{\partial B_i}$. This isn't too bad either, and then to do backpropagation, we have to be able to pass the gradients backward again. This is a weird way of looking at a neural network, because we're taking our k data points, putting them into the neural network together, and then having them interact in a weird way. It's nice here that there isn't really a magic parameter to deal with, and it turns out this helps with overfitting. But we don't have a great story why this is true yet!

Next week, we'll start talking about convolutional neural networks.

7 October 29, 2019

Today's lecture is being given by Professor Tamara Broderick.

We've been doing a two-step process in this class: first, define a model (such as a logistic regression or a fully connected neural network), and then try to optimize the weights or parameters that define our specific hypothesis in various ways (such as gradient descent). So far, all of the neural networks we've seen have been **fully-connected**, but we're going to see a new kind of model today called a **convolutional neural network**.

First, let's take a moment to understand why this is an important concept. ImageNet is an image database, which has been the center of a large image classification challenge. The neural network (AlexNet) which led the competition for a while was a convolutional neural network! This was the point where image recognition started to become better: from that point on, error rates for classification went from about 25 percent to less than 5 percent.

Remark 46. *But it's important to note that neural networks have been around for a while, and CNNs have been around since the 1980s. They've just re-entered the scene recently, because it turns out that convolutional neural networks are particularly good for image data, which has many applications: detecting medical tumors, using image search, optimizing autonomous driving, and so on.*

Definition 47

All images are made up of **pixels**, and each one has colors (usually an RGB value). In this class, we'll focus on just grayscale images, so they have some black-and-white values between 0 (black) and 1 (white).

So any image can be represented as an $m \times n$ array of floats between 0 and 1. **How exactly would we want to feed this in as an input into a neural network?**

Example 48

Let's go back to the fully connected neural networks from last class, and see if we can make any connections.

We always started by feeding in an input $n \times 1$ column vector, and then we connected every input to every output unit in the next layer. But it's **not so good** to just feed our image in as a string of pixels into our fully-connected neural network, because of two main principles regarding our images:

- Spatial locality: if we're looking for a fire hydrant or something like that, the pixels that dictate where our fire hydrant is must be in one specific area of our picture. Meanwhile, our fully connected neural network doesn't actually have any way of quantifying "nearby-ness."
- Translation invariance: if we're looking for a tumor, it's a tumor regardless of where it is in our image.

So let's try to build up a convolutional neural network! There are two new concepts: a **convolutional layer** and a **max pooling layer**.

Example 49

Let's say we have a one-dimensional image which looks like

$$(0, 0, 1, 1, 1, 0, 1, 0, 0, 0).$$

We're going to apply a **filter**, which gives us a transformation of the image that gives us some new information. Let's try $(-1, 1, -1)$: to apply this filter, we put this on top of three pixels at a time and take the corresponding dot product of the two vectors. For example, the first entry is

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = 0 \cdot -1 + 0 \cdot 1 + 1 \cdot -1 = -1.$$

Remark 50. *If we've taken a class like signal processing which has a "convolution," note that our "convolution" is a different definition from that one. Our definition just has to do with this dot product.*

And now if we move the filter over, we get the dot product of $(0, 1, 1)$ and $(-1, 1, -1)$, which gives us 0, and so on. Doing all of this yields a final output after convolution of

$$(-1, 0, -1, 0, -2, 1, -1, 0).$$

Often, we apply a ReLU or other activation function at this point, which yields

$$(0, 0, 0, 0, 0, 1, 0, 0).$$

Now it might be a bit more clear why we did all of this: the only 1 that survives our convolution corresponds to the pixel that is an **isolated** 1. So that's cool: we now have a way of determining (at least some simple) local features of our image!

Remark 51. *Notice that we have decreased the size of our image (we have an eight-pixel output instead of a ten-pixel output). To fix this, we can* ***pad*** *our image by adding a zero to both ends of the image, so that we'll have the same size image after convolution.*

We can also add a bias to our filter, so that every value gets an extra $+1$ (for example) before being applied to our ReLU function. Then (with padding) the result that comes out of convolution is

$$(1, 0, 1, 0, 1, -1, 2, 0, 1, 1),$$

and now we have a different output: it no longer tells us something as clear-cut as "isolated 1s," but it's still some detected feature of the image. And we also have a lot more nonzero entries, which means we're extracting a bit more information out of our image.

So we have a lot of parameters we can vary here: we can have a filter of size 2 or 3 or 4, we can change the weights in our filters, and we can change the bias. But notice that changing the filter does mean we need to change the amount of zero padding.

Question 52. *How many weights are there in this layer, without the bias?*

If we have a filter of length 3, there are just $\boxed{4}$ weights here. Compare this to a fully connected layer: we have 10 inputs and 10 outputs, so we'd need $\boxed{110}$ weights, including biases. It's not that we couldn't learn all of these ideas like translation invariance or spatial locality by using a fully connected network: it's just potentially harder and requires more time or computing power.

So now let's consider a two-dimensional image! For example, let's look at the following image "Hi" and filter:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

We can place the 3 by 3 filter on top of any 3 by 3 subsquare and take the element-wise dot product again, which yields the results (after convolution) $\begin{bmatrix} -7 & -2 & -4 \\ -5 & -2 & -5 \\ -7 & -2 & -5 \end{bmatrix}$. Notice that things have again gotten slightly smaller in both directions – in fact, we've lost many pixels! So if we do one column/row zero padding in all directions, we can get our initial image size back, which yields (before and then after ReLU)

$$\begin{bmatrix} 0 & -4 & 0 & -3 & -1 \\ -2 & -7 & -2 & -4 & 1 \\ -2 & -5 & -2 & -5 & -2 \\ -2 & -7 & -2 & -5 & 0 \\ 0 & -4 & 0 & -4 & 0 \end{bmatrix} \implies \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

What has this filter done? Again, we're detecting a lone white pixel! And we've indeed found the dot in the "i" of "Hi." And just like before, we can add a bias, which will give us a slightly different feature detection (we want "mostly" isolated pixels).

Remark 53. *This generalizes to more dimensions, too: for example, we might have an image where we care about depth, or we might want to represent our images with separate red, green, and blue-scale images, or we might have a live video.*

Of course, in actual applications, we won't just be told the outputs: we need to learn them.

Definition 54

A **tensor** is a generalization of a matrix: for example, an example of a one-dimensional tensor is a vector, and an example of a two-dimensional tensor is a matrix.

In general, we apply multiple filters to our initial image, which yields a tensor. We usually call the collection of filters in a given layer the **filter bank**.

So now that we've applied our convolution and ReLU layer, we often want to summarize what happens by asking the question of "is there anything that happened in this general area?" To do this, we just take the maximum value of the arguments in a given area: for example, consider the following max pooling of size 3×3, which takes 3×3 squares in an image and takes the maximum value of the entries:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \implies \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}.$$

This may seem like we're doing a lot of overlap, so maybe we don't need to know twice that nothing is found in an area. So we often use a **stride** to avoid this redundancy, which tells us that we skip over some number of squares each time we do the max pooling. (This is also a useful concept for filters.) For example, a stride of 3 would just yield $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$: we just break up our 6 by 6 matrix into 3 by 3 squares. Notice that this **has no weights**: we're just taking the largest value, so there's nothing that we have to learn in this layer!

So a typical CNN follows the following architecture:

- Convolution + ReLU
- Max pool
- Convolution + ReLU
- Max pool... repeat as many times as we want.
- Flatten the result into a single vector.
- Put this into a fully connected neural network.

The first part of this is "feature learning," and the second part is a classification (so often uses a softmax as the final activation function). Remember that we said earlier that we care about spatial locality, as well as translation invariance: the former is done by using a filter, and the latter by using max pool.

We'll finish this class by getting a taste of how to do backpropagation here. Let's do a simple model: say that we start with an $n \times 1$ column vector $X = A^0$ (which is our input image). Apply a filter of size k (with no additional bias) W^1, which yields a result Z^1 after convolution. Finally, apply the ReLU to yield A^1, which we then feed into our fully connected network to get an output $Z^2 = A^2$ (assume no activation in the fully connected network).

For concreteness, let's say $n = 10, k = 3$. A forward pass takes in our input and carries it forward to the end of our network, so let's write that out. First of all, the convolutional layer yields

$$Z_i^1 = W_1^1 A_{i-1}^0 + W_2^1 A_i^0 + W_3^1 A_{i+1}^0 = (W^1)^T \cdot A_{[i-1,i,i+1]}^0$$

(the dot product of the sub-vector of A^0 with W^1). Then

$$A^1 = \text{ReLU}(Z^1),$$

and then we get our fully connected layer

$$Z^2 = A^2 = (W^2)^T A^1,$$

which is our final guess (and we can apply a loss function to it relative to the actual value y). It's important to note the **differences between the layer** Z^1 and Z^2: we use the same weights for all elements of Z^1, but we use different weights for the elements of Z^2.

So now that we know the value of our loss, how do we actually update our weights? Everything boils down to computing a gradient, so we need to be able to compute the gradient of the loss with respect to our weights W. We, again, will use the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial \mathcal{L}}{\partial A^1}.$$

Most of this is very similar to the backpropagation that we learned with the fully-connected layer, and the only term that we haven't learned how to deal with yet is this first term $\frac{\partial Z^1}{\partial W^1}$. Since Z^1 is $n \times 1$ and W^1 is $k \times 1$, our derivative should be $k \times n$:

$$\frac{\partial Z^1}{\partial W^1} = \begin{bmatrix} \partial Z_1^1/\partial W_1^1 & \partial Z_2^1/\partial W_1^1 & \cdots \\ \partial Z_1^1/\partial W_2^1 & \partial Z_2^1/\partial W_2^1 & \cdots \\ \partial Z_1^1/\partial W_3^1 & \partial Z_2^1/\partial W_3^1 & \cdots \end{bmatrix}.$$

And it's not hard to calculate each of these: for example,

$$Z_1^1 = W_1^1 A_0^0 + W_2^1 A_1^0 + W_3^1 A_2^0 \implies \frac{\partial Z_1^1}{\partial W_1^1} = A_0^0, \quad \frac{\partial Z_1^1}{\partial W_1^2} = A_1^0, \quad \frac{\partial Z_1^1}{\partial \partial W_3^1} = A_2^0.$$

(Notice that A_0^0 is the zero padding on the left side of the image, because we've been using 1-based indexing.) So in general, this gives us a gradient matrix of

$$\frac{\partial Z^1}{\partial W^1} = \begin{bmatrix} A_0^0 & A_1^0 & A_2^0 & \cdots & A_9^0 \\ A_1^0 & A_2^0 & A_3^0 & \cdots & A_{10}^0 \\ A_2^0 & A_3^0 & A_4^0 & \cdots & A_{11}^0 \end{bmatrix}.$$

And now it's easy to do the other components: $\frac{\partial A^1}{\partial Z^1}$ is an $n \times n$ diagonal matrix (because we have the ReLU activation), where each entry is 1 if Z^1 is larger than 0 and 0 otherwise. And finally $\frac{\partial \mathcal{L}}{\partial A^1}$ is an $n \times 1$ matrix, which tells us the gradient from the fully-connected layer:

$$\frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} = (\text{whatever the loss gradient is}) \cdot W^2.$$

It's always important to make sure we make sure all of the dimensions match up! And now we can do gradient descent to optimize our weights.

8 November 5, 2019

This week, we're going to take a temporary detour from learning, because starting next week, we'll look at learning models that are more complicated than what we normally call a **feed-forward network.** In all cases so far, we've been outputting a function: given an input, this function returns an output. But in the coming weeks, we'll look at recurrent neural networks and reinforcement learning: in both of these, the output might depend on the history of the inputs in some kind of way! So we'll look at some background that will help us do both of those things.

Definition 55

A **state machine** is, in some way, a generalization of a function: the output might depend on the history of inputs. Formally, we can write it as a tuple $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f, g)$, where

- $\mathcal{X}$ is a set of possible inputs to the state machine (sometimes discrete, sometimes continuous or vector-valued), and $\mathcal{Y}$ is a set of possible outputs.
- $\mathcal{S}$ is a state of **internal states**: they're a kind of memory, and s_0 is the starting state for our particular machine.
- f and g describe how the function works: $f : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ takes in a current state and an input and gives a new state (often called a **state transition function**). $g : \mathcal{S} \to \mathcal{Y}$ is the **output** of our machine.

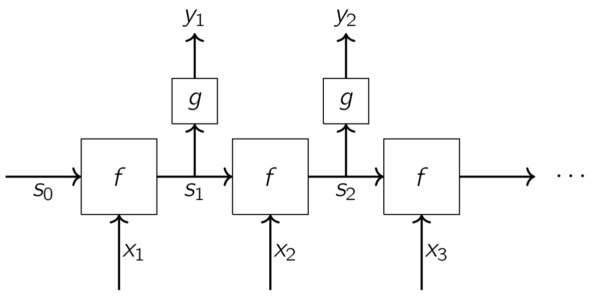
So what does this machine exactly do? f and g are at the center of what's going on here: we get a sequence of inputs x_i, and the machine computes its state (memory)

$$s_t = f(s_{t-1}, x_t)$$

in terms of the previous state and the input we receive. From this, it also generates an output

$$y_t = g(s_t).$$

So we can see that given our sequence of inputs $x_1, x_2, \cdots$ and a starting s_0, we can easily generate a sequence of outputs by just repeatedly applying these functions. Here's a picture of what's going on, which is the "unrolled" diagram:



This process of going from a sequence of x_is to a sequence of y_is is known as **transduction**. This picture includes **finite state machines**, where the $\mathcal{S}, \mathcal{X}$s are discrete: in those cases, it's common to draw a **state transition diagram**. The idea is that we draw states as nodes, and we draw arcs between states, labeled by inputs. For example, if we get $x^{(2)}$ as an input, we go from $s^{(0)}$ to $s^{(1)}$ (this means that $f(s^{(0)}, x^{(2)}) = s^{(1)}$). Basically, we draw a directed graph, labeling each possible transition between our states based on our inputs. (Often, we also include the outputs $g(s_i)$ in our transition diagrams, too.)

So really, a **recurrent neural network** is just a state machine family, parameterized by some weights. Here, $\mathcal{S}, \mathcal{X}, \mathcal{Y}$ are sets of m, ℓ, and n-dimensional vectors, respectively: then our transition function takes the form

$$f(s, x) = f_1(W^{sx}x + W^{ss}s + W_0^{ss}).$$

Here, f_1 is an activation function (such as a ReLU or tanh), W^{sx} is a weight matrix of dimension $m \times \ell$ (which transforms our input into the state space), W^{ss} is an $m \times m$ matrix (which transforms our old state into a "new state"), and w_0^{SS} is just an offset and is $m \times 1$. So we make a neural network which runs a linear layer through an activation function! Similarly,

$$g(s) = f_2(W^0 s + W_0^0)$$

takes in our state s and returns f_2 of our linear layer, where W^0 is $n \times m$ and W_0^0 is $n \times 1$. This is no longer a finite state machine, and we'll study this a bit more on two weeks.

Example 56

Say I have a robot, and I want to train my robot to act in a certain way. The way we can model this is that our robot can take **actions** that affect the state of the world, and it can make **observations** that tell it something about the state.

From the robot's perspective, we're modeling the world as a state machine, but we need to generalize our ideas a little bit more:

Definition 57

In a **Markov decision process** (MDP), our states have a **reward**, and we have probabilistic transitions between our states. Formally, we can write this as a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$, where

- $\mathcal{S}$ is a set of states: we'll mostly study the discrete-state case, and we'll eventually generalize.
- $\mathcal{A}$ is a set of actions: we can think of this as the inputs into our MDP (from our robot into the world).
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the **transition model**: $T(s, a, s')$ tells us the probability that $S_t = s'$, **conditioned on** S_{t-1} being s and A_t being a. In other words, if we had state s and we get action a, what's the probability that we end up at s'? We can often describe this as a set of matrices, one for each possible action a.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a **reward function**: this measures how good an action was (in the context a particular state).

In other words, we get a probability distribution for what states we end up in, and this makes sense in the real world: if we tell a robot to move forward 1 meter, that won't exactly happen. For now, we're assuming **complete observability**: this means that $y = s$, so although we're not sure what's going to happen when we tell our robot to move, we do know exactly what happens after the state machine transition occurs. So this is like saying g is the identity, or that we don't have hidden states!

So this lets us describe how the robot should describe: basically, we're trying to maximize our reward. And we want to calculate a way of behaving in the world that gets a lot of reward.

Example 58

Say the robot only lives for one step, and it knows its current state s: what action a should it take?

This question is easy: we should pick $\text{argmax}_a R(s, a)$, because we only have one opportunity to get a reward. (And we often have to think about the value of R in expectation.)

Example 59

But now what if the robot takes more steps? Now it's possible that we might accept lower-reward choices now so that we have higher-reward choices later.

In general, we want to find a **policy** $\pi : \mathcal{S} \to \mathcal{A}$ which tells us what the best action will be. One policy is exactly this $\text{argmax}_a R(s, a)$ example from above, and once we equip our robot with this policy, we have a **composite state machine** which enters a sequence of states.

Definition 60

The function

$$Q^h(s, a) = \text{optimal expected sum of rewards in } h \text{ steps if we start at state } s \text{ and execute action } a,$$

where we assume that our future self will know what to do in the following $h - 1$ steps. h is called the **horizon**.

Here, the **expected value** can be defined as

$$\sum_{\text{state sequences}} \mathbb{P}(\text{sequence}) \cdot \text{reward}(\text{sequence}).$$

So if we put some policy in our robot, we'll get a well-defined probability distribution of the sequence of states, and that gives us a well-defined expected reward. Our goal is to basically behave in a way that gives us high reward with high probability.

Example 61

What are some values of $Q^h(s, a)$ for small h?

If we're at $h = 0$, $Q^0(s, a) = 0$, because we can't take any actions at all. If we're at $h = 1$, we just have

$$Q^1(s, a) = R(s, a),$$

because we only get one action and we just get that particular reward.

What about larger values of h? For $h = 2$, notice that if we take action a, we get reward $R(s, a)$, but then we have to add the expected amount of reward we get in the next step. Thus,

$$\boxed{Q^2(s, a)} = R(s, a) + \sum_{s'} T(s, a, s') \cdot \max_{a'} R(s', a') = \boxed{R(s, a) + \sum_{s'} T(s, a, s') \cdot \max_{a'} Q^1(s', a')},$$

because the set of possible states we end up in is probabilistic: T refers to the probability of transitioning to s', and then we have only one more step to go, so we just maximize our reward. This now generalizes: we have for any finite h that

$$Q^h(s, a) = R(s, a) + \sum_{s'} T(s, a, s') \cdot \max_{a'} Q^{h-1}(s', a').$$

This also tells us the optimal policy for any finite horizon h: we should take

$$\pi^h(s) = \text{argmax}_a Q^h(s, a).$$

In general, this means that the best action **depends on our horizon**. And it's pretty easy to compute these Q^h values: start with Q^0, then go to Q^1, and so on.

A much more common setting, though, is where we have **infinite horizon discounting**, and this is where our $\gamma \in (0, 1)$ comes into play. The idea is that if r_i is our reward at step i, we want to maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots .$$

The intuition for this is either that (1) we assume we halt at each step with probability $1 - \gamma$, or (2) we value rewards later in the game less. (If we are worried this doesn't fit into the definition of our state machine, we can just think of there being a "Game Over" state.)

It turns out that in these cases, there exists a **stationary** optimal policy $\pi : S \to A$, which is **horizon-independent**! This is because in this model, surviving to the next step looks identical to to the starting situation (because our chance of dying is still $1 - \gamma$).

How do we find this? Define the Q function analogously as before:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_a Q^*(s', a').$$

where the γ factor comes because all future rewards are scaled by γ, and remember that Q^* is stationary (so it is the best policy when we're in state s' as well).

It's not obvious, but it turns out there is a unique solution Q^*! Basically, follow the following algorithm to get close to that solution:

- Input an MDP and a termination parameter ε.
- Initialize $Q_{\text{old}}(s, a) = 0$ for all s, a.
- **Loop the remaining steps**:
- Compute a new Q function

$$Q_{\text{new}}(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q_{\text{old}}(s, a').$$

- If the maximum difference between $Q_{\text{new}}(s, a)$ and $Q_{\text{old}}(s, a)$ (across all s, a) is ε, then return Q_{new}. Otherwise, set Q_{old} to Q_{new}, and continue looping.

So now $Q_{\text{new}}(s, a)$ dictates our policy: we should just pick the a that gives us the best future, so

$$\pi(s) = \text{argmax}_a Q_{\text{new}}(s, a)$$

is a pretty good approximation of the actual optimal Q^*!

Remark 62. *It's not so easy to show that this converges, but it's not too hard either: the main idea is that this has the property that Q_{new} gets closer to Q^* on each iteration in some well-defined way.*

This algorithm is fairly flexible, and that's why a lot of reinforcement learning works well in practice!

9 November 12, 2019

Let's refine some of the ideas from last time and start introducing **reinforcement learning**. Say we're interacting with some unknown MDP, and we know the possible states and actions we can take, but we don't know the transition model. How can we try to optimize our reward?

Let's start with a simple example:

Example 63 (Bandit problem)
Slot machines used to be called "one-armed bandits," because they steal your money. We can think of us as having a k-armed bandit if we have k slot machines! Formally, the definition is more fair: say that each machine has some hidden probability p_i of giving us money: suppose that we pull the arm of a machine, and with some probability p_i it gives us a dollar.

If we knew the ps, we should just pull the arm of the machine with the highest p. But we don't know the probabilities in this case: let's say that we're given 100 pulls, and we want to maximize our money. What's the best thing for us to do?

There isn't an obvious answer here. We could try each a few times and see which one is the best, and then spend the rest of the trials on that machine. It turns out that if we spend T total trials, the optimal answer is about $\sqrt{T}$ time on exploration. (A less good strategy is "switch on a loser:" basically switch once you don't get money.) We can also translate this into our probability $1 - \gamma$ of terminating.

Notice that the main difference so far is that **in reinforcement learning, we get to influence our data** (while we didn't in supervised learning). Also, errors do actually matter while we're learning. In supervised learning, we didn't care about making errors in gradient descent as long as we ended up with an answer we liked. Meanwhile, in this casino problem, the performance during learning does have an impact on your final reward (pulling a slot machine and not getting money is a lost dollar).

Remark 64. *Generally overestimating the goodness of a slot machine is less bad than underestimating. There's some asymmetry here: if an option looks bad to us, we're less inclined to try it, even if it turns out it's really good!*

So now let's move to a harder problem.

Problem 65
We're interacting with an MDP with some n states, but every time we take an action, we get **both a reward and a transition to another state**. One way to think about this is that we have a multi-room casino, and each time we pull an arm, we get transferred (probabilistically) to another room.

So much like we don't know our p_is in the slot machine problem, we now don't know T, our transition model, or R, our rewards. How can we optimize our money gain?

So we might pull an arm for various reasons: we might want to see what the reward is, we might want to see what the room we're taken to looks like, or we might want to aim for a high reward or for a good follow-up room. A key idea here is **exploitation versus exploration**, where we try to balance getting reward versus knowing more about our model.

One thing we do know is that optimally interacting with an unknown MDP is **very hard**: it's a POMDP (partially observable MDP), which is computationally hard to optimize. So getting as much money as possible while learning is very difficult.

Our strategy will be to still do reinforcement learning and interact with the world (MDP), but we won't worry about the reward during learning (at least not too much). We'll just try to discover an approximation of π^*, the optimal policy, and we're going to assume that we're working in the discounted infinite horizon case. (For example, if we're trying to learn how to play Starcraft, we can just run the reinforcement learning algorithm in our garage, and we don't really care about the reward until the tournament.)

There's really three basic algorithms we can use here:

1. Model-based
2. Model-free: search for a policy
3. Model-free: search for a value function Q.

(1) In a model-based algorithm, our goal is to try to estimate T and R, the transition model and reward function, and then solve the MDP $(S, A, \hat{T}, \hat{R}, \gamma)$. Mathematicians like this, because we can use value iteration to solve the problem once we have the full model! And we know that $\hat{T}$ and $\hat{R}$ can be found by supervised learning: if we assume the states and actions are discrete, we can estimate $\hat{R}(s, a)$ by taking action a at state s a bunch of times, and then look at the expected value. Formally, we get a bunch of **experience** of the form

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \cdots$$

where we progress in states, actions, and rewards. Then

$$\hat{R}(s, a) = \frac{\sum r|s, a}{\#(s, a)}$$

just adds up the rewards from a given state s and action a, and divides by the number of such instances to give us an average. The transition model is a tiny bit trickier: we estimate the probability of transitioning to state s'

$$\hat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |S|}.$$

This is primarily the fraction of the time we end up in state s' when we're at (s, a), but there's some extra add-on terms: the $\frac{1}{|S|}$ is called the **Laplace correction**, and it helps us in the case where there isn't too much data. We default to assuming that if we have no data, (s, a) takes us uniformly to one of the other states, and this effect goes away as we sample (s, a) enough times.

So in this strategy, there's only one thing we haven't considered: how exactly do we pick our actions? In reinforcement learning, we have to pick the actions $a_0, a_1, \cdots$, so that we have enough data to figure out $\hat{T}$ and $\hat{R}$. Acting completely at random is not bad, but it's good to bias so that we take actions that seem to be better. And we might want to keep iteratively getting $\hat{\pi}$, so that we're not bound to a specific estimate.

(2) We won't talk too much about this. In **policy search**, we basically forget that we have an MDP: this is fairly robust to partial observability because we're not explicitly trying to construct our MDP. We are trying to define a function

$$f(s; \theta) = P(a|s, \theta)$$

which gives us a probability distribution of actions given the state that we're working in. The idea is that we just want to know the **way we should behave**.

For example, mazes can be solved in this way: one pretty simple solution is to keep turning left, and if this solves our problem, it's way easier than trying to make a transition model for the whole maze! And our goal is to find the parameters θ (we can think of this as weights) that maximize some expected value of the reward. So given a θ, we have a policy, and given an initial state s_0, our policy π_1 causes us to go through some trajectories in our set of states: we want to maximize these trajectories' rewards.

We often pick some initial θ_1, and we perturb it by a little bit and see whether things improve, so this looks a bit like gradient descent.

(3) Trying to learn a **value function**: we can often do this using **Q-learning**. Basically, we are trying to estimate $\hat{Q}(s, a)$, but this time we don't actually try to estimate $\hat{T}$ and $\hat{R}$ (which can be computationally hard when we have a lot of states).

Remember that by definition,

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$$

gives us the maximum possible reward if we take an action a in state s: if we know T and R, we can do value iteration inductively to solve for these values (and get closer and closer to the actual Q). And we can just turn this into an algorithm by turning math into computer science: basically make the equality into an assignment.

Here's the algorithm:

- Initialize all $Q(s, a) = 0$. **Loop the following:**
- Somehow select an action a given our state s. Picking uniformly at random is fine.
- Execute action a, so we get some reward r and end up in some state s'.
- Assign
$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a')).$$
- Set $s = s'$ (we're in our new state), and repeat.

Notice that we have a moving average of Q here: α tells us how much we take our old and new value into account. There's two really important free variables in our expression: r corresponds to the reward $R(s, a)$ in the definition of Q (or if R is probabilistic, it's a sample of R), and s' is the next state drawn from our distribution: s' comes up with probability $T(s, a, s')$. So **the definition of $Q(s, a)$ is an expectation, and we're drawing samples to estimate that expectation.**

Basically, we mix two kinds of iteration here: we're doing a moving average, and we're also doing the value iteration (starting with terrible Q values, and gradually approaching the correct values). And we can prove that if we explore enough, this algorithm will converge to Q^* and π^*, the optimal policy and reward.

We've assumed in this algorithm that we can actually make such a table of s's and a's (we have discrete states and actions), but that's not really the case that we care about in practice: we often have a continuous or super big set of states if we're trying to control a robot or play Starcraft.

The right thing to do, then, is to try to store Q in a neural network. There's lots of different architectures for this: if we have a discrete set of actions A, we can just think of having a neural network which takes in s and outputs $Q(s, a_1), Q(s, a_2), \cdots, Q(s, a_k)$. We can also extend this to continuous a if we feed in s and a into a neural network and get out $Q(s, a)$: things do get tricky, though, because finding $\max_{a'} Q(s', a')$ is not so easy anymore!

There's two main ways to implement this in practicality. The first is called **deep Q**, and it comes from the fact that we can rewrite our equation as

$$Q(s, a) = Q(s, a) - \alpha[Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))].$$

This looks a lot like a gradient update! α is sort of a step size, and the bracketed term is sort of an error. Roughly, the initial Q algorithm just did a linear regression version of this: instead of $Q(s, a)$ being a table, $Q(s, a)$ is a function that depends on weights. **This isn't exactly a gradient descent update, but we will pretend it is, just like everyone else.** The main problem is that there is no objective function that we're optimizing, so we can't actually prove much of anything about Q-learning! In fact, we can find examples where this fails, but we can often make it work. The main problem is that changing one value of $Q(s, a)$ often changes other ones as well if we're in the continuous case, which makes things less reliable.

One problem we might have is **non-iid-ness**: our model might learn what to do in the daylight if we keep training it on images that are light-colored. But if we try to run it on dark images, the phenomenon of **catastrophic forgetting** often occurs. A cure for this is to use **experience replay**: we remember some of the examples from the light, and pretend that we experience them again in the nighttime, so we force our neural network to come up with solutions that can do both the light and dark cases.

An alternative to deep Q is called **fitted Q**. We start with an empty data set $\mathcal{D}$, and let's say that $Q(s, a)$ is some neural network. Loop the following:

- Behave according to π_Q, whatever Q currently is, and do this for k steps. (This could be an arbitrarily stupid policy.) Usually, we add in some randomness as well: we say that there is probability ε that we pick something at random.
- This gives us some new dataset $\mathcal{D}_{\text{new}}$: add this to $\mathcal{D}$.
- Train our network with the new data: make a supervised dataset

$$\mathcal{D}_{\text{supervised}} = \left\{ \left((s, a), r + \gamma \max_{a'} Q(s', a') \right) \right\},$$

  and do regression. This gives us a new Q function, which we feed back into the first step.

As we can see in the demo, it unfortunately takes an enormous number of steps to converge to the right answer. So somehow it's a miracle that it works on very complicated domains!

10 November 19, 2019

Today, we'll continue with our state machines and talk about **recurrent neural networks.** Basically, we're again going to find a state from our input, and then extract an output from our state, but instead of doing something like Q-iteration, we use a neural network to predict our next state and estimate our output (because we may have too many possible states to deal with like in the discrete case).

We're going to study a particular (more basic) form of the neural network, but we could throw basically any neural network as we want into the boxes f_1 and f_2: the story would still work out. The equations we're going to work with are of the form

$$s_t = f_1(W^{sx} x_t + W^{ss} s_{t-1} + W_0^{ss})$$
$$y_t = f_2(W^0 s_t + W_0^0),$$

which is a state machine parameterized by three matrices of weights and two vectors of offsets.

Remark 66. *If the f's are identity functions, we get a linear system, and this is still pretty interesting (the state machine isn't as degenerate as if we didn't have nonlinearities in a feed-forward neural network).*

Earlier in the class, we looked at how to encode our inputs and outputs into a neural network: all of these ideas still apply here. For example, we'll look at examples where the inputs are characters of an alphabet: if we're trying to map a string of characters into another string of characters, we might represent this with a one-hot encoding.

So we'll think of this as a **sequence to sequence** problem: our training data set takes the form

$$\mathcal{D} = [(x^{(1)}, y^{(1)}), \cdots (x^{(q)}, y^{(q)})],$$

so we have q training examples to work with. Each $(x^{(i)}, y^{(i)})$ is a pair of sequence of length $n(i)$: it's okay for the lengths to be different among our dataset, as long as the lengths of the input and ouput are the same.

When we're trying to compute our loss here, we care about how well we predict each of our sequences: suppose that we generate a sequence $p^{(i)}$ on data point i, but we were supposed to generate $y^{(i)}$. How do we gauge how badly we've done? One thing we could do is just sum over our whole sequence and compute those individual losses:

$$\mathcal{L} = \sum_{t=1}^{n(i)} \mathcal{L}(p_t^{(i)}, y_t^{(i)}),$$

and the loss on each element depends on the context we're using our neural network in (squared loss, NLL loss, and so on). We can write this formally as follows: if our parameters can be summarized as $\Theta = (W^{sx}, W^{ss}, W_0^{ss}, W^0, W_0^0)$, then the objective function here is

$$J(\Theta) = \sum_{i=1}^{q} \mathcal{L}(\text{RNN}(x^{(i)}; \Theta), y^{(i)})$$

(we want to minimize the sum of the losses over all sequences).

We're going to cover an algorithm, called **backpropagation through time**, which has the right principles but isn't used in practice anymore. It can be used in batch or mini-batch or pure stochastic gradient descent: for simplicity, we'll consider the weight update based on a single data point (x, y), which are sequences of length n.

To do this, let's unroll our recurrent neural network: here are the steps that we're going through repeatedly.

- Feed in a state s_n.
- Feed in our new input x_{n+1}, and add the contributions $W^{ss}s_n$ and $W^{sx}x_{n+1}$ together to get some output z_{n+1}^1. (Then add an offset.)
- Apply a nonlinearity f_1 to z_{n+1}^1 to get our new state s_{n+1}.
- Take s_{n+1} and feed it through our output weights to get our output y_{n+1}.
- Repeat with our new state.

We want to do gradient descent here: since we can track what happens forward, we can also think about how to go backwards with backpropagation. We'll focus on W^{ss} and W^{sx}, because those turn out to be trickiest: something like W^{sx} comes into our loss function at a lot of different layers! It contributes to every single output, so we need to look at every single y to figure out its contribution. And also our sequences might all have different lengths, which further complicates our calculations.

So first, we need to do a forward pass, which is pretty easy: we just carry our state s_0 forward through the n elements of our sequence, and then we know all of our outputs. Then the loss with respect to some weight matrix can be written as

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{u=1}^{n} \frac{d\mathcal{L}_u}{dW}$$

(where $\mathcal{L}_u$ refers to the individual element loss $\mathcal{L}(p_u, y_u)$).

Fact 67

Note that we're summing over **total derivatives** on the right side, while the left hand side is a partial derivative. The purpose is that W contributes to the loss in a lot of different ways, and we care about more than just the ways where it's explicitly being used.

Now $\mathcal{L}_u$ can depend on the weight matrix W in a lot of ways, dependent on the states s_t, so we use the multivariable chain rule:

$$\sum_{u=1}^{n} \frac{d\mathcal{L}_u}{dW} = \sum_{u=1}^{n} \sum_{t=1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t} \frac{\partial s_t}{\partial W}.$$

The first term here doesn't depend on the weights at all, and the second is easier to deal with than our initial expression as well. Changing the order of the summation, and noting that the second term doesn't depend on u, we can write this alternatively as

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t} = \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=t}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

The reason for the last equality is that the inner sum asks us "how does the loss at time u depend on our states at time t?", and this can only have a dependence if $t \le u$. And now the inner sum can be rephrased as "how much do we blame state s_t for all losses that come after it?"

Well, we can do this recursively if we rewrite our sum a little bit:

$$\sum_{u=t}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t} = \frac{\partial L_t}{\partial s_t} \sum_{u=t+1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t},$$

and we'll call this second term δ^{s_t}. We now just have three jobs: figure out how to compute $\frac{\partial s_t}{\partial W}$, $\frac{\partial \mathcal{L}_t}{\partial s_t}$, and δ^{s_t}, and we'll have what we want!

We'll start with finding δ^{s_t}, and we'll do this with recursion by going backwards. (For simplicity, we're going to say the f is the identity function here, but it doesn't change very much.) The base case is easy:

$$\delta^{s_n} = 0,$$

because n is the end of the sequence (so there's nothing after it). Then we can write

$$\boxed{\delta^{s_{t-1}}} = \frac{\partial}{\partial s_{t-1}} \sum_{u=t}^{n} \mathcal{L}_u$$

by definition (we're just pulling out the "partial operator" from all terms), and by chain rule, this is just

$$= \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial}{\partial s_t} \sum_{u=t}^{n} \mathcal{L}_u.$$

If we now just divide this into two terms (intuitively we're just separating out the first term, just like in the definition of δ^{s_t}),

$$= \frac{\partial s_t}{\partial s_{t-1}} \left(\frac{\partial \mathcal{L}_t}{\partial s_t} + \frac{\partial}{\partial s_t} \sum_{u=t+1}^{n} \mathcal{L}_u \right)$$

and this does give us the recursive form

$$= \boxed{\frac{\partial s_t}{\partial s_{t-1}} \left(\frac{\partial \mathcal{L}_t}{\partial s_t} + \delta^{s_t} \right)}.$$

All of the partial derivatives that we've introduced in this recursion are actually pretty easy: $\frac{\partial s_t}{\partial s_{t-1}}$ just depends on our weight matrix (because we have an explicit formula for s_t in terms of s_{t-1}), and $\frac{\partial \mathcal{L}_t}{\partial s_t}$ is solely a function of our output function at layer t (and we actually wanted to find that elsewhere in our problem). So we can now, theoretically, run a for loop and compute all δ^{s_t}s, so we've finished two parts of our problem (we got $\frac{\partial \mathcal{L}_t}{\partial s_t}$ for free).

Finally, let's try to find $\frac{\partial s_t}{\partial W}$. This turns out to be pretty simple: because we're using a **partial** derivative here, we can just take a derivative explicitly in the equation

$$s_t = f_1(W^{sx} x_t + W^{ss} s_{t-1} + W_0^{ss}).$$

And so with this, we now know how to find the derivative with respect to any weights, and that completes our backpropagation algorithm!

Example 68

One case where we use a recurrent neural network is through a **language model**, where we're trying to predict the next character given a sequence of characters. (Characters can be words, or amino acids, or anything like that, too.)

The idea here is that if we have a bunch of sequences of the form $(c_1, c_2, \cdots, c_{t-1})$, we want to predict c_t. This is one way to approach the problem of translating between languages.

Luckily, we can reduce this problem to what we've already been talking about: we can encode our characters via

$$x^{(i)} = [\langle\text{start}\rangle, c_1, c_2, \cdots, c_n]$$

$$y^{(i)} = [c_1, c_2, \cdots, c_n, \langle\text{stop}\rangle].$$

We can notice that this indeed says that we can try to predict c_t based on the first t inputs of $x^{(i)}$, which are indeed c_1 through c_{t-1}. And if we want to do something like predict a probability distribution over characters, we want our input and output encodings to be one-hot, softmax to be our activation function for f_2, and multi-class NLL to be our loss.

Remark 69. *When we're doing translation, we don't expect our outputs to come directly after we feed in each word (translation is context-dependent). So the idea here is that we often feed in the whole sequence in our initial language, and then only after we give it an "end" token, we start looking at the output that is spit out. (And our loss functions won't depend on the outputs during our initial input.)*

Let's look a bit more carefully at our gradient calculation: we find that

$$\delta^{s_t} \approx \frac{\partial s_2}{\partial s_1}\frac{\partial s_3}{\partial s_2}\frac{\partial s_4}{\partial s_3}\cdots,$$

and each of these terms depends on W^{ss}. So this will either go to ∞ or to zero pretty quickly, and that's known as the **exploding/vanishing gradient** problem.

One way to deal with this issue is to use the **LSTM (long short-term memory)** network, which is a bit fancier than what we've discussed. This is known as a **gated** recurrent neural network: intuitively, s_t is the memory of our neural network, and if we multiply everything by weight matrices, any information we want to remember will get blurred. So we don't want to update every memory element each time, and we fix this by adding a **gating network**

$$g_t = \text{sigmoid}(W^{gx}x_t + W^{gs}s_{t-1}).$$

Our new state is then not just s_{t+1}, but a running-average type idea

$$s_t = (1 - g_t) * s_{t-1} + g_t f_1(W^{sx}x_t + W^{ss}s_{t-1})$$

where $*$ refers to component-wise multiplication (and we're ignoring offsets). So we're giving our network a chance to remember the g_t values! Since we're using different W^{gx} weights from the rest of our process, we can train the newtork to (for example) keep the first element of our state fairly constant so we can "remember" some things throughout the process.

And LSTM has three gating networks, so it's just an even more complicated example of a gating network.

11 November 26, 2019

We're basically done with neural networks at this point: we'll come back to a few techniques and problem solving methods that make up machine learning outside of neural nets!

Today, we're going to talk about **recommender systems**: this is an example of how new objective functions can be created to help solve a problem.

Example 70

Some automated systems want to recommend movies to watch or things to buy: one thing that's interesting is that there's always lots of items to pick from.

The problem here is that we have many users and many items, but each user only rates a few items: out of the thousands of movies available, a user might only watch 20 or so.

The first thing to try when solving this problem is to use a **content-based recommendation**. We treat this like a supervised learning problem: the idea is that if we have a bunch of items x, we cook up some features $\phi(x)$ and try to do a supervised learning problem

$$D_{\text{you}} = \{(\phi(x^{(i)}), \text{rating}(x^{(i)}))\}.$$

We can feed this into any algorithm we've developed, and for any new movie m, we can just take $\phi(m)$ and feed it into our hypothesis. The problem, though, is that we have very few ratings: it's very hard to classify movies just based on this.

Remark 71. *Pandora, a music streaming station that used to be popular, actually had humans featurize music manually! But they're not so active anymore, and in general this content-based recommendation isn't used in practice now.*

One idea is that we might want to use other people's data, too: if two movie watchers have rated movies similarly, the recommendations for those two people might line up. And this is the intuition for **collaborative filtering**: what's interesting here is that we don't even need to watch the movie or look at its features to get something meaningful.

The way we set up this problem is to store our ratings in a $n \times m$ **data matrix**: each of the n rows corresponds to a user, and each of the m columns corresponds to a movie. Then if user i rates movie j a score of r, we set $Y_{ij} = r$. It's important to note here that this matrix is very sparse: almost all of the entries are empty, so when we do actual data allocation, we store individual entries in the matrix rather than the entire grid

$$D = \{(a, i, r)\}$$

(corresponding to agent, item, rating). We can think of the nonempty entries Y_{ij} as being our training dataset: we're trying to make accurate predictions for those movie ratings, so it's almost like a supervised learning problem!

To try to figure out what Y looks like, we'll try to predict a matrix X which is completely full: we try to guess the ratings of all movies by all users. This is an optimization problem, so we need a few components:

- Hypothesis space
- Loss function
- Some kind of objective J.

It seems like our hypothesis space here is just X, which is an arbitrary $n \times m$ matrix. But this matrix is very big: there's far more parameters than there are ratings, so there's no way of actually estimating the whole matrix unless we can constrain it! (It's like doing regression with $d > n$.) So if we want to make sure there is generalization, we need to make some kind of structural assumptions.

Fact 72

If an $n \times m$ matrix X is **rank 1**, then there exist an $n \times 1$ matrix U and $m \times 1$ matrix V such that $X = UV^T$.

(Rank 1 matrices have all columns being a multiple of each other, and in particular, this means that $X_{ai} = U_a V_i$.) This is a very strong assumption: instead of having mn parameters in our hypothesis space, we only have $m + n$ parameters. This is an idea, but it's way too constrained: there's not enough flexibility, and everyone's ratings are just scalar multiples of everyone else's! We basically just end up with a bunch of averages.

Let's still go ahead and think about what kind of loss function we can have here, though: the main idea is that if we have a bunch of real numbers, we can just use the squared loss

$$\sum_{(a,i)\in D} (X_{ai} - Y_{ai})^2 = \sum_{(a,i)\in D} (U_a V_i - X_{ai})^2.$$

(Note that we only want to look at loss over ratings we actually know.)

Remark 73. *Note that if a movie hasn't been rated or a person hasn't rated any movies, we should not include them in our set of a or is. Also, note that we can add a regularizer term to this as well.*

To make our hypothesis class a little bit more expressive, we will now generalize a bit:

Fact 74

If an $n \times m$ matrix X is **rank k** (where $k \le m, n$), then there exist an $n \times k$ matrix U and $m \times k$ matrix V such that $X = UV^T$.

Intuitively, this means that each user now gets a vector of length k, and each movie now gets a vector of length k. And our predicted ratings are just the dot product of these vectors: we are now trying to make this generalize to all of the potential ratings in our matrix Y.

Proposition 75

Cross-validation is the best way to pick the correct value of k: we have to strike a balance between needing enough parameters to make a good prediction and having too many and overfitting.

The key idea is that these are k magic features that describe each user and each movie: instead of needing a human to intuit what the important features look like, "the data works better than engineering."

And now we can write down an objective function in terms of our parameters U and V:

$$J(U, V) = \frac{1}{2} \sum_{(a,i)\in D} (U^{(a)} \cdot V^{(i)} + b_u^{(a)} + b_v^{(i)} - Y_{ai})^2 + \frac{\lambda}{2} \sum_{a=1}^{n} ||U^{(a)}||^2 + \frac{\lambda}{2} \sum_{i=1}^{m} ||V^{(i)}||^2.$$

Here, we take a dot product between the user $U^{(a)}$ and movie $V^{(i)}$, but then we add offsets for users and movies (some people are grumpy, some movies are bad). The last terms here are regularizers: empirically, this seems to be useful, but we could also just regularize the entries of the matrix themselves, too.

This looks somewhat similar to the ridge regression problem from earlier in the semester, but unfortunately we don't have such a nice linear problem ($U^{(a)} \cdot V^{(i)}$ is not a linear term in our parameters, because we're multiplying two of our parameters together). In particular, the gradient looks like

$$\frac{\partial J}{\partial U^{(a)}} = \sum_{(a,i)\in D} (U^{(a)} \cdot V^{(i)} + b_u^{(a)} + b_v^{(i)} - Y_{ai}) V^{(i)} + \lambda U^a,$$

which means we won't have a nice linear equation to solve (and therefore no nice closed form). Instead, we can use stochastic gradient descent: our function isn't convex, so we can't be sure we're getting the optimal answer.

Instead, though, we do have another trick:

Proposition 76

If we fix $U^{(a)}$ and $b^{(a)}$ (the information corresponding to the users), then the gradient is linear in the $V^{(i)}$ and $b^{(i)}$ (information corresponding to the movies), and vice versa.

Notice, then, that if our $V^{(i)}$ and $b^{(i)}$ are fixed, we can think of our objective function

$$J = \frac{1}{2}\sum(\theta \cdot W + \theta_0 - (Y_{ai} - b_v^{(i)}))^2$$

as a **linear regression problem for each user**, where $W = V^{(i)}, \theta = U^{(a)}, \theta_0 = b_u^{(a)}$. And that motivates the idea of **coordinate descent**: we'll do the optimization problem by holding U, b_u fixed and finding the best V, b_v, then holding V, b_v fixed and finding the best U, b_u, and going back and forth! And "finding the best" U, b_u or V, b_v is just the ridge regression problem that we already know how to solve.

Question 77. *Why do we do this instead of gradient descent?*

One thing is that there is no step size here: we don't need to turn a knob to see how well our algorithm is converging. We may have to try different initial values, though.

12 December 4, 2019

Lecture notes from this week were transcribed from the Fall 2018 lecture video rather than the Fall 2019 lecture due to a snow day.

This is the last lecture with examinable material! Today, we'll do something different from other methods and strategies so far: we started with linear classifiers and regression, and then we started solving our problem with neural networks. Today, we'll look at a specific category of models called **non-parametric methods**.

This is a pretty bad name, because they do have parameters. But the main idea is that **the complexity of the hypothesis isn't fixed in advance**: it depends on the data that we apply it to. In the past, we picked some number of layers or fixed a neural network (perhaps by using cross-validation), but here our actual learning method does the model complexity choices while we're fitting our hypothesis. Main ideas we'll talk about are **decision trees**, **bagging**, and **nearest neighbor**.

Decision trees were developed almost simultaneously by statisticians and machine learning people, and a few decades ago, they were very popular. The idea is that we **recursively partition** an input space and fit a simple hypothesis to each region.

Example 78

Consider a 2D classifier, where we have a bunch of points labeled + or -. What's a good way to come up with a classifier if there isn't a clear linear classifier?

One strategy is to fundamentally look at the data and divide it into pieces that are positive and pieces that are negative. There might be variability in how we decide to do this splitting, what kind of hypothesis exists in each region, and so on.

Example 79

Meanwhile, consider a 1D classifier, which can be thought of as us trying to fit a function to a bunch of points (x, y).

Similarly, instead of drawing one line through all of the data, we might break up our input space (x) into a few regions and pick some constant on each region. And while this feels very different from our original regression model, it's at least easy to understand. The point is that the complexity comes from our division, because that gives us a lot more flexibility.

So we'll start with this regression example to understand what's going on. In **regression trees**, we partition our input space into M regions $R_1, \cdots, R_M$, and we have output values $O_1, \cdots, O_M$ (all constants). In this simple case, this means our hypothesis is piecewise constant:

$$h(x) : O_j \text{ when } x \in R_j.$$

These regions should not overlap, and they should partition the space. (And if it's on the boundary, we can do whatever we want: it's low probability.) It's a funny kind of hypothesis space, because we may not decide M until the algorithm starts doing its work.

In order to formalize the algorithm, we need to make a few more clarifications. We can define an error in each region: since we're doing regression, let's do a typical squared loss (but a lot of loss functions could work)

$$E_m = \sum_{i:x^{(i)} \in R_m} (y^{(i)} - O(m))^2.$$

Ideally, we'd want to minimize the loss $\sum_{i=1}^M E_m$, plus some kind of penalty on the number of partitions λM. (Regularization is important here, because otherwise we could put every point in its own region.) Unfortunately, this is hard to optimize: our game so far has been to optimize an objective function, but we're making discrete choices for our tree here. So it's not easy to differentiate in this situation (so we won't use gradient descent), and it's a combinatorially hard optimization problem.

So let's think about some stupid algorithms. One thing we could do is look at all possible partitions and pick the one that gives us the lowest loss. But enumerating solutions is also way too complicated, so we'll instead use a **greedy algorithm**.

Here's how that works. We start with $M = 1$ (so we consider putting everything in the same partition), and then we greedily (and recursively) draw new partition lines.

First, we build our partition.

Definition 80

For a data set D (of (x, y) pairs, for example, where $x^{(i)} \in \mathbb{R}^d$), let

$$R^+_{j,s}(D) = \{x \in D : x_j \geq s\}$$

be the set of points x where the jth coordinate is at least s. $R^-_{j,s}$ is defined similarly (it's the complement). Also, define $\hat{y}^+_{j,s}$ to be the average of all y-coordinates in $R^+_{j,s}$:

$$\hat{y}^+_{j,s} = \text{avg}_{i:x^{(i)} \in R^+_{j,s}(D)} y^{(i)},$$

and define $\hat{y}^-_{j,s}$ similarly.

With this, we can write down our algorithm pretty easily:

- If the size of our dataset $|D| < k$, return a tree with just a single leaf. (k is a parameter here: it tells us when we stop dividing.)
- Find a dimension j, and make a split that minimizes the error

$$E_{R^+_{j,s}(D)} + E_{R^-_{j,s}(D)}.$$

- To elaborate a bit more, a split is a real number: it looks something like "if $x_2 > 3.3$, go to one part of the partition ($R^+_{2,3.3}$), and otherwise, go to another part ($R^-_{2,3.3}$)." We don't need to consider all continuous values of x – only the $|D|$ x-values present in the data matter. (We'll generally pick the halfway points.)
- Once we've picked the best option, return a tree with vertex (j, s) and two children: the left child applies this algorithm on $R^-_{j,s}(D)$, and the right child applies this algorithm on $R^+_{j,s}(D)$.

This is fun to implement because it's not too difficult! What remains difficult here is picking k, and the question of regularization: how do we tune these parameters? It's tempting to try to just implement by picking our k, but it might depend on the sensitivity of our specific dataset. So one possible answer is to stop once we aren't improving very much: if none of the splits improves the error by more than some constant, then we terminate. But it turns out that the XOR structure has the property that the next split may not be great, but the one after that will be good: it's hard to decide **when to quit**.

Instead, what we often do is make the tree far too big. In our first phase, we grow a tree, and then in the second phase, we prune it a bit: we take our (possibly not balanced) binary tree, and we go from the bottom up and consider the objective function directly: now we greedily work our way up and see whether it's better to get rid of splits. For example, if two leaf nodes have similar numbers (or predicted averages), it may often be worth cmobining them.

Remark 81. *There's some variations on a theme: instead of just making an average value, we can throw in a linear regression, or we can make our splits not necessarily aligned with the axes! But this is more complicated.*

So now let's move on to classification. The main difference here is how we quantify the utility of a split: our output value O_m wil generally just be the majority inside a region, since it doesn't always make sense to pick the average of a bunch of discrete categories. Now, the error is a counting function:

$$E_m = \#\{i : x^{(i)} \in R_m, y^{(i)} \neq O_m.\}$$

Definition 82

Let $k \in Y$ be one of the classification options. Then let the proportion

$$\hat{p}_{n,k} = \frac{\#\{i : x^{(i)} \in R^m, y^{(i)} = k\}}{|R_m|}$$

be the approximate probablity of being in class k, given our region R_m.

With this, we can define an **impurity measure**: our error measure is no longer just a squared error. We want to ask the question of "is it good to divide here," and the idea is that we'll minimize the (weighted) impurity of the resulting areas. (We care about weights, because if we only have one element in a region, it's not a particularly powerful hypothesis that we're making.)

There's actually a few main ways we can measure the impurity:

- The **Gini index**

$$Q_m = \sum_k \hat{p}_{m,k}(1 - \hat{p}_{m,k})$$

  is a quadratic measure of error: for $k = 2$, it is equal to 0 at $p = 0$ or 1 and takes on a maximum at $p = 0.5$. This is actually often used in economics to measure a lack of uniformity.

- The **entropy**

$$Q_m = \sum_k \hat{p}_{m,k} \log \hat{p}_{m,k}$$

  (defining this term to be 0 at $\hat{p} = 0$) gives a very similar picture. Again, things with proportion p closer to $\frac{1}{2}$ have higher entropy.

- The **majority** is just the prediction error

$$Q = 1 - \hat{p}_{m,O_m}.$$

  This has generally been found to work less well, but it is valid too.

Either way, we look at this weighted entropy or Gini index to decide the best split, and the rest of the algorithm follows similarly to regression.

So why are trees good? The idea is that a human can make use of a tree: if our input variables are things humans can understand, like a blood pressure for a patient in the hospital, then a human expert could look at a decision tree and say that things make sense. So we can do inspection and get insight into the problem for a decision tree, whereas the problem is much harder for a neural network.

On the other hand, decision trees are **high variance**, which means they have high estimation error. This is the idea that we're never really sure that we've nailed things down, so we risk not having very good generalization. In particular, if we change our data a tiny bit, we might get a very different decision tree!

One strategy, which can be used to fix high variance problems in other learning algorithms, is a bit surprising: **bagging** or bootstrap aggregation. Let's look at this in the case of regression: we start by constructing B datasets of some size, say n, by sampling with replacement from D (so our datasets can have duplicates). We'll train our predictor on each of the small datsets b to get $\hat{f}_b$, a predictor for that small dataset. Then we just pick

$$f_{\text{bag}}(x) = \frac{1}{B} \sum_b \hat{f}_b(x).$$

It turns out that there's interesting statistical theory that shows that this does reduce variance! This is sort of like "dropping out" pieces of the data.

So if we do a mixture between decision trees and bagging, we get the idea of **random forests**. This is actually one of the most competitive machine learning algorithms, despite it looking pretty odd:

- Draw B bootstrap data sets, like above.
- For each of those datasets D_b, grow a tree by recursively doing the following: select m variables from $\{1, 2, \cdots, d\}$, and only allow splits along one of those dimensions.
- "Vote" the predictions of the trees, similar to how we average in bagging.

We'll finish with **nearest neighbor**, which is the simplest algorithm. The best thing here is that there is no learning algorithm at all: we just need to remember our data D. In linear regression or something else, we construct a hypothesis after a lot of work, but finding new predictions is quick once we have that hypothesis. The situation is reversed in this case.

Definition 83

A **distance metric** on an input space X is a function

$$d : X \times X \to \mathbb{R}$$

such that $d(x, x) = 0$, $d(x, x') = d(x', x)$, and the triangle inequality is satisfied: $d(x, x'') \le d(x, x') + d(x', x'')$.

WIth this distance metric, our hypothesis is easy to calculate, and it works for both classification and regression:

$$h(x) = y^{(i)} : y = \text{argmin}_i\, d(x, x^{(i)}).$$

So if we want the value of $h(x)$, we find the data point $x^{(i)}$ closest to x and output its y-value. Notice that this, in some ways, also partitions our input space: we get a decision boundary known as a **Voronoi diagram**, where each region corresponds to the closest data point. So there is indeed a hypothesis: we just never need to compute an explicit form for it!

Again, there's a few variations on a theme that make it a bit more efficient. If there's lots of points, this is a bit inefficient computation-wise: a data structure called a **ball tree** make the queries much more efficient. Also, we can do k-nearest neighbors instead of 1-nearest neighbor: if we want to be a bit more resistant to noise, we pick some (usually odd) k, look at the k nearest points, and make a prediction off of it. If we're doing classification, we just do a majority of those k points, and if we're doing regression, we can do a weighted local linear regression (so some points that are closer are more important than others).

One point to note is that all of this is very sensitive to the distance metric d. We started this class by talking about feature representation: the default distance we use is Euclidean distance, and this is okay if the deviation in each dimension is equivalent. So if we do this, we should make sure there is some underlying similarity in our coordinates! Optimizing distance metric can be an interesting cross-validation problem, and now some parameters might stick into this non-parametric method as well.

13 December 10, 2019

Today's lecture is being given by **Professor Jacob Andreas**. The last lecture will put together a lot of the tools we've learned about: it's about **building agents that can follow, generate, and learn from natural language instructions**.

In the real world (and also in science fiction), we're starting to have autonomous systems which can do things like take simple and not-so-simple physical actions, answer questions, transfer information, and so on. Our central question here: how would we build a machine that can **follow natural language instructions?**

There's a few different ingredients that are necessary for this, and they can be divided into two groups:

- Context: the environment and actions for the agent
- Data: the instructions the agent is working with, as well as the type of supervision.

In a way that may be familiar to us, we're going to model environments as a set of states. As a running exam, say that we're telling a machine to mavigate in a household: then each state can be a location and direction that the machine is facing in. We'll assume that in every state, the agent **can make some observation** of that state. The agent can also take **actions**, after which its state transitions. Since these actions can change some things about the world (such as if we turn on the lights), our state representation should reflect that as well. This looks like an MDP (which we've seen before already)!

We'll need some training data as well, but things are a bit more complicated here: there's many different ways to use a language like English to describe a set of instructions like "go forward, then face right." So we'll also need some supervision: that may be a reinforcement learning-type signal (where we're just told that we need to "find the sofa"), or it might give us the actual set of instructions to get there.

Fact 84
The point is that any algorithms we develop for this kind of problem need to be robust to the type of language that people are generating (because it's not necessarily predictable).

So in more formal language, our contextual information is a state space S and action space A, as well as a transition function $T : S \times A \to S$. Our data consists of some instructions X and a demo Y or reward R (either "here's the path to the sofa" or "you get a reward for getting to the sofa"). Our goal is to construct a function $f : S \times X \to A$, telling us what actions we should take given a set of instructions.

There's a few levels at which we can do this.

(1) First of all, if we take a string of words and try to convert them to low-level actions, that sounds like a machine translation problem (which we've thought about). If we're given a set of instructions, we can think of this as just a classification problem: all we care is generating the correct next actions. We've basically pushed everything about our problem into context: our instruction is part of our state observation, so it's being included in the MDP as well. So **different instruction-following problems are just different initializations in the state space.**

This classification approach works for a while, but it has some odd quirks. For example, "go through the door and end facing into the next room" means we need to know whether we've already gone through the first door. So we need a way to keep track of what parts of the instructions have already been followed. The next idea is that we have to **both track the "reading state" and physical state**, so we know where in the instructions we've read up to. One of the simplest is to create a second state space: every action consists of selecting some amount of the text to focus on. So at each step, we pick either an action in environment space or in reading space, and our current state depends on both spaces:

$$S = S_e \times S_r, \quad A = A_e \cup A_r.$$

This works on pretty cool problems, such as following Windows instructions or playing a game. But reinforcement learning can be a bit annoying, so we want to go back to classification. We said earlier that we can treat the translation problem as a many-to-many recurrent neural network: **our hidden state in this RNN can now tell us about the current reading state**! Basically, we can just let that reading space sit inside our RNN, and we can go back to using demonstrations (rather than reward functions) for this kind of problem.

(2) With this, we'll now move on to more challenging formulations: perhaps there's some kind of hard inference task that we need to accomplish that makes it hard to figure out these low-level actions.

Example 85
Instead of having a bunch of actions that correspond loosely to different parts of the text, we might do something like "find a table next to a chair."

There's secretly a lot of structure here: perhaps we can split up this sentence into "[find][a table][next to][a chair]." If we want to hear more about this, we should take the language processing classes offered here, but the point is that this structure can map directly into the **observations** that an agent makes. Modern computer vision problems can do things like pick out a table and chair, so the central idea here is to **predict constraints rather than action sequences**, and then a **planner** can do the rest of the work. Perhaps in our training data, we have indications of these constraints, and then when we make predictions on held-out data, we just assign an appropriate object ("table") or adjacency relation ("next to") to each [component] of the sentence. And if we have access to the right kind of supervision, this is not too difficult.

Fact 86
These kinds of systems can handle more active instructions as well, like "put the cup on the table." It just means we need to reach a slightly different (and more complicated) state space!

And we don't necessarily need to use this kind of logic from language structure: we can pick out our constraints by looking at our observations, too.

So we've talked about ways to do this instruction-following task, and we've looked at some different ways to act in complex environments. The key is that we need a model that can track the progress the agent has made so far, as well as plan ahead, learn cost functions, or reason about the outcomes of those actions. Our next question: **what else can we do with this?**

Once we can map from sequences of words to sequence of actions, we can actually add in a bunch of other capabilities as well. Let's start with the inverse problem of **instruction generation**: perhaps we want to turn the sequence-to-sequence problem (from a string of text to a sequence of actions) in reverse. It turns out that people give instructions in many different ways, so if we're trying to build a model that reproduces human instruction, we'll spend a lot of capacity trying to learn not-very-easy-to-follow instructions. The key is that **good instructions get us to our goal with high probability**, so we want to **minimize Bayes risk** (so make it unlikely that people end up in the wrong spot).

Our scoring function here uses an instruction follower, and this basically simulates a human listener: if we want to generate good instructions, we should search in the space of **instructions** rather than plans, until we find one given a high score by the instruction follower. And this can be layered, whether it's a neural network or a discrete state space system. A cognitive science approach here is that we're maximizing based on the **belief** of an instruction follower.

With this, we'll transition to another application about **machine teaching**. In the reinforcement learning part of this class, we stopped talking about generalization: all we cared about was that there was a reward function, and we wanted to find a way to do well with respect to that specific reward. (Maybe we just have three routes that we're really good at following, and we don't need anything else.) This can still be challenging, especially when we don't get any demonstrations! So let's go back: we can think of learning a bunch of mini-reinforcement-learning policies, one for each component of our text ("go forward" or "face the sofa"). This addition of language into our problem can help give us a lot more structure for our reinforcement learning problem (for example, if we're learning how to navigate a maze).

Another thing we can do is look back at how to make supervision better: when we teach humans, we don't just give rewards (that's what we do for dogs): instead, we also give corrections. So we give our models a few more chances and more context: they can observe not just single instructions but **histories of instructions**. So perhaps the agent takes some sequence of actions, and then it receives some feedback for its next sequence of actions. This allows us to construct a more meaningful dataset! This requires special supervision, but again, it's similar to our other models we've been discussing, and now we have a way to solve problems more interactively.

We'll now tackle a task that doesn't involve any language at all. Suppose we've trained a model that can follow these kinds of navigational models that we've been talking about, and say it's dropped in some new model. We have no real feedback here except for a sparse reward: what can we do? Well, we can **generate random instructions** and see what reward we get there, instead of **taking random actions**. Here, we are now doing exploration by using language as a pre-training scheme for reinforcement learning. This works well for **few-shot learning** as well, where we can only get a few training examples, because it gives us a natural way to generalize.

Example 87

For example, if we're training a model to play a game, it's been found that it's effective to generate instructions for our agent to follow, rather than just doing learning blindly.

To conclude, what are some challenges we are facing right now? The first step of learning, where we need to get a lot of data, can be difficult to collect: currently, instructions are being synthesized from a pre-defined grammar (it's scripted and fake). This is still an interesting problem, but it isn't going to generalize too well, so we're curious about how to **transfer between synthetic and real languages**.

Another major challenge is the integration of better planning machines: old-school symbolic planners can do lots of things that modern RNN-type models cannot do, so figuring out how to properly narrate problems is an ongoing problem.

The summary here is that we've looked at a set of general-purpose policy learning tools, and the key here is to remember what we've done and figure out how to align the structure of language with other aspects of our model.