How Machine Learning Works

by Paolo Perrotta

Course Overview

Course Overview

(Music) Hello, I'm Paolo Perrotta. Welcome to How Machine Learning Works, a short training that will take you all the way from scratch to building a real computer vision program. Machine learning is everywhere these days, but it's also an intimidating topic, even for experienced programmers because it feels so foreign, like these amazing sci-fis think that it's reserved for mathematicians and researchers. It doesn't have to be that way. In this course, I want to demystify machine learning and give you a technical idea of how it works. I'll tell you about supervised learning, which is the most important type of machine learning today. We'll write our very first machine learning program together, and then we'll evolve that code step by step until we have a short program that recognizes images of handwritten digits. And here is the best part. We're going to do that all by ourselves. No magical libraries, no AI tools. All of that code will be yours. You'll be able to read it, understand it, and experiment with it. We only have two requirements for this training. First, you should know some high school mathematics, or it's okay even if you have some recollection of it. Second, and most important, you should be a programmer. You should know how to read and write code. We'll use Python for our examples, so if you know Python, you're golden. Otherwise, no worries. Python is a friendly language. It's easy to read, so as long as you know any programming language, I think you'll be able to follow. By the end of this

Pluralsight training, you'll know how machine learning works. Modern artificial intelligence won't look like magic anymore. So, let's dive in.

Introduction

Getting to Know Machine Learning

Machine learning is amazing. Computers translating text or recognizing images, it's incredible stuff if you think about it. On the other hand, machine learning is also intimidating. Most machine learning tutorials involve a lot of mathematical formula, which can be off-putting, and when they don't involve math, those tutorials tend to be very high level. They teach you how to use a library as a black box, but they don't tell you how that library works, so the entire thing looks a bit like magic. So, machine learning is fascinating and intimidating and I'm making wild bet with this training. I'm betting that I can give you a good understanding of basic machine learning in about as long as you would take to watch a long movie, let's say below 2.5 hours. Of course I cannot make you an expert in 2.5 hours, but that's not the goal here. Our goal is to demystify machine learning, to remove the magic and get a concrete no nonsense idea of how it works. So, don't feel like you have to understand all of the details as you follow through. What's important is that you get the gist of what we say here. And of course if you have questions, you can ask them on this training's discussion board and I'll do my best to answer them. So, let's review our prerequisites for this training. You should know how to code and you should remember some high school math. Not a lot of it. I'll do my best to keep it intuitive and explain the math in plain English, plain English with an Italian accent. And that's it. If you've got these, then you have what it takes. Let's get started.

Programming vs. Machine Learning

Here's a story. Back in the '90s, a developer friend of mine was asked to code a software system for radiology. This system was supposed to look at x-ray chest scans and say which scans show pneumonia. I say it was supposed to because my friend thought that building that system was impossible. She didn't believe that she could code a computer program to do that, and I agreed with her. It's just too hard to teach a computer to read x-rays. So we thought that was a pie-in-the-sky idea that would never really work. And in fact it didn't. The project was canceled eventually. Now, fast forward to modern times. In 2017, a team at Stanford University built exactly

that, a system that diagnoses pneumonia from x-rays, and it does it better than a team of human radiologists. That was supposed to be impossible. So you might wonder, how could they write that code? And the answer is that they didn't really write it. You see, traditional programming is all about telling the computer what to do, so if you are writing a program to diagnose pneumonia, I'd imagine it would look something like this: instructions like look for opaque areas, look for their shape, that kind of stuff. And these instructions would be really detailed. They would tell exactly what to do and when because that's what programming is like. And programming works great for many things, but not for things that require human judgment like radiology. For example, how are we supposed to look for opaque areas, really? Which shape should they have? How do we recognize shapes anyway? The more you get into the details, the more you realize how hard it is to write this program because we don't know how a radiologist detects pneumonia. That's the truth. Yes, we know that at a high level of abstraction maybe, but we don't know the details of every step so we cannot code a computer to do those steps. So the people at Stanford followed a different approach. They used machine learning. In machine learning, you don't give instructions to the computer like in programming, you just give data to the computer, for example x-ray scans, and the computer learns by itself how to make sense of that data and what to do with it. But how, because that is the magical part, right? How can a computer make sense of the data?

Supervised Learning

There are a few ways to do machine learning to let a computer make sense of data. In this training, we'll look at one of those ways that is very important today. It's called supervised learning. Supervised learning is behind most of those amazing innovations in artificial intelligence. Here is how it works. To do supervised learning, you start from examples. Let's say that you want to build that Radiology system. What you do is collect a lot of examples, a lot of x-ray scans labeled with a flag that says this scan shows pneumonia and this one doesn't. So you need the humans to collect these examples and label them. And you need a lot of examples, maybe hundreds of thousands or even millions of them. Then you give the examples to an algorithm and the algorithm looks at the examples and understands the relation between the images and the labels, between the x-ray scans and the presence or absence of pneumonia. This is called the training phase of supervised learning. Once the algorithm is trained, then you can move on to the second phase of supervised learning, the prediction phase. And the prediction phase is when you get your money's worth. You give an unlabeled image to the computer, an x-ray scan where you don't know whether it shows pneumonia or not, and the computer predicts the label, like I can tell you, this scan does contain pneumonia. And that's how supervised learning works. Two phases, a

training phase and a prediction phase. During the training phase, the computer takes labeled examples and notices patterns in them, and during the prediction phase, it takes unlabeled examples and predicts their label. In general, supervised learning finds the relation between data and its label. Let's call them x and y. In our case, data is a chest scan and the label is a flag that tells us whether the scan contains pneumonia. But the data could be, for example, the day of the year and the label could be the number of hot dogs sold at a hot dog stand. A supervised learning program could learn how they relate and predict future hot dog sales. Or maybe x can be a sentence in English and y can be the same sentence in Indian. You'd need a lot of examples to pull this off, but if you, then congratulations, you just built Google Translate. Those are all valid examples of supervised learning. You can use supervised learning for many, many different things. Now, I do realize that this still sounds a bit magic, right? I said vague things like the algorithm learns the relation between x and y, it notices patterns, but computers don't notice things. So, how does the training process happen in practice?

Approximating a Function

Let me show you how supervised learning works in practice with a simpler example than x-rays. Let's say that you installed a solar panel on your rooftop and the solar panel is generating power throughout the day, and the amount of power changes with the time of day. So, that's our x and y for this problem. X is the time of day and y is the power generated. How do we predict the power generated with a supervised learning system? Well, we start by collecting examples. Maybe we take random measures for a few days like its 4 pm and the solar panel is generating 150 kW. Let's say that we collect a few of those examples and plot them on a chart, like this, showing time of day on the x axis and power generated on the y axis. If we plot all the examples, then we get something like this. For me and you it takes just a look to see that there is indeed a relation between time of day and power generated. Power is higher around noon when the sun is shining and it drops to 0 during the night. But how would the supervised learning algorithm look at this data and understand this data? What supervised learning does is it approximates these points with something simpler, with a function like this. So, that's what the algorithm does. It finds a function that approximates the data. This function is called the model because it's a model of the data, an approximation. But if this model is good enough, then the system can forget about the data and use the model to predict the future. For example, how much power is the solar panel going to generate tomorrow at noon? The model tells us that's going to be 260 kW, give or take. This approach works for this simple dataset, but it also works for much more complicated datasets like x-ray scans. Of course, the function that approximates the relation between x-ray

scans and the presence of pneumonia is going to be much more complicated than this one. In fact, it's so complicated that we couldn't draw it on a bidimensional chart like this one. But we'll get to that topic later. For now, just know that supervised learning always works like this. Whether you have a simple problem like this one or a much more complicated problem, it approximates the relation between the data and the label with the function and model, and then it uses the model to predict unknown labels. This is the trick behind the magic. So let's recap this first module. We said that machine learning is different from programming in that in programming, you tell the computer what to do step by step, while in machine learning, you pass data to the computer and ask it to make sense of that data. How? Well, we've seen one way to do that. Supervised learning, and we said that in supervised learning there are two phases, a training phase when you give labeled examples to the algorithm and the algorithm finds the relation between the examples and the labels, and then there is a prediction phase when you give an unlabeled example to the computer and ask it to predict the label. And finally, we saw how supervised learning finds that relation between the examples and the label. It does it by finding a model that approximates the examples, and then it uses that model to do prediction. But now we need to go one level of detail deeper. How exactly can a computer find that model, that function, that approximates the data? It takes an algorithm to find that function. In the next module, we'll get coding and write that data algorithm.

Building Your First Machine Learning Program

Tackling a Machine Learning Problem

In this module, we'll build a supervised learning program. A pretty limited program, but it will be a starting point. We'll make it more powerful later. To write this code, we're going to use Python 3. Again, don't worry if you don't know Python, as long as you get the gist of what the code is doing, you'll be fine. We're also going to use a few libraries. These are not specifically machine learning libraries, I promised we won't use those, but they are popular libraries in the machine learning community. One is called Jupyter, also known as Jupyter Notebook. To be precise, Jupyter is not really a library, more like a web application that's distributed as a Python package. Essentially, it's a system to write Python code in a web page and execute that code from the same page. You'll see it in a minute. Second, we're going to use NumPy. That's a numerical library.

It makes it easy to do sophisticated operations on arrays and matrices. And finally, Matplotlib. That's a library for plotting charts. You might want to follow along and run the code yourself. That's optional, but if you want to, then you'll find a README file in this training's downloadable code with instructions to install Python and the libraries. Okay, let's start. Here is a concrete problem. Imagine that we have a small restaurant, a pizzeria. I like food-based examples. And every afternoon, we get a certain number of reservations and every night we sell a certain number of pizzas. And these two numbers are probably related, right? Not a simple one-to-one relation, it's not like we sell exactly one pizza for each reservation. But we do expect that there is a relation, so that's our X and Y for this supervised learning problem: reservations and pizzas. We want to write a program that learns how they're related, and then uses that knowledge to predict pizzas from reservations. That's our goal. So, for example, we might say we got 25 reservations this afternoon, and the program would say, okay, so you can expect to sell, I don't know, 40 pizzas. And once we know that, we can prepare dough for 40 pizzas and avoid preparing too much or too little, which would be a waste of money. This is supervised learning, so we should start from examples. I collected a few in this file. I'll open it in my text editor. It has two columns for Reservations and Pizzas recorded on a few random days. On this first day we had 13 reservations and we sold 33 pizzas, and so on. Each line in this file is an example of this relation that we want to predict. And we have a few tens of examples. I mentioned Jupyter Notebook before. Let's use it.

Making Sense of Our Data

I'll start the Jupyter from the same directory where I have the pizza.txt file with this command, jupyter notebook. That opens the Jupyter interface in my browser. This is the directory that we started it from. Here is the pizza file. And from here I create a new Python 3 notebook, which is essentially a web page that contains code. This is called a cell. It's a section inside this notebook, and I can write code in it and execute it. I'm a programmer, so I usually don't use Jupyter in my personal work. I'd rather write my code in an IDE like Visual Studio or even a regular text editor. But many machine learning professionals use Jupyter all the time, so let's go with the flow and use Jupyter ourselves. It's a pretty friendly tool. By default, Jupyter cells contain code, but in this case I want the first cell to contain a title for our little project using the Markdown text format. There we are. This sharp sign means format this as a title, and when we run the cell it gets formatted. Now you start to see what's the point of Jupyter. You can mix text and code in the same notebook so you can explain in detail what the code is doing. That's a useful feature when you do machine learning. Okay, now let's do some data analysis. Here is what I want to do. I want

to load the data from this file into two variables, reservations and pizzas. We can do that with the NumPy library. So let's import NumPy. It's usually imported like this. This means import NumPy, but don't force me to use its full name every time I call one of its functions. Instead I will use this shorter name. The reason I imported NumPy is that it has this function called loadtxt that I can use to load data from a text file. I have to give it a couple of parameters. One tells loadtxt to skip the first row because the first row contains the headers, not data. And the second parameter unpacks the data into two separate columns. Otherwise we'd get all the data into one big matrix. Instead, it's easier to have two separate arrays for the reservations and the pizzas. Let's call them X and Y. Of course, don't feel like you have to remember all this stuff. I didn't remember the details of this function myself. I just looked it up. So, let's execute this code and we end up with two arrays. X, the reservations, see, 13, 2, it's the data from this column here. And Y is the pizzas, 33, 16, this column. Okay, we expect that there is a correlation between X and Y, but that's hard to say by looking at all these numbers, so let's plot them. Let's generate a diagram of reservations to pizzas. You can create this chart anyway you like, you could load the data in a spreadsheet for example. I will use the Matplotlib library. By the way, if you're wondering how I can type this fast and never make a mistake, that's because I'm cheating. I'm actually editing away my mistakes and speeding up this video. I'm nowhere this good. About this code, I will not explain the plotting code in this training. Learning Matplotlib is not the point here. However, in the downloadable version of this notebook you'll find comments to help you understand how this code works. So if you're curious about Matplotlib, there is that. Otherwise, just know that this code plots a chart of reservations versus pizzas. Let's run it. Boom! Each of these points is an example, and now we can really see the correlation, right? It's jumping at us. More reservations generally mean more pizzas sold, as we expected. Let's write a supervised learning program that learns this correlation.

Understanding Linear Regression

In the previous module, we said that supervised learning finds a function to approximate the data. Now, let's see a concrete example of that. Here is our pizza data, how can we approximate it? Well, these points are roughly aligned, so we can approximate them with a relatively simple model, a line, like this. And then we do the supervised learning thing. We can forget about the data and use the model to make predictions. We have these many reservations. How many pizzas do we expect to sell? It will be these many pizzas. Now, this idea of using a line to make predictions goes back to way before supervised learning. Statisticians have been using it for a long time. It's called linear regression. Regression means predict a variable from another variable, like pizzas from reservations. And linear means that we're using a line as opposed to some kind of

curvy function. One good side of linear regression is that a line is simple, simpler than most functions, at least. If you look up the mathematical equation of a line, it looks like this, y = y * w + b. Now, maybe you studied this equation in high school and maybe you called these two parameters with different names, m and b or a and b. It depends in which country you attended school mostly. I call them w and b here because they're short for two machine learning names. W stands for weight and b stands for bias. I know they're weird names, but the machine learning vocabulary is like that. A bit weird. You don't really need to understand intuitively what w and b are, but in case you want to, w is a measure of how steep the line is. So if w goes up, then the line gets steeper like this. And if it goes down, the line gets less steep. And b is a measure of this distance, the point where the line crosses the vertical axis. Maybe you called it the y intercept back in school. If b increases, then the line stays just as steep, but it moves upwards, and if b decreases, it moves downwards. It's nice to understand intuitively what these parameters mean, but it's not really necessary. The crucial piece of information here is this equation that represents the line. That means that if we know w and b, then we know the line. In a sense, the line is w and b. It's uniquely identified by them. So, if we decide that our model is a line, then we can represent it in code with two variables and call them w and b. So let's write that code.

Implementing Prediction

So here are the two phases of supervised learning again. We decided that we're going to use linear regression, so our approximation function, our model, is going to be a line. So, the training phase finds the parameters of the line, w and b, and the prediction phase uses them to make predictions. We have to implement both phases, but instead of starting from the training phase, let's start from the end, from the prediction phase, just because it's easier. To do prediction, we should apply this equation, the equation of a line. I'll write that code in our notebook. We want to define a function, let's call it predict, that takes a number of reservations, x, and it also takes the line as two arguments, w and b. And then it applies the formula and predicts the number of pizzas. And we're done. That's all we need to make predictions actually. Of course we need to know w and b, but finding that line is the job of the training phase. Once we have w and b, we can make predictions. For example, let's say that we got 14 reservations and we found out during training that w is 1.2 and b is 12. I just made up these values. How many pizzas are we expecting to sell? We call predict and it says expect to sell about 28.8 pizzas. Maybe we don't sell pizzas in fractions, so it's 29. But the exact prediction is 28.8. There is something I want to show you about this function. See what happens if x is not just one number, but an array. For example a NumPy array that contains three reservations for three different days. In this case what happens is that

these operations inside predict, the multiplication and the sum, are applied to all the elements of the array independently, and the result is another array with three predictions. For 14 reservations we expect to sell 28.8 pizzas like before, for 5 reservations we expect to sell 18 pizzas, and so on. This is called broadcasting. It's a feature of NumPy. It's the idea that you can apply the same operation to all the elements of an array in one shot. It's a useful feature as we'll see in a few minutes. So now the second phase of linear regression is done. If we have a line, if we have w and b, then we can predict the pizzas from the reservations. Let's look at implementing the first phase, the one that finds the line.

Understanding the Loss

The training phase of linear regression is trickier than the prediction phase. It's not super hard, but it does require a few steps, so I'm going to need your full attention for a few minutes. First, remember what we want to do during the training phase. We get a bunch of examples, that's our input. And we want to find a model, a line that approximates those examples. Or to put it differently, the inputs are the examples from the pizza.txt file and the outputs are the parameters of a line that approximates them. How do we find those parameters? Well, let's start with an observation. Whatever line we find, it's going to be wrong, right? A line is an approximation of the examples, and unless the examples happen to be perfectly aligned to begin with, which would be very lucky, the line must have an error associated with it. We don't want to find a perfect line. There is no perfect line. We just want to find the line with the lowest error. So, let's find a way to measure that error to stick a number on it. For example, let's say that we have this line that has these values of w and b. These are the same values I used in the Jupyter Notebook before. And when we used this line to predict the number of pizzas for 14 reservations, we got 28.8 pizzas. That's what the line said. But if we look back at our file of examples, that's not what the real data says. On this day here we got 14 reservations and we sold 32 pizzas, not 28.8. We can even find this example on the chart, it's this point here. So there is a difference between our prediction and the real data. I marked it in orange here. Let me make it more visible. This orange distance is the error in our prediction for this point. If we use the line to predict the number of pizzas for all our examples, then all those predictions will have an error. The error is smaller if the point is close to the line, like this point here, and bigger if the point is far away from the line, but unless a point is exactly on the line, it will have an error. Let's recap where we got so far. We're trying to find a way to measure the error of a line. So far we did this for a single example. Now let's do it for all the examples we collected. We loaded them into this X array. Let's calculate the line's prediction for all of them. Remember that broadcasting feature, NumPy can calculate all these predictions in

one shot. Then let's take this array of predictions and we subtract the real values, Y. And the result is an array that contains the prediction errors for all the examples. And now we can calculate the average of these errors with NumPy. We're almost there. There is still something wrong in this formula because some of these errors are going to be positive and some are going to be negative, right? Depending on whether the prediction is bigger or smaller than the real number of pizzas. And if you think about it, it doesn't make much sense to have negative errors. We don't want negative and positive errors to cancel each other out. So we should make all these errors positive. We could take their absolute value, but people usually just square them so they all become positive. Let's do that. The double star is the power operator in Python, so this means squared. So now we have this code, and let's wrap it in a function. This is called the mean_squared_error because that's what it is, it's the errors, squared, and then averaged. In machine learning lingo, this error is also called the loss. Let's use that more specific name. Loss means approximation error, basically. Okay, it took some effort, thank you for sticking with me so far, but look at what we have now. We have a function that takes our model, a line, and it takes the examples, and it calculates the error of the model at approximating the examples. The lower the loss, the better the model. We want to find the model with the lowest loss. How can we do that?

Implementing the Training Algorithm

Here is one way to find the model with the lowest loss. We could use an iterative algorithm that works like this. Start with a random line, two random values for w and b, and calculate the loss of this line. This little gauge here shows the loss. It's pretty high. This line is a bad approximation of the examples, so it has a high loss. Now find the line with the same w as this line minus a small amount. That amount is usually called lr, that stands for learning rate, another weird machine learning name. So, this line is less steep than the original line because it has a lower w. In reality the difference would be really small. I exaggerated it here to make it more visible. How is the loss of this new line? Is it higher or lower than the loss of the original line? We can calculate it, and in this case it's higher, so this line is even worse than the original line at approximating the examples. So it seems that we're going in the wrong direction, the direction of increasing loss. So let's try to make w a bit larger instead, incremented by the learning rate so the line becomes steeper. What happens to the loss? Well, this time the loss decreased so this line is a bit better than the line we started from. So, this line with a lower loss becomes our best line so far. And then, we do it again. Once again, consider the line that's a bit less steep and the line that's a bit steeper. Which of these three has the lowest loss? This one. Okay, this one becomes the new line. W + lr becomes

the new w. Repeat this process a few times, and at some point you get to a situation where you cannot make the line better by changing w. For example, this line. If we decrement w or increment w, in both cases we get a higher loss. When that happens, we can try the same approach with the b parameter. Try the line with a slightly bigger b and the line with a slightly smaller b. Does either line have a lower loss than the current line? If so, that becomes the new line, and so on, you catch my drift. At each iteration, we try lines with slightly different values of w and b, and every time we keep the line with the lower loss, with the least error. And when we cannot push down the loss anymore, that's our best line, the line that approximates the data points. This algorithm is a bit like a radio operator looking for the best signal, rotating two knobs called w and b slowly in the direction where he hears a better signal, a lower loss in our case, until he gets the best signal he can get. So, I went straight ahead and implemented this algorithm in code. Here it is. I called this function train because it's our training phase. It takes the examples and two more arguments, the maximum number of iterations, and the learning rate, lr. We'll see how to use them in a moment. The algorithm starts with a random line. I just set both w and b to 0 because it's as good a starting line as any. Then we get into a loop that iterates this number of times. The idea is that if we don't find our line within this number of iterations, then maybe the algorithm might be stuck and oscillating between the same two lines forever. So, let's stay on the safe side for now and just fail. We raise an exception. Now let's see what happens at each step of this loop. First, we calculate the current loss, the error of the current line. Then we log our progress on the screen, we print out the iteration and the loss, and now comes the juicy part. We try to decrement w by the learning rate. Does that result in a lower loss? If so, then this value becomes the new w and we go to the next iteration. Otherwise, we try to increment w by the learning rate. And if that doesn't work, we try decrementing b and finally incrementing b. And if none of these work, that means that as soon as we change either w or b, the error becomes bigger. So the line that we have now is the best that we can find and we return it. So it took us a while, thank you for your patience through this lengthy explanation. But now we implemented both phases of supervised learning. We have a train function that finds the model and we have a predict function that uses it. Let's see what happens when we run this code.

Running the Code

So, let's run train. These are the examples and this is the number of iterations, say 10, 000 iterations, and this is the learning rate. Let's make it small so at each step we move the line just a little. So we might be slow at approximating the data points because the line is moving slowly, but in the end we should get a pretty precise approximation. Let's run it. See what happens? At

each iteration the loss drops a little. The algorithm is moving the line, turning the knobs like that radio operator to decrease the loss. After about 1, 500 iterations it stops because it cannot decrease the loss any further. As soon as it changes either w or b, the loss raises. And that's the line we were looking for, these values of w and b. I'll write some code to plot this line on the screen. Once again, we don't need to linger on plotting code, so I'll use my magical time accelerator. Let's run it, and here is our line. It looks like a decent enough approximation. And now that we have this line, this w and b, we can use them for prediction. Let's say that we have 42 reservations. How many pizzas do we expect to sell? It's going to be a bit more than 59 pizzas. And that's it. And if you think about it, we've built something really cool, a system that learns from examples and predicts the future. Yes, it's very simple, but we'll make it much more powerful during the rest of this training. Here's a look back at what we did in this module. We started by looking at our examples and we decided to use linear regression to approximate the examples with a line. And then we wrote a supervised learning program composed of three functions, loss that measures the error of a line, train that finds the best line by minimizing the loss, and predict that uses the line to predict pizzas from reservations. However, I've got to be honest with you. This train algorithm I wrote, it's not very good. It's actually pretty bad. In the next module, we'll see why it's bad and how to make it better. We'll replace it with a different algorithm that's faster and more precise, and it just so happens to be the most important algorithm in machine learning. See you in the next module.

Improving the Algorithm with Gradient Descent

Our Algorithm Doesn't Cut It

Welcome back. In the previous module, we wrote a learning program. In this module, we are going to improve on that program with a new algorithm, one of the most important algorithms in machine learning, gradient descent. Why do we need a new algorithm? Let me show you. I copied the code from the previous module in the new notebook here. Look back at the train function again. This function finds a line that approximates the data points, right? It does that by trial and error basically by decreasing and then increasing each parameter of the line, first w, then b. And picking the first of those changes that gives us a better line, a lower loss. And this approach kind of works as we've seen, but as it turns out, it isn't really correct. The reason why it's not correct is

that we don't really know whether we can find the best line or even a very good line by changing w and b separately. The loss depends on both parameters, so maybe at some by changing w, we might increase the loss caused by b, or the other way around. In other words, if we want to find the best line or close to the best line, then we shouldn't change w and b separately. We should change them at the same time. At each iteration, we should try all the possible combinations of changes to w and b. Decrease w and leave b unchanged. Then decrease w and decrease b. Decrease w and increase b. And so on. And at each iteration, we should calculate the loss for each combination and take the combination with the lowest loss. Unfortunately, we cannot do this. I mean, we could do this in our case because we have a linear model with two parameters, and if you do the math, that's nine possible combinations to try, nine losses to calculate. That's not a big deal, but we couldn't do this in general because the number of parameters in a supervised learning system can grow a lot. Even in this basic training, we are going to see models with hundreds of parameters and there are models out there with millions of parameters. And as you add parameters, the number of combinations grow really fast. If we had to calculate the loss for every one of those combinations, it would take forever, even on a very fast computer. So these algorithms we wrote, it needs fixing, and we could fix it for a simple model with two parameters, but not for a large model. It would be too slow. So we need to find another training algorithm.

Understanding Gradient Descent

To find a new training algorithm, let's start by visualizing the loss by seeing how the output of this function changes when its symbols change. I just added some new code to this notebook that plots a chart. As usual, I won't explain the details of plotting code. The downloadable version of this notebook has comments in it in case you want to understand this code, but that's completely optional. You just need to know what this code does. It plots this chart that shows how the loss changes as w changes. Note that I made a simplification here. The loss depends on w and b, but to plot this chart I set b at a constant value of 0. Now the loss only depends on w. That's only a temporary thing, I will reintroduce b later. For now, I already moved b to simplify things as I explained this algorithm. So with this model, a line, and this formula for the loss, the mean squared error, we have this nice smooth loss curve. This is the loss for these values of w. Now, remember what the train function is about. It's about finding this point, the value of w and b eventually, that results in the minimum loss. Here is a new concept that is going to come useful to find this minimum. It's the idea of the gradient of a curve. The gradient measures how steep a curve is at a certain point. For example, the gradient at this point would look like this. By mathematical convention it points uphill, and it would be a relatively large number at this point

because the curve is quite steep here. While this other point, for example, the gradient would still point upwards and it would also be smaller in absolute value than the previous gradient because the curve is less steep here. It's not as heavily sloped. And right here at the bottom of the curve in the minimum point, the gradient is 0 because the curve is flat. It doesn't have any slope. So that's what the gradient is, and in a few minutes we'll see how to calculate the gradient. For now, just assume we can calculate it, and here is an algorithm that uses the gradient to find this minimum point here. The first step is start at a random point on the curve. A random value of w, for example, here. Then take the gradient of the curve at that point. The gradient shows us where the uphill direction is and we want to move downhill to reach the minimum. The minimum is like a river, it's at the lowest point in this valley. So we have to reverse the gradient to go in the opposite direction, and also we don't necessarily want to take this large step towards the minimum. We want to be more precise, so we can multiply this gradient by some small number so that it gets smaller. And then we take this result and we add it to the current w, like this. We just took a step downhill, and then we do it again. Calculate the gradient, it's smaller at this point because the curve is less steep here, invert it, make it smaller, take another step. If we keep doing this, eventually we'll get to the minimum. Actually, we might overshoot the minimum, for example, from here, we could take another step, calculate the gradient, invert it or reduce it, step, and we just breezed past the minimum to the other side of the curve. But this algorithm takes care of that because now the gradient is pointing in the opposite direction and it's very small because the curve is pretty flat here, so when we invert it and reduce it and take a step, we end up close to the minimum. And we can take as many steps as we want like that. The more we take, the closer we get to the minimum. The more precise we get. This is the algorithm called gradient descent. It's only about descending along the gradient of the loss curve, rolling down the curve like a little ball until we reach the minimum. Or rather we get very close to the minimum. This is not a trial and error algorithm anymore, it's much more directed. We move downhill straight towards our goal, towards the optimal value of w. It's not a perfect algorithm, it doesn't work for every possible curve. For example, if we had a loss like this, then gradient descent might get stuck in this hole with the orange cross. This is what is called the local minimum. A place where the loss is lower than the loss is surrounded, but not the lowest overall. That's the green cross, the global minimum. Or we might have a loss curve like this with a flat area. This area has a gradient of 0 because it has no slope at all, but it's not a minimum. So if gradient descent gets here, then it gets stuck. It cannot move downhill because it cannot know which direction downhill is. And so on. There are situations where gradient descent doesn't work well. It depends on our model and the loss function, but you don't have to worry too much about that in practice because machine learning researchers already did a lot of work for us and they found a few models and ways to

calculate the loss of those models that generate a good loss curve, like this one that are less likely to have local minima or flat areas. In our case, we're using linear regression and we're calculating the loss with the mean squared error, and we know from machine learning literature that this is a good combination. It gives rise to a loss curve that works well with gradient descent. So, that's how gradient descent works when we only take into account the w parameter, but our model also has a second parameter, b. So what happens if we consider b as well?

Descending the Gradient in Three Dimensions

To visualize the loss based on both parameters w and b, we need a 3D chart like this with weight w and bias b and the vertical axis, that is the loss. And the mean squared error of loss would look something like this, a surface. Every point on this surface identifies a line, a specific value of w and b, and its loss. And once again, we want to find the minimum that in this case might be here, these values of w and b. We want to get this minimum loss with gradient descent. So the first thing we need for gradient descent is the gradient, and the gradient of a surface is the same concept as the gradient of a curve, an arrow that measures how steep the surface is and it's pointing directly uphill. And as it turns out, this gradient is the combination of the gradients along the directions of w and b. Intuitively, imagine slicing this surface in two directions. First like this in the direction of w, so w varies and b is a constant. This slice that we got is like the card that we had before when we set b to 0. So, we can get the gradient of this curve in this point. We don't know how to calculate the gradient yet, but we will soon, bear with me. And then we can slice in this other direction, the direction where b varies and w is a constant, and calculate this gradient as well, and the gradient of the surface is the combination of these two gradients. That's the intuition behind getting the gradient of a surface. So in gradient descent, to take a step along the gradient of the surface, we can take a step along the gradients of w and b. So as usual, we should invert these gradients to point downhill, multiply them by a small constant, and take a step along both directions, that is change both w and b separately. And there we are, we just took a step downhill. And if we do this again and again, just like a bowl rolling over these smooth loss surfaces, we'll get close to the minimum. The same thing that we did before, only in three dimensions. So, now you know how gradient descent works conceptually. However, we have one less piece missing before we can implement it. We need to be able to calculate the gradient of those curves, and I didn't tell you who to do it yet.

Calculating the Gradient

Let's see how to calculate the gradient. This is the last thing we need to do gradient descent. Here comes the formula. It might look intimidating if you're not used to this notation, but it's actually the same formula that we used to calculate the loss, the mean squared error, only mathematical form. This is the prediction, then the difference between the prediction and the actual label, that is the error on a single example, squared and averaged over all the examples. This part here is the average, the sum of all the errors divided by the number of examples that I called m. To calculate the gradient of this loss, we can use a mathematical tool that's called the derivative, or more precisely, the partial derivatives of L with respect to w and with respect to b. This is the notation. Now maybe you studied derivatives in school, and if so you might want to calculate these derivatives on your own, but if you don't know how to do that, no worries, I can give you the results straightaway. And it's in all machine learning textbooks anyway, so you certainly don't need to calculate these derivatives yourself. These are the derivatives, and we can use this formula to calculate the gradients along w and b. This discussion might feel abstract, so let's make it concrete. Let's look at code. I wrote this function called gradient. It calculates the gradients of w and b using the two formulae I showed you earlier, the partial derivatives with respect to w and b. I just converted them to code. So if you understand the math behind this formula, that's great, but if you don't, all you need to know is that this code calculates the gradient and then it returns both gradients. So I could say give me the gradient of the loss for our dataset at this point and coming up with a random point here, the line where w is 2 and b is 3. And I get the components of the gradient along w and b. And now we can use these to do gradient descent. Let's do that at long last.

Implementing Gradient Descent

Let's go back to our old training function and modify it for gradient descent. To begin with, gradient descent starts from a random point, that is random values of w and b. It doesn't necessarily have to be random, it can be some point, so let's skip this line of code here and start at the point where w and b are both 0. It's as good as any other starting point. Then we get into the loop. This line we can also keep, and I'm also saving these two lines that print out the iteration and the loss. They're still useful. Maybe I will just in line this function call, but the rest of this code is going to change. So let me get rid of it and replace it with gradient descent. First, calculate the gradient components for w and b. Then, remember what we do. We invert the gradient like this. First the w component. Multiply it by a small constant. We can reuse the learning rate for this purpose. It used to mean something different earlier on, but it's still a similar concept. The bigger the learning rate, the bigger steps the algorithm takes. And we take a step in this direction. That

is, we increment w by this amount. We can actually write this in a more Pythonic way, like this. So, invert, scale, take a step, and we do the same for b. And we repeat this for this number of iterations and then we return w and b. And that's it. This is all the code we need for gradient descent. Let's run it with say 10, 000 iterations and a small learning rate for precision. And when we run it, boom, you can see the loss decreasing, and in the end we get a line, w and b, and we can use it to make predictions. If you compare this loss with the one we had before, you'll see that it's a bit lower. That's a good thing. Gradient descent finds a better line than our previous algorithm. And if we let it run for more iterations, then we'd find an even lower loss if we wanted to and gradient descent is also faster than the previous algorithm. That precision and speed maybe don't make a difference when it comes to selling pizzas, but once we get to more complicated models and data, that extra precision and speed can make all the difference. So, here is a recap of this module. We explained why our first version of training didn't really work well, and we turned to a better algorithm, gradient descent. And the idea of gradient descent is that we can start at a random point on the loss surface, random values of w and b, see where the gradient is pointing, and take a step in the opposite direction. We always move downhill like a ball rolling across the loss surface until we reach the minimum, or rather we get close enough to the minimum. And this algorithm doesn't work on every surface, but it works on many surfaces including the mean squared error loss that we're using. So, now that we have a good training algorithm for linear regression, that opens a lot of opportunities. We can move on to much more interesting problems. So far we had this toy example with pizza predictions, but real-life examples are usually more complicated than that. In the next module, we'll graduate to a more advanced program that could actually be useful in the real world.

Expanding Regression to Multiple Variables

Dealing with a Complicated World

Welcome again. In the previous modules you learned how to predict one variable from another with linear regression. Pizzas from reservations, a simple example. Even a bit simplistic, you might think, and you'd be right, honestly. In the real world, the number of pizzas sold wouldn't depend just on one thing, like reservations. It would probably depend on other variables as well, like the weather, maybe. If it's warmer, then there might be more people in the streets hungry for pizza.

Or say the number of tourists in town on a scale from 1 to 10. Maybe you scrape this number from the website of the local tourist office. For most interesting real problems, the value you want to predict depends on multiple inputs. In this module, we'll see how to do linear regression with multiple inputs. We are turning our toy program into something that you could actually use to solve real life problems. That will be a major step forward on our way to a computer vision program. Disclaimer, this is going to take a while. In fact, this module might be the hardest module in this training. As usual, don't feel like you have to remember everything on your first view through. This training is just an introduction to machine learning, so as long as you understand these ideas, you'll be fine. You certainly don't need to memorize them. And on the positive side, if you get through this module, then the rest of the training will feel easy, so buckle up. First, let's start with the linear regression we already know of one input, and see what happens if we add more inputs. You know that linear regression is all about approximating the examples with a line. If we add a second input variable, like temperature, then we must add a dimension to this chart, so the examples are not points on the plane anymore. Now they are points in a 3D space, like this, with these two inputs, like reservations and temperature, and one output, like pizzas. And to approximate these points, we cannot just use a line. We need the equivalent of a line, only with one more dimension, a plane. So our model would be a plane, and during the training phase the system moves this plane to try and approximate these cloud points. For the line model, we had this equation here. Now we added another dimension to the model, so we need one more variable in this equation, like this. This is the equation of a plane. It has two input variables, each with its own weight, and a bias. The bias plays the same role here that it did before. By changing the bias, we move the plane up or down. And by changing the weights, we rotate the plane around. So by changing these three parameters, the training phase can find the plane that best approximates the examples. We don't have to stop there, we can keep adding inputs to linear regression. This is also called the multiple regression. Only if we have more than two input variables, we cannot draw the model anymore. We'd have to draw a multidimensional space and our human brains can only visualize three dimensions. But we can still use mathematics to represent this higher-dimension space. We just add new terms to this formula, X3, W3, X4, W4, and so on. And this is the equation of a straight shape, a linear shape in mathematical terms, like a line or plane, except with more dimensions. It's also called a hyperplane. That sounds pretty cool, so there is that. So to recap, with one input variable we approximate the examples with a line. With two input variables, we use a plane. And with three or more input variables, multiple regression, we cannot visualize it anymore, but it's the same basic idea of approximating the points with a linear shape. And the more inputs we have, the more parameters these linear shapes have. But we can still use gradient descent to find those parameters. By the way, before we

implement this model, there is a little trick that we can use to simplify it. That's a nice one, let's talk about it.

Tricking Away the Bias

Here is our model again, and it looks okay, right? It looks quite uniform, except for one special case. That is the bias, b. Let me put it here in the beginning of the formula. As a programmer, I don't like special cases, they make it harder to write code, so this b parameter is annoying because it's the only parameter that doesn't work like the others. But there is a neat trick to turn the bias into a parameter like the others, like the weight. To begin with, we can rename b to w0 to mean this is just another weight. But other weights are multiplied by an input variable, and w0 is not. To fix that, we can invent a new input variable called x0, and we can decide up front that x0 is always 1 because then we're always multiplying w0 by 1, and that's just w0. And now this formula really looks uniform and nice. This might look like a silly trick, but it's useful because now we don't need to have a special case for the bias anymore. Instead, we can just go to our examples and add a new column with a constant value, 1. I made it the first column. It might be in any position. Doesn't really matter. And then our model just becomes multiply each variable by weight. Very consistent. That makes it easier to implement, as we're about to see in a minute.

Switching to Matrices

Here is the code of our linear regression program from the previous module, the code that loads data, the prediction code, the loss, and its gradient, and the train function that uses gradient descent to find the parameters of the model. Let's see how we can upgrade this code to multiple regression, starting from predict, because that's the function that implements our model. Remember how predict works so far. It takes x, that is a reservation, or an array of multiple reservations, and the parameters of the model that so far was a line, and it applies the model to make a prediction, or one prediction per reservation in the case where x is an array of reservations. Now that we have multiple input variables, the first thing that changes is that x becomes a matrix. Let me show you. Look back at our data. X was an array of reservations, right? Now x is a matrix with one row for each example and one column for each input variable, plus a column full of ones for the bias, so four columns in this case. Another thing that changes is that w isn't a single weight anymore. Now w is one weight for each column in x, so for weight. And b just doesn't exist anymore. It became one of the four weights. I can delete it. So what we need here is an operation that combines x and w, like this, our new model. As it happens, there is a

mathematical operation that does exactly this, as long as the input data counts in a very specific shape. This operation is called a matrix multiplication. Let's see what it looks like. To begin with, let's start from the case where x is only one example, so it's a matrix with one line and one column for each input variable. This is the bias, the reservations, the temperature, the tourists, and this is w, that also has one weight per input variable, so four weights. And if we multiply these two matrices, then the result is a matrix with one element that contains exactly the operation we want, x0 by w0 + x1 by w1, and so on. However, I just lied a little. This matrix multiplication isn't strictly correct because the mathematical convention tells us that this multiplication can only happen if w has a different shape, like this. So to multiply x and w, we need to transpose w to turn it around, like this, so that it has one column and as many rows as x has columns. This is the basic condition so that two matrices can be multiplied. So, NumPy does have a matrix multiplication operation, but we'll have to shape w this way before we can use that operation. So for this case where x has only one example, matrix multiplication is exactly the implementation of the multiple linear regression model. And even better, it still works if x is a matrix with multiple rows. In that case, matrix multiplication does the same thing for each row, and the result is a single-column matrix that contains one result per row, one prediction for each row in x. So to recap, if x and w have these shapes, then we can just use matrix multiplication, that is a single operation in NumPy called matmul, and boom, that's all we need. This is a ready-made implementation of our model, as long as x and w have the right shapes, so let's make sure they do.

Shaping Data

Let me recap what we are doing here. We had this predict function that is ready to run except that it wants data in a specific shape that respects the mathematical conventions of matrix multiplication, this shape. The input variables should be a matrix of one row, for example, and one column per variable. The weights should be a matrix with one column and as many rows as the input variables. And the result, y, the predictions, should be a matrix with one row per example and one column. I remember when I was learning this stuff for the first time, and I won't lie, to me, matrix dimensions were really frustrating. I lost count of how many programming errors I got because I was getting the dimension of my matrices wrong, confusing rows with columns, and so on. Maybe that's just me. Anyway, once you start using machine learning frameworks, those frameworks go a long way to make those calculations easier. But here we are writing code at very low level, so we need to be extra careful that all the matrices have the right dimensions, so let's reshape them starting with the code that loads the data. So far, our data file had two columns and we loaded them into two arrays, x and y. Now we have four columns, so we can load them

into four arrays, like this, and then we can use this useful NumPy function called column_stack to merge these first three arrays into the columns of a matrix. And the result is x. See, one row per example and one column per input variable. It's the same data that we had in the original file. We're not done yet. X is still missing the bias. Remember, we want the bias column filled with ones. We could add that extra column straight into the original data file, but in general it's a good idea not to meddle with the original data. Instead, let's create this extra column right here in the code as an array of 1s with this NumPy function called ones. We need to delete the size of the new column. That must be the same as any of the existing columns, say x1. And if we are on this code now, x has the expected shape, four columns, one for the bias, one for each input variable, and one line per example. We took care of x. Now we need to shape y, the labels. At some point, we'll compare these labels with the predictions so they should have the same shape. In the rules of matrix multiplication, here are they are again, they tell us that the predictions are a one-column matrix with one row per example. So we have to shape y like that, and we can do it with another NumPy method called reshape with this weird syntax. Most of NumPy's syntax is weird, not very user friendly. These numbers are the number of rows and columns that we want to shape y into, and in particular we're saying reshape y so that it has one column, and this -1 means as many rows as necessary. So this lowercase y is an array, but after these are shaping, uppercase Y is a matrix with one column and as many rows as it needs to contain all the labels from the original array. So, let's check our diagram of matrix dimensions again. We took care of x and y, but we still need to shape w, and w isn't initialized in the loading code, it's initialized inside the terrain method here together with b. By the way, we can remove all traces of b from this code because we don't use it anymore. Now we only have w, and w doesn't start its life as a single 0 anymore, it needs to be a matrix of 0s. You've already seen the ones function that initializes a matrix of 1s. Guess what? There is also a zeros function. We need to tell it the dimension of this matrix of 0s. This is one argument with two values. That's why we have another pair of parentheses in this function call, and these dimensions should be one column and as many rows as there are input variables. We can get the number of input variables from the shape of x. Let me show you. I'll create another cell. If we ask for the shape of x, we get 30 rows, the number of examples, by 4 columns, the number of input variables, and we care about the second element. So index 1, the number of input variables, then this case is 4 input variables including the bias, so this must be the number of rows in w in one column. So, this is a bit more complicated than I would have liked, but you don't need to memorize these details as long you understand what we are doing here. We are adapting the dimensions of our matrices so that we can ultimately do this, this matrix multiplication. I told you this was going to take a while, but actually we're almost done. We just have these two functions to upgrade. Let's finish the job.

Upgrading the Loss

Here are the two remaining functions that we must upgrade to cope with multiple variables. Loss, that calculates the error in the model, and gradient, that calculates the gradient of the error, for gradient descent. Loss is the easier of the two because if you look at it, it doesn't care about the dimensions of these matrices as long as those dimensions are consistent, like when it subtracts predictions and labels it wants those two matrices to have the same shapes, and we already took care of that, so bottom line, all we need to do in loss is to remove b, because b is gone. It's part of the weights now. Neat. One thing that makes me happy is that you would imagine that the code becomes more complicated when you move from one input to multiple inputs, but it actually becomes simpler. Isn't that lovely? Then we have gradient, and here as well we can remove all traces of the bias, and that's a nice simplification. And the next step is where we had the gradient with one variable, now we want the gradient with multiple variables. That gradient, like the previous gradient, comes out of a mathematical calculation. In the server it wouldn't make sense for us to go through that mathematical proof in this training. That would take quite some time and it wouldn't carry much value. We just want to get an intuitive sense of how the new gradient works. So, let me start by telling you straightaway that math textbook in hand when we go from one input variable to many input variables, this code that calculates the gradient changes like this, and maybe you can even see intuitively that these two lines basically calculate the same thing. This multiplication by 2 stays the same, and this calculation of the error is also the same, even though in the upper version it's an operation on arrays, and in the lower version it's an operation on matrices. Then this multiplication here becomes a matrix multiplication because we have matrices now, and there are a couple more details worth noting. One is that to make matrix multiplication work, you remember I told you that matrix multiplication has stricter rules about the dimensions of the matrices in involved, to make it work I had to reshape the x matrix with this T operation, where T stands for transposition. We've already seen an example of matrix transposition earlier on with weight matrix. In short, it's an operation where you flip a matrix on its diagonal, like this, so that its columns become its rows and its rows become its columns. It's just something I had to do to make all the dimensions consistent. And the very last detail, you might wonder where the average operation went. It's here in the old version of the code, but it doesn't seem to have an equivalent in the new version, and the answer is that the average is the sum for all the examples divided by the number of examples, and it turns out that in this new code the sum is already part of the matrix multiplications, so all that's left to do is to divide by the number of examples, and that's what this division here is doing. So we still have an average, even though it's harder to see. So that's the intuitive explanation of the new gradient code. It's essentially the

old code, only with matrices. We want to go through a formal proof, although you might enjoy arriving that proof yourself if you like algebra. In this training, we can just trust the textbooks and place this code right here, and voila, our multiple regression program is done and we can run it.

Running Multiple Regression

So we're ready to run this thing. To do that, we need to decide the values of these two arguments to train, the number of iterations and the learning rate. These are also called the hyperparameters of the system. This name, hyperparameters, is meant to avoid confusion because the term parameters potentially means two different things. It means the parameters of the train function, and it also means the parameters of the model, that is the weight. So to avoid that confusion, the parameters of the model are just called parameters, and the parameters of train are called hyperparameters. The values of the parameters are found by the train function. That's the purpose of the train phase. But finding the values of the hyperparameters is up to us, and it's often hard to find the best values for them that result in an accurate model. In fact, you might say that finding good values for the hyperparameters is one of the challenges of machine learning. In this training, we just want to dip our toes into machine learning, so let's just use some values, not necessarily the best ones. Let's say that we want to train the system for 100, 000 iterations, which sounds like plenty, with this small learning rate, so that gradient descent moves in tiny baby steps and hopefully that makes it precise. Time to run. You can already see the loss decreasing, as usual. Picture gradient descent, that ball rolling along the surface of the loss. This time it's a higher dimensional surface, but otherwise this is the same gradient descent algorithm that we had with regular linear regression. And in the end we have the weights, and we can use them to make prediction. For example, let's take the first line of x as inputs. It has the bias, that's always 1, then 13 reservations, 33 degrees of temperature, and so on. And in this case, the system predicts that we'll sell 51 pizzas, and the label from the original file for this day is 52 pizzas, so pretty close. And if I take, say, the sixth example, then we predict 61 pizzas while the original label is 60. Again, close. Nice. I encourage you to run this code yourself and make your own checks, but from this quick smoke test, it seems that our multiple regression program is learning from the data and making decent predictions. We had to make a few changes to our code to get to this point, but those changes made our system so much more powerful. Now it could actually be useful for more than just toy examples. Cool stuff. Let's recap this module. We extended our linear regression program, we made the jump from simple examples with one input variable to examples with multiple input variables. To get there, we changed our code to use matrices essentially everywhere. We loaded our data into matrices and we used the matrix weights to represent the

model's parameters. In particular, we used the matrix multiplication to implement our model and predict the pizzas from the input variables. So, we had to give very specific shapes to our matrices so that we could multiply them, subtract them, and so on. And by the way, by switching to matrices, we could get rid of the bias parameter and just have a nicely consistent matrix weight as our model's parameters. Okay, it's been a long module. Congratulations for getting over it, but now we are in a very good place. Believe it or not, we are edging close to a computer vision program. We need to take one last big step to get there. We'll do that in the next module.

Predicting Discrete Outcomes

Getting to Know Classification

Hello again. We are well over the half point of this training now, so you might wonder where is my computer vision program? It still looks like an impossibly ambitious goal. However, in this module we are going to take a big leap towards that goal. We'll overcome the gap from our current multiple regression program to a program that identifies images. Let me show you what that gap is. Imagine a hypothetical program that recognizes images. For example, it looks at pictures of a marsupial and predicts whether the picture contains a platypus, an echidna or neither. That would be three possible labels. And we'd probably encode each label with a number. Say, 0 for platypus, 1 for echidna, and 2 for neither. That's one possible encoding. This program is called a classifier because it takes a piece of data, an image in this case, and it tells me which class this piece of data belongs to where class means category, one of these three results. So far we used a technique called regression to predict pizza sales, and there is a crucial difference between classification and regression. That is, a regression makes a numerical prediction. It can output any number. Okay, in the case of pizza sales we expect a positive number and we would also approximate it to the nearest integer because maybe we don't want to sell fractions of pizza, but technically the regression program could output any number. By contrast, the output of the marsupial classifier is categorical, not numerical. It cannot just output the number, it has to output one of these three classes, 0, 1 or 2. I know that doesn't sound like a big deal, but this distinction between numerical and categorical does change things. For example, look back at the pizza data we had and let's say that instead of predicting the number of pizzas sold, we want to predict something categorical, like whether our pizzeria is going to break even to turn a profit on a given

night. Break-even is categorical because it can only have a limited number of values, in particular two. Either we break even or we don't. It's binary. For example, on this day the restaurant did break even and on this other day it didn't. So, what we want is a binary classifier that predicts either 0 or 1. Now let's put this data on a chart like we did for the pizza data. I cannot easily visualize data in four dimensions, but just for the sake of explanation let's ignore these two columns and just plot the first column, Reservations, and the Break-even label. And if we plot those two columns, we get this chart. And now you can see why we cannot just apply linear regression to categorical data. Remember what linear regression does. It tries to approximate these points with a line, but how are we going to draw that line on these data points? I mean, linear regression assumes that the points are roughly aligned to begin with, but in this case they definitely are not aligned so we cannot approximate them well with a line. And we have the same problem in the case of higher-dimensional data. If the data is categorical then we cannot approximate it with a straight linear shape, like a plane or a hyperplane. So, that's the mismatch between linear regression and categorical data. We cannot get a good approximation of categorical data by tracing a linear shape. So, what can we do instead?

From Regression to Classification

To recap, the problem we want to solve is multiple linear regression and categorical data don't mix. I'll show you how to solve this mismatch using something called a sigmoid. In this module, we'll solve this problem for binary categorical data like this Break-even category. In the next modules we'll see how to classify data with more than two categories. So, here is an approach to fix the mismatch. The problem is that our linear model outputs any number, not just 0 or 1. One thing we can do, you know what they say about programming that you can solve any problem by adding a level of indirection? Well, we can do something similar, only applied to mathematics. We can solve this problem by taking the output of linear regression and passing it through another function, and this function would squash that number so that it only spans values between 0 and 1, and that would give us a model to approximate something that can only be 0 or 1. So to recap, this function takes any number in input and it outputs something in this range between 0 and 1. Now you might be tempted to say okay, so, let's use a function like this, a step function. Everything under a certain value, say 0, becomes 0 and everything over that value becomes 1. But there is a problem with that. Maybe you remember when we talked about gradient descent and we said that gradient descent doesn't work well on functions that have sudden jumps or flat areas, and this step function has both. So if we put a step in our model it will eventually become part of the loss calculation because the loss depends on the model. And because of that, the loss

will also have jumps and flat areas, and that doesn't work well with gradient descent. Instead, gradient descent likes functions that are smooth and gently sloped, so something more like this. This is more like it. A function that goes from 0 to 1 smoothly and it never reaches either extreme. It never goes completely flat. It's always gently sloped. And if we check the textbooks, there is a nice function that looks exactly like this. It's called the logistic function. It belongs to a family of functions called the sigmoids, so most of the time it's just called the sigmoid for short. And here is its formula. It's not a hard formula to implement. With NumPy it's a one-line implementation, like this. Okay, time to update our program to use the sigmoid. Here is the regression code again, and I also copied the sigmoid here, so let's make use of this function here where our model is. Take the output of this matrix multiplication and wrap the sigmoid around it. So now predict will output something that ranges from 0 to 1. And you might think that we are done, that we need nothing else to turn regression into classification. However, there is one last wrinkle that we need to take care of.

Introducing the Log Loss

Let me cut straight to the chase. Now that we introduced the sigmoid, our model doesn't work well anymore with the way that we calculate the loss, the mean squared error formula. We need a new formula for the loss. Let me show you why because that's not obvious at all. I made an experiment. I generated some random data with two input variables and I plotted the mean squared error loss for it using the model we just created, this model that includes the sigmoid just to get an idea of what the loss surface looks like when we use the sigmoid and the mean squared error together. And the result looks like this. And honestly, it's pretty nasty. You can see that this wouldn't be a good surface for gradient descent. I mean, look at those creeks running along the surface. If gradient descent gets into one of these it will probably get stuck in a local minimum. And this surface also includes areas such as this one that are flat or maybe nearly flat. So their gradient would be close to 0 and gradient descent will be very slow around here. So long story short, we introduced the sigmoid, we tried to make it smooth and nice, but once we calculate its loss we still end up with a surface that's not gradient descent friendly. So, we need another way to calculate the loss, one that gives us a smooth, gentle surface. And as usual we certainly don't need to invent a formula ourselves because researchers tackled this problem already and they came up with multiple ways to calculate the loss that mix well with different models. For example, for classification models there is this alternate implementation of the loss that works pretty well. It's called the log loss where log is short for logarithmic. And if we use this code to calculate the loss instead of the mean squared error, this uneven surface turns into this. Look how nice and

smooth it is. It makes you want to roll in it and smoothly descend its gradient to the minimum point. I'm getting all lyrical here. The log loss code looks long and complicated at first sight, but actually if you give it a try you'll see that it's actually simpler than it looks. In particular, remember that each label Y is either 0 or 1 because this is a binary classifier that we are building. And if Y is 0, then this term is multiplied by 0 and disappears. And if Y is 1, then this term that is multiplied by 1 - Y, that is again 0, it disappears. So actually, the log loss is just one of these two terms for each example. And the last line takes the average overall examples. Just saying. We don't really need to know, but it's nice to be aware that this code is simpler than it looks at first sight. The important thing to know is this is a good loss. That is, just like the mean squared error loss, it's a measure of the error in our model. The better the model approximates the examples, the lower the result of this function. And by descending its gradient, we can find the minimum loss, that is the set of parameters that result in the best model that best approximates the examples. Now, since we just changed from the mean squared error loss to the log loss, we also need to change its gradient because this is still the gradient of the mean squared error loss. And there is more good news here because even if the log loss looks so different from the mean squared error loss, it turns out that the formula for calculating its gradient is almost the same. The textbooks tell us that the only difference is that in the log loss we don't have this multiplication by 2. Otherwise it's the same gradient formula. So, it's quite easy to update this code. And as usual, if you like math and you know calculus, you can calculate this gradient on your own, but we won't do it here. We'll just trust the textbooks. And now that we updated loss and gradient to switch from mean squared error loss to the log loss, we are done. We can run our system.

Testing the Classifier

We made quite a few changes to this code, so let me recap them before we run the code so that we know exactly what we're running. Our goal in this module was we wanted to turn our multiple regression program into a binary classifier, a program that predicts binary labels. To do that, we wrapped a sigmoid around our linear regression output and the sigmoid guarantees that the model's predictions are always in the range from 0 to 1. The sigmoid is a nice, smooth function so we expected that it's good for gradient descent, and it is except that it doesn't work well with the mean squared error. It generates a loss surface that's uneven, it has areas that are mostly flat, creeks, and other unpleasant features. So we switched to a different loss formula. The log loss. The log loss is different from the mean squared error, but its gradient formula turns out to be similar, so we updated both the loss and the gradient. And that's what we did. Now let's check how it worked out. I'll run all these cells and train the classifier on our pizzeria's break-even

dataset. As usual, we can see the loss decreasing. It's calculated in a different way, but it still means the same thing. It's the error of the model and the gradient descent is searching for the parameters that minimize this error. And at the end we get this bunch of weights. Okay, let's see how good these weights are. Let's predict the labels for all the examples in our dataset. There you go. We got a bunch of predictions all between 0 and 1. You could read these numbers as how certain the system is that the pizzeria will break even. For example, on this day the value is really close to 1. That means yep, we are almost certainly going to break even. And on this other day, we have nearly 0 chances of breaking even according to the system. And on this day, for example, the system is pretty uncertain whether we will break even or not. It's slightly under 0.5, so it tends to say no, but it's really on the fence on this. If we don't like these "maybe" predictions and we want more assertive "yes" or "no" predictions, then we can use NumPy to round these numbers to either 0 or 1, like this. And we get stark 0 or 1 predictions, without all the in-betweens. Here is one interesting thing that we can do. If we do this, it means tell me whether these predictions are the same as the labels. That is, show me true if the prediction is the same as the label and false if it isn't. It's a nice, quick way to gauge how accurate these predictions are. And if I run it, wow! This is good actually. The system has learned to classify all the examples in the dataset correctly except for this one. So, it's not 100% accurate but still very accurate. That's quite impressive for a few lines of code. So, we have a binary classifier. And now, believe it or not, we can use this binary classifier to do computer vision straightaway without any changes to this code. I'll be waiting for you in the next module. This is going to be exciting.

Recognizing Individual Digits

Introducing MNIST

So, image recognition. I told you we were going to do it, and the moment has come. We're about to get familiar with a dataset of images, and then in this module we're going to run our classifier on some specific images of that dataset. And in the next module we'll run it on all of them. And, here is that dataset. It's called MNIST, that stands for Modified NIST. The NIST is the National Institute of Standards and Technology that originally collected this data. MNIST is the classic image recognition dataset. People used it for many years as a benchmark. And it's a relatively simple dataset, but still a great starting point when you get into machine learning. MNIST contains

images of handwritten digits like this one. Each digit is a 28 x 28-pixel grayscale image where each pixel is a single byte. Zero for perfect white, 255 for perfect black, and everything in between. You might expect 0 to represent black, but it's the opposite here. In MNIST, higher byte values are darker. And each image carries a label that is the value of the digit in the image, 4 in this case. MNIST contains 70, 000 images, 7, 000 for each digit; 7, 000 zeros, 7, 000 ones, and so on, up to nine. And as you can see, these images are the real deal, not artificial examples. Somebody collected these digits from the wild, scanned them, cleaned and centered them and so on, but they're still authentic handwritten digits and sometimes pretty hard to recognize. So, if our classifier does well on this dataset, then we win bragging rights. That would be a reason to celebrate. To download MNIST, I tried to make your life easier, and in the Jupyter Notebook for this module I wrote a cell that will go and get it from the official site. If you run this cell and then wait for a while, let me speed up time here, you should get an mnist directory in the same directory as the notebook, and this directory contains the dataset. Four binary files. Two contain the images and the labels for the training set, the data that we should use to train the system. The training set includes 60, 000 examples. And the other two files contain the remaining 10, 000 images and labels. That's the test set for checking how accurate the system is after training it. Now you might wonder, why set aside 10, 000 precious examples for testing alone? That feels like a waste. Instead, we could use them to train the system a bit more and maybe make it a bit more accurate. And then we could test the system on the same data that we used for training it. That's an important doubt, and it deserves a brief aside, let me clarify it.

Understanding the Test Set

So, MNIST comes in two sets, a training set and a separate test set, and so do most other popular datasets. And when they don't, you're expected to split them into a training in a test set yourself, but why? Why do we need a test set in the first place? Can't we just use the same examples for training the system first and testing it later? The reason for the test set is that you cannot really know how accurate your system is until you test it on new data, data that it hasn't seen before during training. As a counterexample, I wrote this fake MNIST classifier. It's pseudocode, not Python, just to prove my point. And what this system does is during training it saves all the images and the labels in a dictionary, a map of MNIST images to their label. So, once this system has seen an image once, it will always remember its label. And during prediction, it just retrieves the labels for the given images. Boom, 100% accuracy in just one iteration of training. But of course that's cheating, right? This system didn't really learn anything. If you ask it to predict a label for an image that it hasn't seen before during training, even slightly different from the

images it's seen during training, then it has no idea how to classify that. It's just like a student who memorizes the textbooks without understanding them. And as soon as he gets an exercise that's not in the textbook, this student gets completely lost. And that's my point. We want our system to generalize from the training data, to understand the data so to say, to understand what makes a 5 a 5 and how that's different from a 6. We don't want it to memorize the data, to just store a bunch of specific 5s and 6s. But as it turns out, all powerful machine learning systems, even if they're not cheating, have a tendency to memorize the training data. That's a problem known as overfitting. You say that a system is overfitting the training data when it memorizes it instead of generalizing from it. And that's a serious issue because then you get the illusion that your system is really accurate until you run it on new data, and then suddenly it's not very accurate at all. Overfitting is one of the hard problems with machine learning in general, and avoiding overfitting is an art in itself and outside the scope of this training, but there is one basic rule that goes a long way towards at least making overfitting more visible and less of a sneaky problem. That is, never test a machine learning system with the same data that you used to train it. If you use new data, then at least you'll be sure that you're getting a realistic metric. Maybe the system is still overfitting the training data, but at least that overfitting won't give you a delusion of accuracy. You'll get a realistic measure of how accurate the system is on new unknown data. And that is why MNIST comes with a separate test set. Okay, thank you for sticking with me through this short aside. Let's get back on track and see how to interpret MNIST with our classifier code.

Preparing the Images

So, we have the MNIST dataset and now we need to load that data and run our classifier on it. But it might not be clear how to do that because the examples in MNIST are images encoded in their own binary format, and our classifier wants something else in input. It wants a matrix where each row is an example and each column needs an input variable in the example. How can we go from here to there? And the answer might surprise you a little. What we're going to do is to take each image in the dataset and squash it to a long row of pixels, and each pixel becomes an input variable. A byte in a row of the input matrix. An image is 28 x 28 pixels, 784 pixels in all, so we have 784 columns. This is a 784-dimensional problem, which might look like a lot, but hey, we did regression with 3 variables, we can do it with 784, right? Actually sorry, 785, because remember we need another column full of 1s for the bias. So, X will be a big old matrix with 785 columns and 60, 000 rows, one per example. One reason why you might be surprised of this method is you might wonder, aren't we destroying the images this way? Don't we lose all geometric information when we flatten them like that? And yeah, we do lose a lot of information in the images when we

jump on them and squash them into these flat rows of bytes, but we still hope that they contain enough information for the system to recognize them. For example, if we have a 0, then we expect that the central pixels in a 0 are mostly white while the central pixels in 1 on average tend to be more black. So, we're betting that this distribution of colors over the pixels is enough information to recognize them. We are trusting the power of statistics if you wish. There are also more sophisticated machine learning systems that do computer vision by looking at the geometric information in an image, but the system we're writing is not one of them. So to recap, we need to load the images in MNIST and squash them into the rows of a big matrix and add a bias column to that matrix. Both the images and the labels are stored in binary files and the exact format is described on the official MNIST site. I checked that documentation and wrote this function to load the images into this big matrix. I won't go into many details because this code doesn't really matter to us in this training, it's just specific code to load this specific dataset. Anyway, I added comments in case you want to understand all of it. In short, this function takes the name of an MNIST image file, either the training images or the test images, and it unzips this file, parses its binary content, and it reshapes it to a matrix with one line per image and one column per pixel, plus a bias column in the beginning. And if we print one of these two matrices, here it is. It's a large matrix, so NumPy is only showing a part of it. Here is the bias column and these pixels all look like 0 because we're on the edges of the image and all pixels are white there. And here are the dimensions of the training matrix, 60, 000 rows, like the number of images, and 785 columns for the 784 pixels plus the bias, while the test matrix has 10, 000 rows and the same number of columns. And that's it for the images. Now, let's load the labels.

Preparing the Labels

The labels in MNIST are a sequence of numbers, each representing a digit, 0 to 9. While our classifier wants the labels as a one-column matrix, so we should load the labels from MNIST and reshape them so that they're a matrix with one column and as many rows as the number of labels, but there is also something else that we need to take care of. Remember, our classifier is a binary classifier. It can only deal with labels that are either 0 or 1. So now we have a problem here. These labels are not binary, they range from 0 to 9. So, it looks like our classifier cannot deal with them. There is a way around that problem, but we'll discuss it in the next module. For now I know that you're eager to see the classifier running, so let's do something simple. Instead of recognizing all the digits in MNIST, let's start easy and recognize only one of them, say the digit 4. So in other words, we want to use a binary classifier to recognize if a digit is a 4 or any other digit, and that is a binary problem. Again this is only temporary, and in the next module we'll recognize all the

digits. But by only recognizing 4s, we can run our classifier right now. So, we need to change the labels so that every label that is 4 becomes a 1 and all the other labels become 0. I put these ideas in code and here is a function that loads the labels in the format we've just seen. Again, you don't need to bother with the details, although if you wish to do that, that's what these comments are for. The code is similar to the one that loads the images, and in the end it reshapes the labels into a single-column matrix. And here it goes through some NumPy voodoo. This line turns all the labels 4 into 1s, and all the other labels into 0s. NumPy's syntax is kind of obscure as usual, but I have to say it's quite powerful. In this case we're saying, create a matrix that contains True whenever the label is a 4 and False otherwise, and then convert that matrix to a matrix of integers so that True becomes 1 and False becomes 0. That's tricky code, but it's also very terse. But again, what matters for us here is the result. And the result is we can call this function with the names of the label files in MNIST and get two matrices, like this. Let's check the training labels. One column and it contains a 1 where the digit is a 4 and a 0 for all other digits. And it's 60, 000 rows, one per label. And the matrix of test labels is the same only with 10, 000 rows. So we're done. We have our images and labels in the right shape and the right format for our classifier. Let's run the classifier on this data.

Recognizing a Digit

I copied our classifier code into this notebook, and now we have everything we need. We have the code that loads the data and the code for the classifier. Let's run training. We need to decide the hyperparameters for the training, the number of iterations, and the learning rate. I made a few experiments here and I found that 200 iterations seem to be okay. We could do more, but this dataset is bigger than the toy examples we had so far, so iterations tend to be slower and we don't want to wait forever. Actually, let me run this now so that it goes through the training as we speak. There, we can see the loss decreasing. And again, about the hyperparameters, I also found that we need a pretty small learning rate, and if the learning rate is too large the result is that gradient descent takes steps that are too big and the loss ends up increasing instead of decreasing at every step. In fact, if you look close you'll notice that this is still happening during the very first iteration. The loss is increasing a bit here, which means that gradient descent took too large a step downhill and it ended on a higher cliff, a higher loss. That's usually a sign that we should make our learning rate even smaller, but then the system would take smaller steps and we'd have to train for longer. So, it's a balancing act. I encourage you to experiment on your own with different values of the hyperparameters. Here I'm sticking with these values because I see that after this temporary increase the loss starts decreasing again, so I guess we can live with this

first gradient descent misstep. So see, tuning hyperparameters is hard. It's more art than science sometimes. So the training is done and the loss kept decreasing for 200 iterations. So now we have a set of 785 weights, one per input column. And we can use them to make a prediction on the test set. Remember, we don't want to check this on the training set, that wouldn't mean much. We want to check it on new characters that the system hasn't seen yet. And these are the predictions. Let's make them more readable by rounding them to 0 or 1. And it's a long array of 10, 000 predictions, so let's take just a few of them. For example, I don't know, 10 elements. This is the NumPy syntax to get a slice of elements from an array. So the system predicts that there are 2 fours amongst these 10 elements. One in this position, and one here. And the rest are digits that aren't 4. And if we check the encoded labels, they say that wow, actually it nailed them. It exactly recognized the 2 four digits in the bunch. Okay, this might be a lucky one. There is no guarantee that the system is this accurate over the entire test set. Also, maybe 4 is a particularly easy character to recognize and it's not quite as easy to tell 5s from 6s, for example. So, I'll leave you to your own experiments to verify how accurate the system is on other examples and other digits, but what I'm really eager to do here is cover the last mile of this character recognition thing and recognize not just one, but all the digits from 0 to 9, and then measure the accuracy of the system on the test set. And that's what we'll do in the next module, so see you there. But first, a quick recap. In this module, we introduced MNIST, a standard image recognition benchmark. We explained why MNIST and machine learning datasets in general should be split into a training set and a test set. And then we encoded the MNIST images and labels for our classifier. We squashed the images into rows of bytes and we encoded the labels to either 0 or 1, and we checked that our classifier seems to do a decent job at identifying the digits in the dataset that represent a 4. And now, let's take that last step and recognize all the other characters. I can't wait to measure the result of this.

Figuring Out Image Recognition

Planning for Multiple Classes

Okay, we are this close to building a system that recognizes handwritten digits from 0 to 9. We need one last step, and we'll take it in this module. Here is where we are now. We have a classifier that recognizes one digit. In the previous module, we used it to recognize the digit 4. And it's a

binary classifier because it returns 0 or 1 for no, this is not a 4, or yes, this is a 4. To be precise, it doesn't literally return 0 or 1. It returns a number between 0 and 1, and the closest that number is to 1 the more confident the classifier is that it's looking at a digit 4. Now we want to extend this idea to a multiclass classifier to recognize any digit from 0 to 9. How do we get from here to there? Here is a way. To recognize any digit, we could have one binary classifier for each digit. One for the digit 0, let's call it the 0-classifier, a 1-classifier and so on until the 9-classifier. And if we have a digit, a 7 in this case, and we pass it to all these classifiers, then we'll get 10 results. So for example the 7-classifier might return 0.9, that means I'm pretty confident that this is a 7, while the other classifiers would hopefully return values closer to 0. The 1-classifier, for example, returned 0.4, which means this is probably not a 1. And some of the classifiers might get it wrong. So maybe the 4-classifier returned 0.8, meaning I am fairly confident that this is a 4, and that's wrong. But what we can do is just trust the classifier with the maximum output, the most confident. In this case, nobody is quite as assertive as the 7-classifier. That's the highest number we got. So, let's predict that this digit is a 7. We let them vote, so to speak, and we pick the most confident. So, this is how we can build multiclass classification from multiple binary classifiers. Now, if we literally built 10 binary classifiers and trained them, that would take a long time, but the good news is we can optimize this process by merging these 10 classifiers together and using matrix operations to train them all together. Let's see how to do that.

Encoding the Labels

To go from 1 binary classifier to 10 binary classifiers in parallel, let's start from the labels, Y. The labels in MNIST are numbers from 0 to 9, but in the previous module we encoded them to contain a 1 where the original label was a 4, and a 0 otherwise. Now we can do the same thing for each of the 10 digits by having 10 columns. So, Y isn't a single-column matrix anymore. Now it has 10 columns that each encode a specific digit. This is the column we had before that encodes the digit 4. It's one when the label is 4 and 0, otherwise, like before. But now we also have a column that encodes the digit 0. It contains 1 when the digit is 0 and 0 otherwise. And a column for the digit 1, and so on. This method to encode the labels is called one hot encoding because if you look at the rows of this matrix, only one element per row is a hot 1, the others are cold 0s, I guess. So, conceptually it's as if we had 10 binary classifiers, but in practice those classifiers are columns in this matrix, and each column recognizes one digit. Let's translate this concept to code. This is the code that loads MNIST. This is the function that loads the images. We don't need to change this one. The function that we need to change is this one that loads the labels. In particular, this last line that encodes the digit 4 isn't relevant anymore, I'll delete it. And now we should one hot

encode all the digits. Actually, I don't want this function to do too many things, so I'll just return the labels as they are here and write a separate function to one hot encode the labels. It took me a few minutes to write this function, so I'll skip forward to show you the finished code. Here it is. What it does is it takes the MNIST labels, that's a one-column matrix, and prepares a matrix of 0s with as many rows as the original labels and 10 columns, one per digit. And then it goes through the rows of this matrix, and for each row it takes the element in that column that matches the original label and flips it from 0 to 1. So if the label is, say, 0, then it flips the element of index 0, the first element in the row. If it's 1, then it flips the second element, and so on. And that's one hot encoding. Let's see whether it works as we expect. I'll load the original labels first and then encode them. Notice that I one hot encoded the training labels, but not the test labels. I'll show you why in a few minutes. And now let's look at the first 10 rows in the original labels. This is the NumPy syntax to index the first 10 rows. Here they are. And the first 10 one hot encoded labels look like this. And you can see that the first label here is a 5, and in this matching row the element with index 5 is 1 and the rest are 0s. And the second label is 0, so only the first element is 1, and so on. Of course, changing the labels like this has consequences. Let's update the rest of the system to deal with this new shape of Y.

Updating the Weights

So we changed the shape of the labels matrix, right? Now we need to take care of the other matrices in the system so that their dimensions are all consistent, and as usual the dimensions of matrices can be a bit complicated to deal with, but in this case it won't take long. Let me show you. The key matrix operation in this code is this multiplication here. We multiply the inputs by the weights, and we also pass the result through a sigmoid, but the sigmoid doesn't change the dimensions of the matrix, it just applies to each cell so we can ignore it. As far as matrix dimensions are concerned, we need to make this operation consistent. The inputs by the weights equal the predicted labels. But now the labels aren't a single column anymore, they're a matrix with 10 columns. It's like we have 10 binary classifiers, right? And each of them needs its own weights. So in other words to get 10 columns in the labels, we need 10 columns in the weights. So w is now a matrix of 10 columns, one for each digit from 0 to 9. And it still has the same number of rows as before. If you remember, the rows of w were the number of input variables, the same as the columns in X. And if you check out the rules of matrix multiplication, this is a legitimate multiplication dimensions-wise. So, let's go straight to the train function where the weights are initialized and change w to have these dimensions, as many rows as before, but 10 columns, not 1. Actually, let's keep it generic instead of hard coding it, it's nicer this way. W has as many columns

as the columns in Y, one per digit, and as many rows as the columns in X, the input variables. The number of pixels in an MNIST image plus the bias. And now the dimensions of the matrices are consistent. So, almost without noticing, now we have a matrix of thousands of weights. Remember the first version of this system that used a one-dimensional linear regression model with two parameters, a weight and a bias, and now just like that we have close to 10, 000 parameters. You see how quickly this model has grown. But the idea of this system stayed the same. The train phase is still going to tweak those parameters to try and approximate the dataset, only now there are many more parameters. Okay, we're almost there. There is one last thing that we have to fix.

Updating Prediction

One thing that we still need to do is update the predict function. We changed the dimensions of these matrices to run the equivalent of 10 binary classifiers in parallel. So now if we ask this operation to make a bunch of predictions, it doesn't return one number per prediction anymore. Now for each prediction it's going to return 10 numbers, one per classifier, like this. A column for the 0-classifier, one for the 1-classifier, and so on, up to 9. But we don't want 10 numbers, we'd just like to have one clear prediction like I think that this digit is a 7. So, we need to decode these numbers. And I told you earlier that the way we decode them is to take the classifier that's more confident that returns the highest number. So, let's write the code to do that, but I wouldn't change this predict function because it's also used to calculate the loss and the gradient of the loss, so let's leave it alone and write another function instead. Let's call it classify, that calls predict, and then decodes it to come up with the final classification. So, predict comes up with a matrix with 10 columns and classify decodes that matrix into regular labels from 0 to 9. These names are a tad confusing maybe. If we had production code here, then we'd probably look for better names. But for such a small piece of code, I think it will be fine. So, what does classify do? For each row, it wants the digit that's associated with the highest confidence. And we can find it with a NumPy operation called argmax. That means, give me the index of the cell with the highest result. We have to say axis=1 to mean apply argmax to rows, not columns. Another case of obscure NumPy syntax. So in the case of this row, for example, argmax would return 2 because that's the index of the maximum element. And it's also the digit associated with this column. And one more thing. We want to have these labels in the same shape as the original labels, a single column and one row per label. So, let's use the reshape operation. You might remember it from an earlier module. We want to reshape this matrix so that it has as many rows as it needs, that's what -1 means, and one column. So, this code is a bit hard to read, but NumPy documentation at hand,

what it does is it decodes a matrix of predictions with 10 columns into a single column of labels, each from 0 to 9. And by the way, this also explains why we didn't one hot encode the test labels, remember? That's because in the end we want test labels and predicted labels in the same human-readable format so that we can compare them. So we don't want to one hot encode them. One hot encoding is something that we only use during training. And with that, we're done. Now we can run our MNIST classifier. How well will it recognize characters? We have a cliffhanger here.

Running the Final Test

Okay, let's start training with the same hyperparameters that we used in the previous module. It will take a couple of minutes to go through 200 iterations on my laptop, so let me speed up time. And at the end, we have a matrix of weights, and we can use them to classify the 10, 000 MNIST test examples and get 10, 000 predicted labels. How many of these classifications are correct? Let me write some code to find out. Let's put the classifications in a variable. And this code means give me True whenever the classification is the same as the original label, and False otherwise. Then we can use this NumPy function to count the number of True results because this function counts False as 0. And finally, I'll get the ratio between the correct classifications and the total number of examples, that is the number of labels multiplied by 100. That's the percentage of matches, of digits that have been classified correctly. And if we print it, wow! We're over 90% correctly classified handwritten digits. I don't know what you think, but I think that's pretty awesome. This small machine that we built can classify something as messy as handwritten digits with this kind of accuracy. That's very cool, especially because, let me show you, I took all the classifier's code and copied it into one Jupyter Notebook cell just to make the point that if we really try to make this code shorter and remove all the stuff that's not necessary, then it becomes extremely short. So for example, we don't actually need to print the loss. We don't need the loss at all actually, that's just for logging. What we need is the gradient of the loss, and we can also inline the gradient code. And we can inline the predict function here and here and all these temporary variables. And we end up with these few lines of Python, 13 lines, whitespace included. And there is nothing magic and hidden in them. I obfuscated the code a little by making it shorter, but after this training you understand everything in these lines. The gradient descent, the tuning of the parameters to approximate the dataset, you know it all. And these 13 lines of code train a model to recognize handwritten characters with over 90% accuracy. It's quite amazing, actually. I just want to close this module here on a high note. I'll spare you the recap because in

the next module we'll recap the entire training anyway and we'll also have a higher-level look at the system we built.

Seeing the Big Picture

From the Basics to the Perceptron

We covered a lot of ground in this training. Let's look at what we accomplished and how it fits into the larger history of machine learning. In the first module, we introduced supervised learning, one of the most important types of machine learning. Supervised learning is all about approximating labeled examples with a model function, that's the training phase, and then using the function on unlabeled data to predict its label, the prediction phase. In the following model, we saw a concrete case of this approximation process called the linear regression where you approximate data with a line. A line has two parameters, a weight and a bias. We built a simple training algorithm to move this line around trying to approximate the examples, and we used this system to predict pizza sales. In the next module, we found a better algorithm to move the line, gradient descent. Thanks to gradient descent, we could be faster and more precise at approximating the examples, and more importantly, this algorithm paved the way to more complicated models that were the subject of the next module when we tackled multiple linear regression, linear regression with multiple input variables that required one weight for each input plus the bias. And we combined the inputs with the weights through matrix multiplication so we could predict pizza sales with multiple variables. That was a much more interesting and realistic problem. Next, we switched from regression that predicts a numerical quantity to classification that predicts a categorical quantity. In particular, we did binary classification. We added a sigmoid to our system and predicted whether our restaurant would break even on a given day. And in the following module, we applied this binary classifier to computer vision on the MNIST dataset of handwritten digits. We squashed the images from MNIST into rows of pixels that were our input variables, and we used the classifier to identify one of the digits. And finally, we learned how to do that for all the digits in MNIST. We one hot encoded the labels so, conceptually that was like having one binary classifier per digit and taking the prediction of the classifier that gives us the most confident positive result. And this algorithm got us our final outcome in this training that was a pretty accurate MNIST classifier, over 90% accurate, and that's what we ended up with. A surprising little Python program that takes a bunch of inputs like the pixels in an image, it

multiplies those inputs with a matrix of weights, passes the result through a sigmoid, and gets back a bunch of binary outputs. That's all it does basically, and this machine is called a Perceptron. It's a pretty old idea, actually, dating back to the 1950s. To be precise, the original Perceptron is even simpler than this in a few ways, the most important of which is the original Perceptron didn't use gradient descent, it used a simpler algorithm to train the model. But let's say that this is a more modern, more powerful version of a Perceptron. So, we built a Perceptron and we got pretty impressive results out of it, but it's also important that we understand the limitations of this thing. So, let me explain what Perceptrons are not good for.

Understanding the Perceptron's Limitations

Let's see what the limitations of the Perceptron are. Imagine that we are a Hollywood studio and we want to predict whether our next movie will be successful at the box office based on two factors, the production budget and the marketing budget. There are hopefully other factors that impact the success of a movie, but in this example it's all about the money. And let's draw a different kind of chart than the ones we had so far with the input variables on the two axes, and the training examples are represented by these shapes, blue squares for successful movies and green triangles for flops. And now, imagine training a Perceptron to do binary classification on this data to predict whether a movie is likely to be a success based on these variables. See what happens if we train a Perceptron on this data and then we color each pixel in this chart blue or green, depending on whether the Perceptron classifies that position on the chart as a success or a flop. We get something like this, two areas for the two classes separated by a nice clean line. This line is also called the decision boundary of the Perceptron. Movies that fall on one side of the boundary are classified as successes, and movies on the other side are classified as flops, which is nice except that this dataset is fake, as you probably expect. It's not realistic. In real life, things wouldn't be nearly as clean cut as this with two neatly separated clusters of data. It's more likely that things would be something more like this, with some distinction between the two classes of movies, but nowhere as straight and clear as the one we had before. And that's where Perceptrons get into trouble. If you train a Perceptron on this more realistic dataset, then you get a decision boundary like this. And you can see that the Perceptron did its best to separate most triangles on one side of the boundary from most squares on the other side, but it still ended up with misclassified points because at their core, Perceptrons are based on linear regression, right? We've seen it in this training. And, being based on linear regression, they can only trace a straight decision boundary, a line or a plane or a hyperplane depending on the number of dimensions, but still a straight linear shape while most interesting data aren't linearly separable. That is to say, they

cannot be neatly partitioned with a linear shape, they need some kind of curve instead. So to recap, Perceptrons work great on linearly separable data, but not as well on non-linearly separable data. And you might say, but Paolo, we just ran a Perceptron on MNIST that is a real dataset, most likely not linearly separable, and we still got a pretty good result. And yes, that's true, we did well, but that's just about how well we can do with a Perceptron. We could train our Perceptron on MNIST for years and maybe get an extra percent point or 2, but we'd never get more than 92 or 93%. And as impressive as that sounds, it's still not enough for a production-quality character recognition system. It still misclassifies almost 1 number in 10. That is too many errors for a real useful classifier. So that's the main limitation of Perceptrons. They don't work well on data that isn't linearly separable, that is most interesting real-world data. And there is actually a good story about the Perceptron, how it came to be, and its limitations, and this story actually had big consequences on today's computing and even our jobs as programmers. So it's a story worth telling to cap off this training.

The Story So Far

In the '50s, the first modern computers had been built and artificial intelligence didn't seem far behind. Many people believed that soon enough computers would think. There were two schools of artificial intelligence back then, and they had very different takes on how to build AI. One camp was called the symbolists. Some of them are legendary names today, like Marvin Minsky and John McCarthy, and their philosophy was we can build intelligence by programming it essentially. After all, we are thinking machines, so we can code computers to be like us. Back then there were no high-level programming languages, so you had to use machine code to write programs. So, the symbolists built their own languages, in part as a stepping stone to build intelligence. McCarthy created the LISP, one of the first high-level programming languages, in part for that reason. The other camp in this quest for AI were the connectionists, led by Frank Rosenblatt, and they had a very different approach that can be summed up as the brain is the organ of intelligence, so let's build a brain and intelligence will follow. And as a first step towards engineering a brain, they built a pretty complicated piece of hardware, the Perceptron. The hardware Perceptron looked like a big tangle of spaghetti essentially. So we had these two camps, the symbolists and the connectionists, and well, let's just say they didn't really get along. The symbolists had more street cred in the academic world while the connectionists were a bit like the underdogs. Symbolists, like Minsky, thought that the connectionist approach was flawed and wouldn't ever get anywhere. And in the '60s, Minsky wrote an entire book to analyze the Perceptron and essentially show the world that the Perceptron was limited, that it couldn't tackle non-linearly separable data. And he

implied essentially that the Perceptron was a scientific dead end, that it was too limited. The jury is up on whether Minsky intended to discredit connectionism with this book, but if it did, it worked great. In fact, connectionism fell into disrepute after the Perceptrons book, and for many years only a few researchers were left to study Perceptrons. And then Frank Rosenblatt, the inventor of the Perceptron, sadly died in a sailboat accident, and that seemed to be it for connectionists, which was a pity because then everybody focused on the symbolists' concept of programming AI and, ironically, that branch of AI ended up being a bit of a disappointment, at least this far. It kept over-promising and under-delivering. And on a side note, Minsky also mentioned in his book that while Perceptrons are limited to non-linearly separable data, there is a way around that limitation. You could take two or more Perceptrons and chain them so that the output of one feeds into the other, like this. And this multilayer Perceptron can approximate data that isn't linearly separable. It's not limited to straight decision boundaries, it can trace curved ones. Only back then, nobody knew how to train multilayer Perceptrons and people suspected that it just couldn't be done. And with so few researchers working on this stuff, it took many years to find out that it was, in fact, possible to write an algorithm to train multilayer Perceptrons. By the way, by then, multilayer Perceptrons were mostly called with another name, neural networks. And that's the story of how connectionism was discredited, but after many years it came back with a vengeance to take the world by storm with the modern neural networks and deep learning. But that's another story, and maybe good for another Pluralsight training or two. This training, How Machine Learning Works, ends here. We just scratched the surface of machine learning, but I hope that I gave you a basic idea of the technology behind the magic. If you enjoyed this training and you want all the details that couldn't realistically be crammed into an online training, I wrote an entire book on these topics. It's called Programming Machine Learning, by Paolo Perrotta. That's me. Check it out if you wish. It's the one with a hammer on the cover. And that's it. Leave a rating for this training here on Pluralsight if you wish, and drop me a message with your questions if you have any questions or just your feedback. I love feedback. I hope you had fun following this Pluralsight training because I sure had fun producing it. Thank you very much!

Course author

Paolo Perrotta

Paolo Perrotta is the author of "Programming Machine Learning" and "Metaprogramming Ruby". He has hundreds of articles, conference speeches and training deliveries under his belt. He developed...

Course info

Level	Beginner
Rating	★★★★★ (55)
My rating	★★★★★
Duration	2h 23m
Released	15 Nov 2019

Share course