1 Intro to Lesson 2

In the last lesson we've trained a simple logistic classifier on images. Now, we're going to take this classifier and turn it into a deep network. And it's going to be just a few lines of code, so make sure you understand well what was going on in the previous model. In the second part you are going to take a small peak into how our optimizer does all the hard work for you computing gradients for arbitrary functions. And then we are going to look together at the important topic of regularization, which will enable us to train much, much larger models.

2 Number of Parameters Quiz

The simple model that you've trained so far is nice, but it's also relatively limited. Here is a question for you. How many train parameters did it actually have? As a reminder, each input was a 28 by 28 image, and the output was 10 classes. Enter the number of parameters here.

3 Number of Parameters Solution

The matrix, W, here, takes, as an input, the entire image, so 28x28 pixels. The output is of size 10, so the other dimension of the matrix is 10. The biases are just 1x10. So the total number of parameters are 28x28x10, plus another ten, that gives you 7,850.

4 Linear Models are Limited

That's the case in general. If you have N inputs, and K outputs, you have (N+1)K parameters to use. Not one more. The thing is, you might want to use many, many more parameters in practice. Also, it's linear. This means that the kind of interactions that you're capable of representing with that model is somewhat limited. For example, if two inputs interact in an additive way, your model can represent them well as a matrix multiply. But if two inputs interact in the way that the outcome depends on the product of the two for example, you won't be able to model that efficiently with a linear model. Linear operations are really nice though. Big matrix multiplies are exactly what GPUs were designed for. They're relatively cheap and very, very fast. Numerically linear operations are very stable. We can show mathematically that small changes in the input can never yield big changes in the output. The derivates are very nice too. The derivative of a linear function is constant. You can't get more stable numerically than a constant. So, we would like to keep our parameters inside big linear functions, but we would also want the entire model to be nonlinear. We can't just keep multiplying our inputs by linear functions, because that's just equivalent to one big linear function. So, we're going to have to introduce non-linearities.

5 Rectified Linear Units Quiz

Let me introduce you to the lazy engineer's favorite non-linear function: the rectified linear units, or RELU for short. RELUs are literally the simplest non-linear functions you can think of. They're linear if x is greater than 0, and they're the 0 everywhere else. RELUs have nice derivatives, as well. Which of these plots do you think best represents the derivative of a RELU?

6 Rectified Linear Units Solution

When x is less than zero, the value is 0. So, the derivative is 0 as well. When x is greater than 0, the value is equal to x. So, the derivative is equal to 1. The second answer is the right answer.

7 Network of ReLUs

Because we're lazy engineers, we're going to take something that works, a logistic classifier and do the minimal amount of change to make it non-linear. We're going to construct our new function in the simplest way that we can think of. Instead of having a single matrix multiplier as our classifier, we're going to insert a RELU right in the middle. We now have two matrices. One going from the inputs to the RELUs, and another one connecting the RELUs to the classifier. We've solved two of our problems. Our function in now nonlinear thanks to the RELU in the middle, and we now have a new knob that we can tune, this number H which corresponds to the number of RELU units that we have in the classifier. We can make it as big as we want. Congratulations, you've built your first neural network. You might ask, wait a minute, where's my neuron? In the past, when talking about neural networks, I remember seeing diagrams with dendrites, axons, activation functions, brains, neuroscience. Where is all that?

8 No Neurons

Yes, I could talk about neural networks as metaphors for the brain. It's nice and it might even be true, but it comes with a lot of baggage and it can sometimes lead you astray. So I'm not going to talk about it at all in this course. No need to be a wizard nor a scientist. New networks naturally make sense if you're simply a lazy engineer with a big GPU who just wants machine learning to work better. Now, though, is a good time to talk about the math.

9 The Chain Rule

One reason to build this network by stacking simple operations, like multiplications, and sums, and RELUs, on top of each other is that it makes the math very simple. Simple enough that a deep learning framework can manage it for you. The key mathematical insight is the chain rule. If you have two functions that get composed, that is, one is applied to the output of the other, then the chain rule tells you that you can compute the derivatives of that function simply by taking the product of the derivatives of the components. That's very powerful. As long as you know how to write the derivatives of your individual functions, there is a simple graphical way to combine them together and compute the derivative for the whole function. There's even better news for the computer scientist in you. There is a way to write this chain rule that is very efficient computationally, with lots of data reuse, and that looks like a very simple data pipeline.

10 Backprop

Here's an example. Imagine your network is a stack of simple operations. Like in your transforms, whatever you want. Some have parameters like the matrix transforms, some don't like the rellers. When you apply your data to some input x, you have data flowing through the stack up to your predictions y. To compute the derivatives, you create another graph that looks like this. The data in the new graph flows backwards through the network, get's combined using the chain rule that we saw before and produces gradients. That graph can be derived completely automatically from the individual operations in your network. So most deep learning frameworks will just do it for you. This is called back-propagation, and it's a very powerful concept. It makes computing derivatives of complex function very efficient as long as the function is made up of simple blocks with simple derivatives. Running the model up to the predictions is often call the forward prop, and the model that goes backwards is called the back prop. So, to recap, to run stochastic gradient descent, for every single little batch of your data in your training set, you're going to run the forward prop, and then the back prop. And that will give you gradients for each of your weights in your model. Then you're going to apply those gradients with the learning weights to your original weights, and update them. And you're going to repeat that all over again, many, many times. This is how your entire model gets optimized. I am not going to go through more of the maths of what's going on in each of those blocks. Because, again, you

don't typically have to worry about that, and it's essentially the chain rule, but keep in mind, this diagram. In particular each block of the back prop often takes about twice the memory that's needed for prop and twice the compute. That's important when you want to size your model and fit it in memory for example.

11 Training a Deep Learning Network

So now you have a small neural network. It's not particularly deep, just two layers. You can make it bigger, more complex, by increasing the size of that hidden layer in the middle. But it turns out that increasing this H is not particularly efficient in general. You need to make it very, very big, and then it gets really hard to train. This is where the central idea of deep learning comes into play. Instead, you can also add more layers and make your model deeper. There are lots of good reasons to do that. One is parameter efficiency. You can typically get much more performance with fewer parameters by going deeper rather than wider. Another one is that a lot of natural phenomena that you might be interested in, tend to have a hierarchical structure which deep models naturally capture. If you poke at a model for images, for example, and visualize what the model learns, you'll often find very simple things at the lowest layers, like lines or edges. Once you move up, you tend to see more complicated things like geometric shapes. Go further up and you start seeing things like objects, faces. This is very powerful because the model structure matches the kind of abstractions that you might expect to see in your data. And as a result the model has an easier time learning them.

12 Regularization Intro

Why did we not figure out earlier that t-models were effective? Many reasons, but mostly because t-models only really shine if you have enough data to train them. It's only in recent years that large enough data sets have made their way to the academic world. We'll look at another reason now, we know better today how to train very, very big models using better regularization techniques. There is a general issue when you're doing numerical optimization which I call the skinny jeans problem. Skinny jeans look great, they fit perfectly, but they're really, really hard to get into. So most people end up wearing jeans that are just a bit too big. It's exactly the same with deep networks. The network that's just the right size for your data is very, very hard to optimize. So in practice, we always try networks that are way too big for our data and then we try our best to prevent them from overfitting.

13 Regularization

The first way we prevent over fitting is by looking at the performance under validation set, and stopping to train as soon as we stop improving. It's called early termination, and it's still the best way to prevent your network from over-optimizing on the training set. Another way is to apply regularization. Regularizing means applying artificial constraints on your network that implicitly reduce the number of free parameters while not making it more difficult to optimize. In the skinny jeans analogy, think stretch pants. They fit just as well, but because they're flexible, they don't make things harder to fit in. The stretch pants of are called L2 Regularization. The idea is to add another term to the loss, which penalizes large weights. It's typically achieved by adding the L2 norm of your weights to the loss, multiplied by a small constant. And yes, yet another hyper-perimeter. Sorry about that.

14 Regularization Quiz

The nice thing about l2 regularization is that its very, very simple. Because you just add it to your loss, the structure of your network doesn't have to change. You can even compute its derivative by hand. Remember that the l2 norm stands for the sum of the squares of the individual elements in a vector. Which of these formulas gives you the derivative of the l2 norm of a vector?

15 Regularization Solution

You know that the derivative of 1/2 of x squared in one dimension is simply x. So when you take that derivative for each of the components of your vector you get the same components. Therefore, the answer is the third one.

16 Dropout

There's another important technique for regularization that only emerged relatively recently and works amazingly well. It also looks insane the first time you see it, so bear with me. It's called dropout. Dropout works likes this. Imagine that you have one layer that connects to another layer. The values that go from one layer to the next are often called activations. Now take those activations and randomly for every example you train your network on, set half of them to zero. Completely randomly, you basically take half of the data that's flowing through your network and just destroy it. And then randomly again. If that doesn't sound crazy to you then you might qualify to become a student of Jeffery Hinton who pioneered the technique. So what happens with dropout? Your network can never rely on any given activation to be present because they might be squashed at any given moment. So it is forced to learn a redundant representation for everything to make sure that at least some of the information remains. It's like a game of whack-a-mole. One activations gets smashed, but there's always one or more that do the same job and that don't get killed. So everything remains fine at the end. Forcing your network to learn redundant representations might sound very inefficient. But in practice, it makes things more robust and prevents over fitting. It also makes your network act as if taking the consensus over an ensemble of networks. Which is always a good way to improve performance. Dropout is one of the most important techniques to emerge in the last few years. If drop out doesn't work for you, you should probably be using a bigger network.

17 Dropout Pt 2

¿¿ When you evaluate the network that's been trained with drop out, you obviously no longer want this randomness. You want something deterministic. Instead, you're going to want to take the consensus over these redundant models. You get the consensus opinion by averaging the activations. You want Y e here to be the average of all the yts that you got during training. Here's a trick to make sure this expectation holds. During training, not only do you use zero out so the activations that you drop out, but you also scale the remaining activations by a factor of 2. This way, when it comes time to average them during evaluation, you just remove these dropouts and scaling operations from your neural net. And the result is an average of these activations that is properly scaled.

18 Next Assignment Regularization

Now it's your turn. In the assignment you'll be building a network by making the logistic classifier deep. You'll also play with the various forms of regularization. The amount of code that you need to change is very small, so don't hesitate to play around and try new things.