Module 21: Deep Neural Networks (Part 1)

Video Transcripts

Video 1: Introduction to Neural Networks

Today we're going to talk about another technique for machine learning known as a neural network. Neural networks were invented decades ago. And while they did see some use, it was only in the 2010s that their popularity suddenly exploded when researchers found great success applying them initially to image classification.

Consider the problem of trying to classify whether an image contains a horse or not. Well, an image is just a set of features like any other observation. So for example, if we have a picture that is 640 pixels wide by 480 pixels tall, and with three color channels for red, green, and blue. Then the image is just a list of 921,600 features. In other words, its dimensionality is 921,600. Now in principle, we could train a decision tree or a logistic regression model, or an SVM on these 921,600 features. But with so many features, we run the risk of overfitting. Unless maybe we had some absolutely gigantic corpus of example images. And this input set would need to potentially cover all kinds of different lighting conditions, rotations, et cetera.

Now one approach to improve our training process is to first preprocess the images, converting them into some often lower-dimensional form that captures the most important aspects of the image. For example, one such preprocessing technique that was rather popular in the late 2000s was to compute the histogram of oriented gradients of each image in your dataset.

I will not describe how this featurization process works. Though an example is shown on the screen. The resulting preprocess data has only around 10,000 features per observation, rather than a million. And more importantly, the features are also more useful for training a classifier than the original raw pixels.

Now, after preprocessing all of our images, in other words, converting them all into 10,000ish HOG features, where HOG stands for histogram of oriented gradients. We'd then train our classifier – for example, an SVM classifier – on our corpus of hogified data. Then when we wanted to classify a new image, we'd first hogify it. And then we would have our classifier give its best judgment based on these HOG features. This technique of first preprocessing then feeding to an SVM classifier worked pretty well in practice, though not as well as more modern techniques. The way forward seemed at that time to be to find the best possible preprocessing technique. And researchers all around the world spent huge amounts of time and effort trying to come up with newer, better techniques for preprocessing images.

Image classification is an incredibly important problem. Automated classification of images has near limitless applications, from social media to archaeology, from national defense to medicine. With so much potential impact, it's important to be able to compare the performance difference between different image classification techniques. To that end, researchers worked together to create a standard benchmark to compare their algorithms and preprocessing techniques. The most famous such benchmark is called ImageNet and was created by Fei-Fei Li at Stanford University. ImageNet consists of a massive database of labeled images. For example, this picture is of dogs, this one is of bell peppers, et cetera.

And in the 2010s, this dataset made an enormous impact on the machine learning field and in fact, the broader world. I'll briefly discuss the history in the rest of this video. And there's also a reading for this module that more fully covers the story.

So to participate in this ImageNet challenge, teams would come up with a new approach to image classification, often a new preprocessing technique. They would then train their classifier using the labeled training set. Then lastly, they would generate their best predictions for the unlabeled test set. Each year, the test set accuracy rates were published along with papers describing each team's approach. When ImageNet was first released as a contest, machine learning algorithms were far inferior to humans on this dataset. This dataset acted as a grand challenge, attracting researchers from all around the world to try to build an algorithm that exceeded human capability.

So here we see a plot of the participants in the ImageNet challenge from 2010 to 2012. In the year 2010, the spread of entrants was fairly wide, and no team was anywhere near human-level correctness, which is very roughly around 5%. Now by 2012, you'll see that the spread of the teams' performance was narrower. But human-level image classification, it just seemed pretty far off. I also show here a very rough line that gives the seemingly slow progress of the field on this grand challenge. Note, there's no specific reason we expect progress to be linear. This line is simply to provide a crude sense of progress.

Now there's one more entrant in the 2012 contest that I have not mentioned, AlexNet. AlexNet was the first neural network to find success in this challenge. Now, neural networks were not new, but AlexNet was the first time that a convolutional network was trained on a dataset that was large enough and on hardware that was fast enough to yield a neural network that generalized well on an important real-world benchmark. The improvement was immense. Note the massive gulf between AlexNet and the best traditional computer vision approach. AlexNet was an incredible breakthrough, inspiring teams around the world to tweak the approach and do even better. By the following year, almost all entries were neural network based, and many of them performed better than AlexNet had in 2012. By 2014, human equivalent classification was well within reach.

Here I plot a line showing just how rapid the progress was once neural networks entered the scene. By 2015, neural networks had reached almost human levels of performance. And by 2017, the last year of the contest, actually the vast majority of teams participating had algorithms that could perform at or slightly better than humans. So in less than a decade, ImageNet was solved. And, even more impressive, off-the-shelf software was capable of beating humans at this important task.

Now it's hard to overstate how important this moment in history was. AlexNet kicked off a revolution that didn't just affect the ImageNet challenge, but everything. Today, neural networks have been applied in many domain areas, resulting in vastly better performance in a lot of different fields, including speech recognition, artificial face generation, and have even beat humans at the ancient game of Go, an AI benchmark that was thought to be decades away. In this module, and the next, we'll be giving you a brief taste of the theory behind neural networks, as well as some practical skills that you'll be able to use to build and deploy neural networks for your application. In the next video, we'll start our journey by having Gabriel lay out the theoretical foundations.

Video 2: Foundations

In this video, we will understand the basic structure of neural networks as a generalization of linear regression. Let's recap our setup. We start with a dataset that consists of measurements from some system. There are D inputs or independent variables, x_1 through x_D, and one dependent or target variable y. Those show up in the DataFrame as the columns. We have N samples for x and y, which are arranged into rows. Our goal is to construct a model that, given some previously unseen input value x_{new}, will make a good prediction for its corresponding target value y.

One thing that we've found useful is to define a set of features, ϕ_0 through ϕ_{m-1}, based on the original inputs x. These are nonlinear transformations of x that are designed to improve the performance of the model. We've seen examples of this in linear regression and logistic regression, where nonlinear features introduced curvature into our regression lines and decision boundaries. Recall that in linear regression, we proposed a simple model, $h(x)$, that predicted the output as a linear combination of the inputs. In the one-dimensional case, this is simply a line with intercept b and slope a. And the task of linear regression was to find the values of a and b that minimized the loss function.

In general, if we have m features instead of just one, then the model is a sum of m terms. Those are $b + a_1\phi_1(x)$ through $a_{m-1}\phi_{m-1}(x)$. And we can express this compactly in a vector notation by defining a weight vector A, a feature vector $\phi(x)$, and an offset b. The model then becomes $h(x) = A^T\phi(x) + b$. The offset is sometimes called the bias, but this really has nothing to do with the statistical bias of the model. So don't let that confuse you.

The linear regression problem can then be expressed very succinctly as finding the parameters a and b that minimize the squared loss function, which is the sum over all of the data points of the square of the difference between the predicted output $h(x_i)$ and the true output y_i. Plugging our linear regression model into this, we get a summation of $(A^T \phi(x_i) + b - y_i)^2$. In optimization theory, we call this a convex problem. And that's good news. It means that the cost function is shaped like a smooth bowl. And our goal is simply to find the point at the bottom of the bowl. And in gradient descent and its close cousin, stochastic gradient descent, we have very effective methods for finding this minimum.

Here's a representation of the linear regression model as a graph. Think of each of the lines in this graph as transmitting a single number. We enter the numbers on the left at the input nodes, which are marked x_1 through x_D. These are all of the components of the input vector. These numbers then flow from left to right and enter the boxes in the middle marked with ϕ_0 through ϕ_{m-1}. Each ϕ box applies a nonlinear feature transformation. The outputs of these boxes then go through lines which multiply them by weights a_0 through a_{m-1}. And then finally, on the right, they are added together and...with the offset b, and this is the output of the model. Notice how this corresponds with our formula for linear regression. First, the ϕ functions are evaluated, then they are multiplied by weights, and added up. The objective of the training process is then to adjust the coefficients a and b so as to minimize the error.

Neural networks start from the same picture and generalize it in several ways. First, they use very specific feature functions, which in the neural network community are called activation functions. A few of the most common activation functions are the sigmoid function, which we've seen already is the core of logistic regression. In a neural network, it acts like a switch, either opening or closing a channel. The tanh or hyperbolic tangent function; this function puts saturation limits on the output of the neuron. It allows small values to go through more or less unchanged, but truncates large positive or negative values. And the rectified linear unit or ReLU. This just takes the maximum between the input value and 0. And thus it prevents any negative numbers from going through.

A second generalization of neural networks is that they add coefficients to all of the lines in the graph, not just the ones on the output side. So whereas before with linear regression, the input to the feature vector was simply x. Now with neural networks, it will be a matrix $A_1 x$ plus an offset vector b_1.

The third generalization of neural networks is that they allow for many layers to be used. These layers are set out sequentially so that each feature vector ϕ is fed by a matrix A multiplied by the output of the previous layer plus an offset vector b. Let's use capital letter L for the number of layers in the network and count them from left to right. There are $L-1$ internal or hidden layers. And the last one is the output layer. We'll use lowercase c for the outputs from each layer. The final output of the model $h(x)$ is the transformation of c_{L-1} with coefficients A_L and b_L. Then c_{L-1} is the output of layer $L-1$. This equals the feature ϕ_{L-1} applied to the input of that layer, which is the transformation of the output of the previous layer through coefficients A_{L-1} and b_{L-1}. And we continue in this way, feeding each layer with the output of the previous layer, which is transformed through a dense mesh of coefficients A and b until we reach the input.

A final generalization has to do with the output layer. By adding a sigmoid activation function to the output node, we immediately transform the regression model into a binary classification model. Later we will see other output layers that can be used to create multiclass classification models.

As we learn more about neural networks, we will introduce other interesting variations. The training problem for neural networks is to find all of the A and b coefficients such that the loss function is minimized. This is very similar to linear regression. However, the model now is much more complicated since it consists of a set of L-nested functions. This complexity of neural networks allows it to take different forms and adapt to different data sets. However, it also makes neural networks much more difficult to train.

Video 3: Neural Network Playground

Gabriel just gave you some nice notation for neural networks. So I want to show you a specific example, so we know exactly what's going on. Let's suppose we're building a model for Titanic passengers. We know the fare that the passenger paid and we know their age. And we want to know: Did they survive the sinking of the Titanic?

So here I have A_1, A_2, b_1, and b_2, as Gabriel described. And here we have the activation function, which – in my case – I've chosen the logistic activation function. So these are all the same familiar symbols that we saw on Gabriel’s slides. But now I want to show you how we actually use this to compute an output value. So a different way to view these matrices is as follows. Here, I'm showing the resulting weights and intercepts that we get out of the A_1 and b_1 matrices for each of the three neurons in our hidden layer.

So, for example, the top-left entry in A_1 is w_1 for this first neuron. So in order to compute the overall output of this network for a specific value, we'll just plug in the fare and the age, compute some weighted sums, put those into the logistic activation function, and then pass those to the next level to repeat the process again. So for example, if a passenger paid $8 and their age was 20, the network will proceed as shown here. These are all the values that get generated.

Now, to really get this, maybe let's not look at the final answer, but let's see the answer come to exist. OK. So if a passenger paid $8 and they were 20, then what the first neuron will do is it will compute the value −15.4. That's the weighted sum of the fare and the age, including this interceptor. And where did these weights come from? Well, those were from the matrices that Gabriel described A_1 and b_1. We then apply the logistic activation function and get 1.97 times 10 to the minus 7th. OK, so then we repeat the same thing for the second neuron. And so you might try to compute these values yourself. If you'd like, you can pause just to check your understanding.

So if I do that, I get back 1.405 as the weighted sum and 0.803 as the value after the activation function. Lastly, this last neuron at the bottom computes its activation, which is 0.2071. And then these three intermediate features are fed to our output neuron. It in turn generates a value, ultimately, of 0.27. The network believes there's a 27% chance that this passenger survives. So that's what the A_1, b_1 matrices mean, and the A_2 and b_2 matrices.

So the way I think about it is that each layer is computing a set of features that could be used by the later layers. Instead of having the fare and the age, we have three features, which I don't have a name for them. I can't describe to you what they are, but they're used by the output layer to compute the ultimate result.

Now, as we noted, neural networks, they can have many layers. So the front layer, we have the inputs. In this case there are two of them. Then we have the hidden layer, and then we have an output layer, which produces our predictions. We could have many layers. For example, here's three, four, two for neurons in three different hidden layers. And a single output, survival probability. So in this case, the first hidden layer generates three new features, which are then used to generate four new features, which generate two new features, which finally give us our prediction. So let's see a little live demo that showcases just how powerful this idea is.

Consider the data shown. Now this is clearly not linearly separable in the x, y feature space. There's no line you can draw that separates orange points from blue points. However, with a little cleverness, I'd like you to try and think about this. Can you come up with a feature in terms of X_1 and X_2? So now I'm considering the y here, that's X_2. What's some feature, some function of X_1 and X_2, which if you gave it to logistic regression, could predict the data's classes with high accuracy? It's actually kind of a challenging problem. See if you can come up with something. If not, I'll spoil it in a moment.

So if we compute the feature X_1X_2, that's actually really useful. If $X_1X_2 > 0$, then the data is blue, right? So if the Xs are both positive, you're up here. Or if the Xs are both negative, you're down here. OK, so that's basically a trick you could use. A featurization of the data that makes this linearly separable. OK, great. So now let's see how a neural network can come up with that on its own. So instead of us having to be clever and come up with that rule, let's see a neural network do it for us. So I'm going to be using this Neural Network Playground created by the folks at TensorFlow. And I find this very useful to get a sense of how neural networks behave. So the first network that I've linked here is the very simplest network. In fact, this is just logistic regression. And so if I train this model, I do it over and over. You'll see that it generates a model which cannot predict our two classes with perfect accuracy. It's just impossible. There's no line that separates them. Now, if I add more layers, then I can get more complex decision boundaries than just a line.

So this link here shows what happens if I have a single hidden layer with three neurons, similar to our Titanic example from before. Now, every time I run this, I'm going to end up with a slightly different model. Because each time I'm doing a gradient descent and there's just no telling exactly where that gradient descent is going to take us. So if I try this again, this time I get, OK, a model that looks pretty similar. If I do it a third time, this time I happened to get something that is slightly different. Instead of it being a diagonal band. Now it's these two little lobes. OK, and that's similar to what I showed on my slide.

Let's try an even more complicated network. So here, if I have three hidden layers with three neurons, four neurons, and two neurons respectively, if I now train my network, well, nothing seems to be happening. OK, so what's going on here? Well, it turns out that it's pretty tricky to train a network that's this deep with the activation function we've given, a sigmoid. So we need to use something else, like a ReLU, or a hyperbolic tangent. I'm going to use hyperbolic tangent. You could also use ReLU. And in this case, the network will actually behave itself and converge somewhere. So now every time I train this network, you'll see that it gets something a little different each time.

And so what I want to focus on here is something very special that has happened. Remember that each network is computing a new set of features. So in some sense, you can think of this neuron as a neuron that computes: How top-lefty is the data? If the data happens to be more towards the top left, then it will activate highly, and so forth. Now at this final hidden layer, before the output layer, you'll notice something really interesting. Both of these two neurons have basically discovered the feature $X_1 \times X_2$. It has automatically come up with an output. Well, it will reflect the data that we're trying to optimize very well. So what's miraculous about this is we didn't have to be clever. It automatically found this feature.

Now if, instead, I go back to a really simple model with no hidden layers, but now I provide that feature myself of $X_1 \times X_2$. Now it's going to do just fine. So if it has access to this feature directly, it'll do really well. But what we just showed is that if we have a more complex network, it can automatically learn features like $X_1 \times X_2$. And so on the homework, you'll have a chance to think a little bit more about this idea. But it's a really powerful and cool idea that seemingly complex or counterintuitive features can be automatically generated.

Video 4: Keras

Let's see an example of how we can create a neural network in the Keras library. So here I'm in Google Colab, which has everything already set up nicely for doing the Keras library. And one thing you want to always do is make sure that your runtime type has hardware accelerator GPU. If it's None, your code will run much more slowly.

So this code up top, all it does is it creates a dataset, not super important right now. But you'll see it generates a dataset where everything in the top-left is orange, everything in the top-right is blue and so forth. Just like we saw in our Neural Network Playground. I've also written a function here that creates the decision boundaries as a plot. Not very important, but I want to make sure you know, that I've set up this function.

Here comes the neural networks part. So I'm going to import the Keras library, which does all the neural network stuff for us and has a really nice API for specifying a network. So for example, suppose we want to create a neural network with three hidden layers, where the first layer has three neurons, the second layer has four, and the third has two. We would do this right here. These are each of the first three layers. And Dense is just the name for having connections between all of the neurons from one layer to the next, as we've been doing so far.

I picked the ReLU activation function as opposed to the hyperbolic tangent. And it's a pretty common choice in machine learning today because it's very rapid to optimize. Gradient descent has a good time with it. And then we have, at the outer layer, an activation function that is sigmoidal. And that's because we want to produce a probability that is between zero and one. And so that's what this will do. The sigmoidal function will ensure that the output of the whole network is between zero and one.

So once we create this model with this syntax here, we have set up the model, but it doesn't know anything yet. The weights are not yet initialized. Now Keras models require that we specify the optimization function, the loss function, and the metrics to track as the model is trained. And so here we see an optimizer I picked is rmsprop. What is that? Well, that's the flavor of stochastic gradient descent that I'm using. There are other choices, but this is one that's very popular.

For the loss function, since we're doing a binary classification problem, I'm using the binary cross-entropy. And then lastly, I'm going to keep track of the accuracy over time, which you can visualize later. And you'll do that on the homework. So here I'm going to compile my model. And yet, again, I'm not yet trained. The model doesn't know anything about the world. But now it knows its architecture and the properties that it has as it trains itself. It's going to use this stochastic gradient descent solver to minimize this loss function.

So then lastly, I'm going to actually train my model on the data I provide. So given these x and y values, I'm also telling it I want you to do five rounds of stochastic gradient descent. And then also I want you to do, when you're doing stochastic gradient descent, use a batch size of eight. And so if I do that, I see the accuracy evolving over time. And in this case, even after the first epoch, after it goes through all the data points, it's gotten about as good accuracy as it will, ever. But in other examples, as we'll see later, this accuracy tends to grow between these training epochs.

Now what did this model actually learn? What is the decision boundary it came up with that's getting the 70ish% accuracy? Well, I'm using my visualize_decision_boundaries function here. I'll run it live. And you will see, when this pops up, the model that this network has learned, where we had these three hidden layers. So here it is. The model is, everything in the top-right is an orange data point. So it's an OK model, but not a particularly good one because it totally misses the point with this data up here.

Now I can create a more complex model. For example, if I have three layers, that each have 16 neurons, instead of 3, 4, 2. This is basically the exact same code, but just with more neurons per layer. If I train this, it will tend to need more epochs to train with a more complex network. So you'll notice that after the first training epoch, it only had 94% accuracy, but then it was able to get up to 99, then 1, and so forth. Now kind of interesting, you'll see that the loss decreases monotonically, but the accuracy kind of bounces around a little bit. You can have a lower...a higher cross-entropy loss and a higher accuracy as well. Now the long-term trend is that when the loss goes really, really low, you expect the accuracy to be really good. And indeed you see that the losses get indeed very small.

So now, if I look at the model, the decision boundary that the model learned that has this extremely low loss and has 100% accuracy, it'll look exactly as you would expect. So this more complex network creates this decision boundary. So the glory of the Keras library is it makes it very simple in code to create really arbitrarily complicated neural networks. And so in the following videos, we'll see just how powerful this can be in all kinds of useful domains.

Video 5: Multiclass Classification

Up until now, the neural networks that we've worked with were trained to solve binary classification problems. That is, problems with two classes. Survived or perished, won or lost, cat or no cat. In this video, we will learn to build neural networks for multiclass classification problems. We've already seen methods for doing this in the context of logistic regression. Recall that there were three approaches. One versus rest, one versus one, and multinomial logistic regression. The one versus rest and one versus one methods are generic. And they can be applied to classification models other than logistic regression, including neural networks. And they often work well, but both require that we train a set of classification models instead of just one. And then subject these models to a majority or soft vote. Much as we did with ensemble techniques.

Neural networks are often very expensive to train. And so having to create multiple networks can be a drawback. The more common approach to multiclass neural networks is closer to multinomial logistic regression in that it requires only one model, and that model is adapted to produce probabilities for each of the output classes. Let's see how it works.

Adapting a neural network to generate class probabilities is actually very simple. All we need to do is to replace the single-node output layer with a multi-node output layer called a softmax layer. The softmax layer has one output node for each class. So if we are working with the Iris dataset, we will have three output nodes, one for virginica, one for versicolor, and one for setosa. The softmax layer proceeds in two steps. First, it sums the inputs to produce numbers x_1 through x_m for each of the m classes, in this case three. The model is trained so that the largest of the x's corresponds to the predicted class for the given input sample.

The next stage normalizes the predictions using the formula shown here. Recall that we've seen this formula before when we computed the class probabilities in multinomial logistic regression. This function is known as the softmax function, and it is a generalization of the logistic or sigmoid function to multiple classes. The inputs to the softmax function are the x_i's, which can take any value, positive or negative. Softmax then warps those inputs in a way that a) preserves their order so that if x_i was less than

x_j, then y_i will be less than y_j, b) the y's are all between zero and one, and c) the y's add up to one.

But why is this useful, you may ask? If softmax preserves order, then couldn't we have chosen the classes by the largest value of x? Doesn't the fact that softmax preserves order, render it unnecessary for prediction? Well, yes. But the real importance of softmax has more to do with the training of the model than with prediction. As with any supervised learning algorithm, a neural network is trained by presenting it with a series of samples for which we are certain of the answer.

For the Iris dataset, we show the model the dimension of the petals and sepals, and we say, this is a virginica Iris. We are 100% certain that this is virginica and not setosa or versicolor. We encode this statement with a vector in which each entry is our certainty value for that class. In this case, a 1 for virginica and 0s for setosa and versicolor. Other samples would be represented similarly with a 010 vector if they were versicolor, or a 001 if they were setosa. This way of encoding multiclass variables is called one-hot encoding. And we've seen it previously, when we learned about linear regression.

The softmax function fits perfectly with one-hot encoding because it produces values between zero and one that can also be interpreted as a certainty value for each of the classes. Notice that this would not have been possible if we were to eliminate the softmax function from the output layer and instead train the model using the raw x-values. In that case, we would have to encode our output labels with numbers ranging from negative to positive infinity. And it would be impossible to express total certainty that a flower is, say, a virginica Iris.

Using softmax also gives the predictions of the neural network a probabilistic interpretation. If our network predicts values of 0.8, 0.05, and 0.15, we can interpret these as the model saying, I am 80% certain that this is a virginica Iris, but also give 5% chance to versicolor, and 15% chance to setosa. But be careful not to take these outputs as the true probabilities about the real world. Remember that neural networks are difficult to train and often end up in a bad local minimum. So a particular network may not be representative of the true distributions of Iris flowers.

Let's try this out using the Petal_length and Petal_width columns of the Iris dataset. The first step is to one-hot encode the class labels. And we can do this with scikit-learn's LabelBinarizer class. To create an encoder, we first call its constructor and then pass the labels into its fit_transform method. Internally, the label binarizer will build and store a map between the string labels and a set of one-hot encoded arrays. And it will return the arrays corresponding to the labels that we gave it. Here we see the one-hot encoded array is corresponding to the first ten flowers in the dataset, which are all setosas.

We can also go in the opposite direction from one-hot encoded labels to string labels by using the inverse_transform function of our label binarizer. Here we confirm that the first ten entries are indeed setosa Irises. And here, we build a simple neural network with two layers, an internal dense layer with five ReLU units, followed by a softmax output layer with three units for the three output classes. We then compile and train the model using the one-hot encoded labels for the output. Here, we see that the accuracy reaches about 96% after 200 epochs of training.

You should think of softmax as a multinomial logistic regression embedded within a neural network. It's a simple way of adapting the model to produce multiclass predictions. And it is trained along with the rest of the network with stochastic gradient descent. In the next video, Josh will expand on this topic and tell us more about how to effectively train a neural network. I'll see you later.

Video 6: Two Bits of Syntax

In this short video, we'll introduce two features of the Keras library that might be hard to discover on your own. Here we see the code that Gabriel used to train his Iris model. As Gabriel mentioned, this model has one hidden layer with five ReLU units. After computing these five new features, they are fed to a softmax unit with three outputs, representing the estimated probability that a given input flower is a setosa, versicolor, or virginica. Recall that Gabriel transformed the classes into a one-hot encoding. And because he was doing a classification problem with one-hot encoded classes, he used the categorical_crossentropy as his loss function.

So the first Keras feature we'll discuss in this video is integer encoding, an alternate technique for representing outputs. Specifically, instead of one-hot encoding the outputs, this code assigns each class an integer: zero for setosa, one for versicolor, two for virginica. Make sure you understand the difference between this integer encoding and the one-hot encoding approach that Gabriel showed you.

Training a Keras model with this integer encoding is very similar to what Gabriel did before. The only difference is that we must select a different loss function. Specifically, the sparse_categorical_crossentropy. The choice of these two loss functions makes no difference, in terms of model performance. We simply need to pick the appropriate loss function to match the encoding for the observations that we're trying to predict. Categorical_crossentropy for one-hot encoding, and sparse_categorical_crossentropy for integer encoding.

The second feature I'd like to highlight is the ability to plot the loss instead of the accuracy. Here, I show a neural network with one hidden layer containing five neurons for our Iris dataset. This is the same model that Gabriel built. Though, when I ran the experiment, I got slightly different parameters due to the fundamentally non-deterministic nature of neural network training. As you saw in Gabriel's video, the accuracy of our model generally increases with the number of training epochs. The code I used to plot this data is as shown.

Now if we want to plot the loss, instead, we simply replace the string "accuracy" with "loss." This yields the plot shown. Note that even after 500 epochs, the model is still not entirely stable and is eking out just a bit better loss with each epoch. Note also that here we're using the entire dataset and we have not set aside a separate testing set. In other words, if our model were overfitting, we would not see that here.

Video 7: Hyperparameter Tuning

Let's wrap up today by talking about neural network hyperparameters. So neural networks have a huge number of hyperparameters. These include, for example, the number of layers in the network, also known as the depth. The number of neurons in each layer. In other words, the width of each layer. And the activation function for each neuron, and so forth.

So for Gabriel's example, the network had a depth of two. There was one hidden layer and an output layer. And then the width of the hidden layer was five neurons. We also know that the hidden layer used ReLU units and the output layer used softmax. But we could have done something completely different. For example, we could have 30 layers with 100 neurons each. Or 100 layers, where the first layer has 1 neuron, the second has 2, the third has 3, and so forth.

So given this infinitude of possibilities, how do we know which to pick? Well, let's see what happens if we tweak the network architecture. Suppose we add another layer of five neurons. This is trivial in Keras. We simply add another line to our sequential model definition. So here I decided the second hidden layer should have five neurons. But it could have been any other number, right? I could have picked 3, or 6, or even 5,000. That is, the number of neurons does not need to match between the first and second hidden layers.

Now if we compare the loss versus the training epoch, we see that this new two-layered architecture seems to yield slightly better results with fewer training epochs. Now suppose that, rather than having two hidden layers of 5 neurons, we have just one hidden layer of 16 neurons. This model's loss-versus-training curve is quite similar to the model from before, where we had only five neurons in the hidden layer. In other words, the extra expressive power of the other 11 neurons in the hidden layer, it didn't really seem to help very much.

Now Keras lets us easily create a model that has a large number of parameters. For example, this code shown has three hidden layers of 32 neurons each. To very roughly estimate the number of parameters in this model, consider that each connecting line in our network has its own weight, and each neuron has its own intercept. So for just the connections between two of the 32 neuron layers, we need 32 squared parameters, or 1,024. So this model has several thousand parameters and we're only going to be training it on 150 data points.

Now we see that this model actually converges in many fewer epochs than the earlier simpler models, though with more noise in the loss between the training epochs. If we visualize this new model's decision boundaries, we see that it doesn't appear to be obviously overfit. That is, the network isn't drawing little boundaries around just single outlier data points. But actually, if we added enough complexity to the network and we gave it enough time to train, that would eventually happen.

So which hyperparameters do we select? Do we want the simple model with just one hidden layer of five neurons for a total of a few dozen parameters? Or at the other extreme, do we want this new complex model with three hidden layers of 32 neurons each, totaling many thousands of parameters? Well, we've already used the most important tool for deciding this question earlier in this class: Cross-validation.

Note that given the high cost to train neural networks, it is much more common to use simple cross-validation rather than k-fold cross-validation. In other words, we'll typically hold aside a dev set, aka a validation set, and use that to evaluate the generalizability of our model. So for this specific example, we take a random sample of, say, 40 of our 150 flowers and set those aside as a validation set. Which, as usual, are interchangeably referred to as the development set. We'd then train a bunch of models, like the ones we just tried. And then we'd see which yields the best loss on the validation set.

So for example, the code shown computes the training and dev set losses versus the epoch for the network with one hidden layer of five neurons. The two main changes in the code are that, first, we shuffled the dataset and then used np.split to get a training and dev set. And then, when we called the fit method of our neural network, we provided the validation set so it could keep track of the validation loss over time. Now plotting these two losses together yields the figure shown. And we see that the development set loss, it reaches its minimum somewhere around 800 epochs in. After that, the model starts to overfit. And so it would make sense to stop training early, at around 800 epochs, in order to avoid overfitting.

If we look at where the dev loss bottoms out, it comes to around 0.15, though I'll note that number is fairly noisy. Now by comparison, here we plot the cross-entropy loss for a model with two layers of 16 neurons each. And we show that the code here, it generates these losses. Now again, we see that the validation set loss decreases to some minimum. But this time, somewhere around 200 epochs in, before increasing back up due to overfitting. For this network, the validation set loss is much noisier. We'll see on the homework that standardizing our data, using a standard scalar, that will reduce this noise. And actually will also make our models converge more quickly.

Now this model also bottoms out at an error of around 0.15. However, the noisiness of the dev set loss indicates to me that this model is less likely to generalize well. If I had to pick between these two models, I'd choose the simpler one that has just five neurons and a single hidden layer. Note that if we'd looked at accuracy rather than loss, as shown in these two figures, the models are essentially indistinguishable. Both level off at 92.5% accuracy. And so from that perspective, there's no reason to choose one versus the other based on the accuracy metric.

In principle, we can repeat that same process for a large number of candidate model architectures. By generating a table of validation set loss values for each of those network architectures, you can choose the one that works best. In other words, the one that has the lowest development set cross-entropy loss. Keep in mind that the resulting value is a bit biased in its estimate of model performance. And so we'd need a totally separate one-time use test set if we wanted to give a final, quantitative assessment of the model's loss or quality.

Now you can also, in principle, generate curves showing the development set loss versus the depth of the network. Or the development set loss versus the width of one or more layers of the network. Though, we will not do so here. Note that rather than manually generating a table of validation set loss values, Keras has a tool called KerasTuner that allows you to search a pre-specified set of hyperparameters. KerasTuner works similarly to GridSearchCV in sklearn, though we will not discuss KerasTuner in this video today. On the homework, you'll have a chance to explore hyperparameter tuning in the context of a larger dataset.

Video 8: Computing Batch Gradients

Let's discuss how we can look at the loss after every mini-batch, rather than after every epoch. That will help us understand more deeply stochastic gradient descent.

So Keras uses stochastic gradient descent as we described in an earlier module. So that means, rather than compute the true gradient on the entire dataset, the algorithm is only considering a mini-batch at any given time, yielding an approximation of the gradient. So we didn't specify the batch size. And that means it used the default of 32. In other words, every time it revises the neural network parameters, it's only using 32 of the 150 flower observations in order to decide how to revise them.

So the loss tracked by the Keras history, it only gives the error after each training epoch has completed. So an epoch consists of a consideration of all of our data points. Now since we had a 150 data points and 32 observations per batch, that means there's five mini-batches per epoch. So in other words, in a single epoch, the following happens. The stochastic gradient descent algorithm, it computes the gradient of the cross-entropy loss using 32 of the data points. Let's say, data points 0 through 31. Then, after taking a step down, using that approximation of the gradient, we compute another approximation of the gradient on data points 32 through 63. And again we take a step down in the direction of the gradient.

Then we compute the gradient on data points 64 through 95, and we take another step down. Then we compute the gradient on data points 96 through 127 and take another step down. And then, as the last step in our batch, we compute the gradient on data points 128 through 149 and take one more step down. Then, and only then, does Keras record the loss, once that entire epoch of five batches is complete. And that yields the cross-entropy plot here, that we see in this visualization.

So remember from our previous module on stochastic gradient descent that for practical problems, each epoch of stochastic gradient descent should consider a random ordering of the points. So in other words, SGD doesn't really work very well if you do it as I just described, where the first batch is the first 32 points, the second batches the next 32 points, and so forth. It's going to do something more random. Now we can define a custom callback to track the loss after every batch, yielding the array of values called batch losses. I'm not going to show the code that generates this array here, but you've seen it here on the homework.

The result of that code is given in this figure. Here we see the cross-entropy loss after each batch. Since we had 500 epochs, each consisting of five batches, there's 2,500 total training batches on the x-axis. We see now a much noisier descent down towards the lower cross-entropy loss. Note that the epoch loss that Keras gives us is just every fifth sample of the batch losses that we just computed. So in other words, if we plot batch_losses[4::5], we get the exact same figure as we did when we tracked the loss after every epoch. In other words, if we look at the loss after every fifth batch, we're looking at the loss after every entire training epoch, and that gives us the smooth curve from before.

So let's see what happens by contrast, if we set the batch size to one. When we do this, we see that the loss of our model is much noisier after each batch. Now that's not such a surprise since each gradient descent step is now based on only a single data point. So here, we see the cross-entropy loss versus the batch number for our two choices of batch size. In this first figure, we see the loss versus the training batch, if our batches have 32 samples. We saw that figure before. And in the adjacent figure, we see the loss versus the training batch, if our batches have only one sample.

So looking at these side-by-side, we see that after 2,500 gradient descent steps, the model being trained on only one sample at a time, it has not made as much progress as the version that is being trained on 32 samples at a time. Now it's not so surprising because the quality of the gradient approximation for this one-sample-at-a-time approach is slower.

Now if we track the time needed to train these models by each step of the process, you'll see that the gradient descent steps for a batch of size one are much faster. This is because computing the gradient with respect to only one data point is faster than using 32 data points. That's just like we discussed in an earlier module, where the batch size controls the tradeoff between the quality of the gradient approximation, and the time needed to compute the gradient approximation.

So that brings us to this counterintuitive phenomenon, that if we plot the cross-entropy loss versus the epoch number, we do see a less noisy trajectory than before. But if we compare the cross-entropy loss versus epoch number for a batch size of 32 versus one. We see that, strangely, convergence seemed to happen much faster when we picked this batch size of one.

Now at first glance, that seems to contradict the entire story that I told in the stochastic gradient descent lecture, and which I have reviewed today. How is this lower-quality approximation of the gradient yielding faster convergence against the epoch number? So here's where you've got to be really careful about what's happening here. The answer is that for batch size 1, there're actually 150 different gradient descent steps per epoch. But by contrast, for a batch size of 32, there's only 5 gradient descent steps per epoch. So ultimately that comparison was not really fair.

So I hope that helps elucidate why that funny phenomenon popped up that you may have observed.

Video 9: Conclusion

In earlier modules, we saw how linear models could fit nonlinear data. The key idea is that we would compute a set of features, which were nonlinear functions of the original features. For example, the squares of the features, or the cubes of the features, or some kind of polynomial combinations of the features. Other examples include histograms of gradients. Now we can either explicitly compute these features or we can use kernels. In this approach, we must specify the nonlinear featurization functions in advance.

The other approach is what happens with neural networks. Neural networks work by successively transforming inputs through a sequence of layers. Each transformation unit is called a neuron, and each neuron behaves much like a logistic regressor. They compute a weighted sum of their inputs plus an offset, and then they apply some activation function, not necessarily logistic. The earlier layers compute features that are ultimately used by later layers. Now what happens is that several layers combined together, they can yield arbitrarily complex, nonlinear featurizations of the original data. So unlike the classical approach, we do not have to specify the nonlinear featurization functions in advance. Instead, these featurization functions are automatically learned as part of the training process. So there's no need for people to do a whole PhD designing techniques like histogram of gradients because the neural networks designs these featurization functions on their own.

Now in many domains, neural networks have dominant performance. In some ways, neural networks are the best of both worlds. They work really well, and they also don't require engineers to design custom featurization functions. It's a miracle. Now on the downside, neural network training is slow, and it requires special hardware. And for that reason, we switch libraries to Keras. We saw how to train and compare models in Keras. And in the next model, we'll go into more depth, including the special architecture that was used by the AlexNet neural network.

So that's it for today, everybody, and we will see you next time.