[SQUEAKING]

[RUSTLING]

[CLICKING]

RAMA RAMAKRISHNAN: OK. All right. Let's get going. Today is going to be packed. I'm going to spend the first roughly half of the lecture on actually building a model-- a Keras model in Colab-- to solve the heart disease problem we saw earlier, and then switch gears halfway, and then talk about how to solve image classification. So we're going to do two Colabs today.

I've been talking about Colab. Colab, I've been teasing you. We'll actually do Colabs today. All right. So summary of where we were. By the way, I've shut off the lights in the top because, when I switch to Colab, it's going to be much better for you folks, particularly the folks in the back, to be able to see it. But I hope you can see the slide right now. Yes? OK. Great.

So this is just a quick recap of what we did last class. Broadly speaking, training a neural network essentially is no different than training other kinds of models. We have a bunch of parameters, i.e., weights and biases, and we need to use the data to find good values of those weights.

And what does good mean? Typically, it means that we define some measure of discrepancy between what the model predicts for a given set of weights and what the right answer is, what the ground truth answer is. And then we try to find weights that minimize this discrepancy. That's it.

And this notion of discrepancy is called a loss function. So broadly speaking, the overall training flow is that you define some network. It has an input. It goes through a bunch of layers. You come up with some predictions. You take the predictions. You take the true values. And then those two go into the loss function, i.e., the discrepancy function. And then you come up with a loss score.

And then you send it to the optimizer, which then proceeds to calculate the gradient of this loss function with respect to all the parameters. And then it updates all the weights using that gradient, and then the process repeats. That's it. So that is the training flow. Quick recap.

Now, we also talked about the optimization algorithm we're going to use, which is called gradient descent. And gradient descent, as you noticed, in each iteration, every data point is being used to make predictions and therefore to calculate the loss, and then to calculate the gradient. And then we pointed out that gradient descent is actually not as good as something called stochastic gradient descent-- stochastic gradient descent-- where, instead of taking all the points, we just randomly choose a small number of points, pretend for a moment as if those are the only points we have, make predictions, calculate loss, calculate gradient, and go on.

So that was the basic idea behind stochastic gradient descent. Two different kinds of things. Now, what it means is that when we actually start training the model, as we will in a few minutes, because we only take a few points at a time, we have to be a bit careful in what's going on. And I want to make sure you clearly understand what the differences are before we actually get to the Colab. OK?

All right. So there is the notion of an epoch. An epoch essentially just means that we make one pass through the training data. All the training data. We make one pass through it. And so what is one pass is that if you have something like gradient descent, one pass means every data point is sent through the network. We calculate its predictions, calculate the loss, calculate the gradient.

We run every training sample through it. We calculate the gradient, which is just this thing here. I mean, I will sometimes say d of loss, dW. Derivative of loss with respect to W. Sometimes I might use this nabla symbol. These are all interchangeable.

So we'll calculate the gradient, and then we update using some version of this. But we just do it once at the end of the epoch. Because if you have 10 billion data points, every one of them flows through, you get 10 billion outputs. And then we calculate the epoch just once. At the end of this thing, we calculate the gradient and update once. One update per epoch. Yes?

Now, in stochastic gradient descent, what we do is that we process the data in batches. Small numbers of points at a time. And these are called-- technically speaking, they're called minibatches. I don't know about you. I just get tired of saying minibatches, so I'm just going to say batches from this point on. OK? And in fact, that is widely done in the literature.

So we'll have to process it in batches. So we take the training data. And then we divide it up into batches-- batch 1, batch 2, all the way till the final batch. And so what we do is, for each batch, we basically do gradient descent. For each batch, we take batch 1, and then we run just the training samples in that batch through the network to get predictions. We calculate the gradient, we update the parameters, and then we go to batch 2.

Then we go to batch 3 and so on and so forth. So pictorially, this is how it's going to look like. Let's say the first batch is, say, 32 points. We take those 32 points. We run it through the network, get all the stuff out. We calculate the gradient, update the weights.

So when we now get to batch 2, the weights have changed. They have been updated. And then we do the same thing with batch 2, batch 3, and all the way till we get to the end of the thing. And when we are done with this thing, this whole thing is called a what? An epoch. This whole thing is an epoch. OK?

All right. So the question, of course, is that if you have a bunch of data points, and you're going to run stochastic gradient descent on it, in a particular epoch, how many batches are going to be there? How many batches are going to be there?

Now, Keras is going to calculate all this stuff. You don't have to worry about it. But you just need to understand exactly what happens. So my philosophy, by the way, is that you have to know the details of what's going on. If you don't know the details, and if you haven't figured out at least once, you will not actually be able to think new and creative thoughts for a new problem. It's because the concepts are not manipulable in your head yet. OK. Please use the microphone.

AUDIENCE: So when we talk about SDG and we are talking about-- we are only taking some part of it, is it what we are saying is that we only take some variables, or we're only taking some part of the data?

RAMA RAMAKRISHNAN: We are taking some rows.

AUDIENCE: We're taking only-- oh, right. So that--

RAMA RAMAKRISHNAN: Data points.

AUDIENCE: That means the batch itself.

RAMA RAMAKRISHNAN: Exactly. So for example, let's say you have 1,000 data points, 1,000 rows of observations, 1,000 patients in the heart disease example, or 1,000 images that you're trying to classify. You take, let's say, 32 of those images, 32 of those patients, and that's a batch. Then you go to the next 32, then the next 32, and so on and so forth, until you run out of patients or run out of images.

AUDIENCE: And each iterative time you are updating with the weights, the weights that you've got?

RAMA RAMAKRISHNAN: Correct.

AUDIENCE: And it means you keep correcting it or keep moving towards--

RAMA RAMAKRISHNAN: You're basically updating the weights as you go along.

AUDIENCE: Updating the weights.

AUDIENCE: And what we are calling the epoch is ultimately the equation of loss function that we are trying to get.

RAMA RAMAKRISHNAN: No, an epoch-- see, the thing to remember is that here, this whole thing is called an epoch, because we have to do one full pass through the training data. But within that epoch, we update the weights many times. Basically, we update the weights as many times as we have batches.

All right. So to go here-- let's say, for example-- basically, the idea is that you take the training set, you divide it by the batch size, and you choose the batch size. You choose the batch size. And we'll talk about, well, How do you choose that? later on. You choose the batch size.

And once you choose the size, just divide it and round it up. So for example, as you will see in the Colab, a training set is going to be 194 patients. And then we're going to choose a batch size of 32. And we typically tend to choose batch sizes of 32, 64, and things like that because it actually aligns very well with the nature of the parallel hardware we're going to use.

And so here, 32 and so on. So divide 194 by 32. You get 6-point-something. You round it up to 7. And so what that means is that the first six batches will have 32 samples each. And then the final batch has only two samples left. And that's OK. It can be a nice, little, small batch at the end. There's nothing that says that every batch has to be the same size. That's it. Epochs. Batches.

[AUDIO OUT]

AUDIENCE: And are you-- for each batch, you run through the whole network, all the layers? Or each layer is one batch?

RAMA RAMAKRISHNAN: No. For a batch, you run it through the entire network. So the way to think about it is that you take a batch. Just momentarily, you assume that's all the data you have. Just run it through the network, because unless you run it through every layer of the network, you can't get a prediction.

Unless you get a prediction, you can't calculate the loss. And unless you calculate the loss, you can't calculate the gradient. Unless you calculate gradient, you can't update the weights.

AUDIENCE: The last thing. But if you're using all the data, just doing the gradient descent, then you just go through the network once, right?

RAMA RAMAKRISHNAN: Exactly. So in gradient descent, one epoch is one pass and one weight update. And in stochastic gradient descent, the number of updates you make is equal to the number of batches you have, which ends up being the training set divided by the batch size rounded up.

AUDIENCE: Sorry, just to confirm. So initially, when you introduced the concept of batches, the whole purpose was not to run through all the data and be able to do some prediction from a subset. So now, the advantage is that after batch 1, we are using more accurate coefficient to run through batch 2 and so on? That's really the advantage of it?

RAMA RAMAKRISHNAN: Exactly.

AUDIENCE: Or is there something else here?

RAMA RAMAKRISHNAN: Perfectly said. That's exactly the advantage. So we take a small amount of data, and we say, hey, we know this is not all the data, it's just a small subset of the data. So therefore, it's not going to be super accurate. It's going to be approximate. But it's OK.

So we still tend to move in the right direction. So instead of waiting for the whole thing to get done and then updating it, we're just going to update it as we go along. All right. Yes.

AUDIENCE: Another question. Is it that doing this process, SGD, will render us a more better solution, or it requires less compute power?

RAMA RAMAKRISHNAN: Both.

AUDIENCE: Both.

RAMA RAMAKRISHNAN: Both. And the reasons for both are in the previous lecture. Yeah. And I'm saying that instead of repeating it, just because I'm very pressed for time today. That's why. All right. Cool. So that's what we have. Are we good?

OK. So now we come to the last step before we actually fire up the Colab, which is overfitting and regularization. So if you remember from your machine-learning background, when your model gets more and more complex-- use a simple model, then use a more complex model, and so on and so forth-- what happens to the error on the training data, typically? What happens to the error on the training data?

So let's say you have a simple regression model. You get some error. And then you have a regression model in which you use all kinds of interaction terms, you use logarithms, and this and that, and make it super complicated. What do you think is going to happen to the error on the training data?

AUDIENCE: Reduce.

RAMA RAMAKRISHNAN: Reduce. Basically, it's going to go down as the model gets more complex. Correct? Now, of course, comes the punchline, which is, what do you think is going to happen to the training data? I showed you the answer.

Basically, what's going to happen, typically, at least conceptually, is that it's going to get better and better. At some point, it's going to bottom-out, and it's going to start climbing again. And so we typically refer to this phenomenon here, when it starts to climb again, as overfitting because the model is essentially fitting to the idiosyncrasies of the training data as opposed to generalizing patterns.

And then, in this thing, we call it underfitting because still there's a lot of potential to improve. And we really are hoping to find the sweet spot in the middle. That's the basic idea of overfitting, underfitting. And the way we-- and to relate this to neural networks, as you see, as you've learned so far, you have to learn smart representations of the input data.

And to do that, I have argued that you need to have lots of layers in your network. The more layers you have, the better things get. GPT-3, for example, has 96 layers, if I recall. More layers the better. But more layers means more parameters. More parameters means more complexity to the model, and therefore more chance of overfitting.

So it's really important in neural networks that we think about regularization. And regularization, you will recall from your machine-learning background, is the way we handle the risk of overfitting and try to find models that fit just right. And so several regularization methods have been developed over the years. And we are going to use only two of them.

The first one is called early stopping. And this has been famously referred to by Geoff Hinton, who is one of the pioneers, or as he's more colorfully known, one of the godfathers of deep learning-- he also won the Turing a few years ago-- as a "beautiful free lunch." That's what he calls it.

So the idea is very simple. We take a validation set. We take the training data, we split it into a training and a validation set, and then we just keep doing gradient descent. Bup, bup, bup, bup, bup, bup. The training will hopefully keep on getting better and better. Lower and lower error. And then we just keep track of what's going on in the validation set. And then, at some point, if it starts to flatten out and start to climb, we just say, OK, that's when we stop training.

And what we're going to do in the Colab is actually run it through the whole thing, see where it flattens out, and then we say, OK, that's where we should stop. But of course, you don't want to go all the way to the end and then go back and say, well, I want to stop at the 10th epoch. And there are ways you can use Keras to be very efficient about this.

But the fundamental idea is you take the training data, split it into training and validation, and just track what's going on in the validation set to see whether this kind of bottoming out happens. So this is called early stopping. And the other way we're going to do-- this is called early stopping. We're looking for this part.

The other thing is called dropout. And I'm going to come back to dropout on Wednesday's lecture because that's the first time we're going to use it. And so I'll come back to dropout and tell you exactly how it works. It's a very, very clever strategy, but we will not use it today. We'll use it on Wednesday.

So in summary, what do we do? We get the data ready. We design the network-- number of hidden layers, number of neurons, and so on and so forth. We pick the right output layer. We pick the right loss function. We choose an optimizer.

As I mentioned earlier, SGD comes in lots of flavors, lots of variations on the theme. And empirically, much like for hidden-layer neurons we tend to use ReLU as the activation function, for optimization, we tend to use a flavor of SGD called Adam as the default because it's really good. So we'll use Adam, as you will see. We typically use either early stopping or dropout. And then we just fire it up and start training Keras in TensorFlow.

All right. So that is the training loop. Now I'm going to switch gears and give you a quick intro to Teras and Kensor-- Teras and KensorFlow. Keras and Tensor-- no, TensorFlow and Keras. Thank you. And then we'll actually fire up the Colab. So first of all, what's a tensor?

AUDIENCE: Yeah. Just a quick question on the previous thing. If you're looking at the validation sets to avoid overfitting, but aren't you actually overfitting because you're kind of using the validation set as a training set? Or not.

RAMA RAMAKRISHNAN: No, no, no. The validation set is never used to calculate any gradients. It's only used to calculate accuracy and loss. Yeah. It's kept aside and only used for evaluation, not for training. That's what keeps you honest. All right. And this will become clear when we actually go into the Colab. So what's a tensor? All right. OK.

AUDIENCE: Tensor is the input data, which you're giving to the system. It could be in various formats like, for instance, an image. It could be-- like we call it a 4D tensor. If it's time series data, it's 3D. And typically, if we just send numbers in, it becomes a vector, which would coincide. It gives the value of the variable as well as-- the values of the variables associated to it, as well as the information you want [INAUDIBLE].

RAMA RAMAKRISHNAN: You're kind of on the right track, but not entirely. It's actually a simpler concept than that. So--

AUDIENCE: A matrix, but generalizable with higher dimensions.

RAMA RAMAKRISHNAN: Correct. That's also actually correct, but incomplete. The reason is because it can be simpler than a matrix. It's not matrix or higher. It actually could be simpler. In fact, you take a number, it's actually a tensor. All right. The simplest case of a tensor is a number.

The next simpler case is a vector, which is a list. The next higher case is a table. So these are all tensors. So tensors, basically, are a generalization of the notion of both a number, a vector, and a table to higher dimensions.

So you can think of a tensor as having what are called-- every tensor has something called a rank. So a number is just a number. It doesn't have a dimensionality to it. So it has got rank 0. OK? While a vector, it's a list of numbers. You can write it down, top to bottom, and it's one dimension. So that dimension, that one dimension, is called a rank. So it's called rank 1.

A table is 2D, two dimensional, so it's called rank 2. And you can have a rank 3, which is just a bunch of tables. A bunch of tables is a rank 3 tensor. We also think of it as a cube. So these things are very useful because, obviously, we are all familiar with vectors. As you will see very shortly later in this class, black and white, grayscale images, are usually represented using tables of numbers, like this.

Color images are represented using three tables. Can you think of what might be representable as a tensor of rank 4, meaning, every element of a tensor of rank 4 is actually a color picture? Shout it out.

AUDIENCE: Video.

RAMA RAMAKRISHNAN: Video. Exactly. What is the video? A video is basically a stream of color images. A color video. So each element of that stream, the first dimension of the tensor is which frame it is, and then everything else is the actual frame. So the way I think about these tensors always is-- if you take a tensor, you can just think of it as a-- you can think of a tensor as being this array, which has all these axes or dimensions.

This is the first one, this is the second one, this is the third one, and so on. This is a tensor of rank 4. 1, 2, 3, 4. And so if you have a vector-- so you can imagine, if it's just a vector, you can imagine the vector actually living like this. Just a list of numbers. But if it's just-- if it is just a 2D, a rank 2 tensor, which is just like that-- which is just like that. So this thing becomes like that, and that thing becomes like that.

So for example, if this is a 7 comma 3, that means that there are 7 rows and 3 columns. So you get the idea. So the way you think about tensors is always as if there's open square bracket, a bunch of things, a closed square bracket. And that's really what a tensor object is.

So what that means is that anytime you have a tensor-- anytime you have a tensor, however complicated it is, you can always create a more complicated tensor by-- if you want to take a list of those tensors-- let's say that you have a list of videos. Each video is a rank 4 tensor, which means a list of videos is what rank?

AUDIENCE: 5.

RAMA RAMAKRISHNAN: Exactly. So a tensor of rank, say, 10, is just a list of rank 9 tensors. So that is the most important thing you need to understand about tensors. So at any point in time, if I give you a tensor, you can just iterate through the first dimension of it, the first aspect of it. And as you go through each one of these values-- so for example, here, yeah, that can do it.

So if you have this tensor here, and if you want to create a more complicated tensor, no problem. So you add another dimension here. And now it just becomes-- this dimension, let's say, has 9 values. 0, 1, all the way to 9. So you put 0 here. And then what do you get? This whole tensor is a rank 4 tensor.

And you put a 1 here. It's another rank 4 tensor. You put a 2 here. Another rank 4 tensor. So every tensor, you take the first element. It's just a list, but it's a list of the next down-rank tensor.

Now, this tensor concept is actually something Einstein came up with. And so it's simultaneously kind of easy to understand and also slippery. So I would actually encourage you to read the book, which has a really good discussion of tensors. And the more you practice with it, the easier it'll get.

So if you feel you kind of understood but not quite, you're not alone. It happens to all of us. You have to pay the price and go through the crucible. OK? All right.

So to come back to this, that's what we have. And we already talked about a rank 4 tensor. It's a video. So 2.2, the text has a lot more detail. You should definitely read it. So here, TensorFlow is a library. And as you can imagine, neural networks, tensors come in and go through the network and go out the other end.

And since tensors capture everything-- numbers, lists, tables, and so on and so forth-- it's just tensors flowing from input to output. Hence, it's called TensorFlow. And it gives you a couple of things which are really, really important, which is why we use it. The first one is that it will automatically calculate gradients for you of arbitrarily complicated loss functions.

You don't have to calculate the gradient because calculating the gradient is very painful. It will automatically calculate the gradients for you. That's the best part. You don't have to use the chain rule. You don't have to do anything.

The second thing it will do, it gives you all these optimizers, including SGD and all its variations, so you don't have to worry about the optimization itself. You can just pick and choose what you want. Third, if you have a lot of servers, it will actually take the computational load and distribute it across all those servers.

People here with a CS background know that parallelizing computation is actually a very difficult problem. There are things which are called embarrassingly parallel. Many things are not. And it's actually quite tricky to figure it out. You don't have to figure it out. TensorFlow will figure it out.

And then finally, I talked about the fact that there are these things called GPUs, Graphics Processing Units, which are parallel hardware. And so, even if you have just one computer but it has GPUs, there's a particular way in which you have to take your computation and organize it to really exploit the fact that you have a GPU.

And so TensorFlow will actually do it for you, out of the box, automatically. You don't have to worry about any of that stuff. So those are all the advantages of this thing.

By the way TPU is called a Tensor Processing Unit. It's something that-- you can think of it as Google's GPU. They came up with their own variation on the theme.

Now, Keras sits on top of TensorFlow. This is the hardware you have. TensorFlow sits on top of the hardware. Keras sits on top of TensorFlow, and it basically gives you a whole bunch of convenience features. So for example, it gives you the notion of a layer.

We already saw, keras.dense is a dense layer. It gives you the notion of a layer, it gives you the notion of activation functions, and so on and so forth. It gives you easy ways to preprocess the data, easy ways to train the model, report on metrics, calculate validation loss, validation accuracy, training loss, all the metrics we care about.

And then it also gives you a whole library of pre-trained models that you can just use and adapt for your particular problem. So it gives you a whole bunch of conveniences. And that's why it's very popular.

And by the way, many of you might also be familiar with PyTorch, which is a fantastic framework as well for deep learning. And the reason we chose to go with TensorFlow for this course, rather than PyTorch, is because we wanted to make the course accessible to folks who don't have a ton of programming background before coming to the class, and PyTorch is a bit more demanding from a CS perspective.

It requires more knowledge of object-oriented programming, which is why we decided to go with TensorFlow and Keras, because I think it's actually as powerful, in many ways, and it's a little easier to get going. So that's what we have here.

And one other thing I will mention is that there are three ways in which you can use Keras. There are three kinds of APIs-- sequential, functional, subclassing. And we'll almost exclusively use the functional API. And in fact, the model we built for heart disease prediction uses the functional API. And so just read 7.2.2 of the textbook to understand in detail how the API works.

I find in my own work, the functional API is basically all I need. I don't need to do anything more complicated than that. And as you will see as you work on the homeworks and on your project, it's sort of a beautifully designed LEGO-block environment for doing these things. And you can create very complicated models very easily. There is a whole bunch of stuff here on these websites, so check them out. Lots of Colabs are available.

So now, if you go back to the neural model for heart disease prediction, this is what we came up with in the last class. We had an input layer, one dense layer with 16 neurons, ReLU neurons, an output layer with a sigmoid, and then, boom, that was a model. So let's train this model.

And so the training checklist is that we have already done this. Hidden layer of 16 neurons. Sigmoid. We need to use an appropriate loss function based on the type of the output. What loss function should we use? What is the output here?

It's a binary classification problem. So what should the loss function be? I kind of heard it somewhere. You can shout it out.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: The output is a sigmoid. The loss function.

AUDIENCE: Binary cross-entropy.

RAMA RAMAKRISHNAN: Binary cross-entropy. Remember, if you're predicting a number, an arbitrary number, you can use something like mean squared error. If you're predicting a probability, which has to be compared to a 0, 1 output, which is what binary classification is all about, we use binary cross-entropy.

So that's what we do here. So we do binary cross-entropy. And then we will go with Adam. And then we will use early stopping to make sure we don't overfit. I know this-- OK, I promise, this is literally the last slide before I go to the Colab.

I feel like one of those used car salesmen. Wait, there is more! So anyway. So don't worry if you don't understand every detail of what I'm going to go through. I'm going to link to the Colab as soon as the class is over. But once you get your hands on the Colab, make sure you actually go through every line in Colab.

What I typically do when I'm trying to learn something new is I'll actually cut and paste. I won't do that. I won't actually cut and paste the code and run it myself. I will retype the code. If you retype the code as opposed to cutting and pasting, trust me, you will learn a lot more. So I strongly encourage you to do it that way.

And so all the Colabs you're going to publish in the class, the first thing you should do is you should just make your own copy of the notebook. Copy to Drive. And then, if you're using anything other than today's Colab, anything involving natural language processing or vision, you probably should use a GPU.

So just go into-- you go in here, choose the runtime to be GPU, and then you start your notebook, and you're done. And the second time onwards, you can just go directly to this step. You don't have to do all this stuff for that particular notebook. There are numerous tutorials, like five-minute videos and so on, on how to use Colab. Just do that. I'm not going to spend time on it here.

All right. OK. So I just ran it a few hours ago. I'm not going to run every cell now because this is going to take some time. It's going to get in the way of the class time. But I'm going to just go through it slowly and explain what's going on.

So here, this is just an introduction to the data set. We already saw this introduction last week. We have whatever-- 303 patients. Heart patients. We have a whole bunch of variables here-- age, demographics, and a whole bunch of biomarker information. And this is our target variable. 0 or 1. Heart disease, yes or no.

And so, by the way, just some technical preliminaries here, basically every time we load these things, we're actually going to load these packages. So you can see here, these are the two key things we need to do. We import TensorFlow first. And then, from within TensorFlow, we import Keras. That's what these two lines do here.

And folks who have done data science and machine learning a bit before, you'll know this. We will actually load the three packages that were just most commonly used right, which is NumPy, Pandas, and Matplotlib.

NumPy, because it's very easy for manipulating matrices and arrays and tensors. Pandas because, oftentimes, you get some data from somewhere, you need to massage it and wrangle it to a point where you can actually feed it into Keras. So you need pandas for that. And Matplotlib because you just want to plot these loss curves and accuracy curves to see whether early stopping is needed. So that's why we use it. So we import all these things.

And then, I guess the other thing you have to remember is that when we are training these deep-learning models, there is randomness in the process, which enters in a few different places. So clearly, the starting values for these weights are going to be-- the weights are going to be randomly initialized. And therefore, that's obviously a source of randomness.

Now, we talked about how you take-- when you're doing stochastic gradient descent, you take all the data, and then you randomly choose batches right from this data till we finish a whole pass through it. Well, that immediately raised the question, well, what do you mean by randomly choose? So typically, what we do in practice is that-- and Keras will take care of all this for you-- you basically take the data and just shuffle it once, randomly.

And then you just go first 32, next 32, next 32, next 32, like that. But it is a source of randomness. And then when we split the data into train, validation, testing, and so on, particularly if you want to look for early stopping and overfitting, we need to again split the data randomly. And that's another source of randomness.

And then when we do dropout, which we will talk about on Wednesday-- again, dropout has a little bit of a random element to it, and so that's another source of randomness. So all this means is that if you're working with these models and if you want to build a model and you want to hand it off to someone so that they can reproduce your results, well, you better make sure that you make it easy for them to replicate what you have.

And the way you do it is by sending a random seed for all these things. And the way you do it is by having this little handy function here, set_random_seed. And of course, I use 42, just like everybody should. So that's that. By the way, that's just a pop culture reference to this book called *The Hitchhiker's Guide to the Galaxy.* So when I say something about the number 42, and you'll know what I mean.

So by the way, the question inevitably comes at this point, OK, if we do exactly this, will we actually get the exact same numbers that you have in your version of the notebook? And the answer is, hopefully, most of the time, but it's not guaranteed. So this is called bitwise reproducibility.

It's not guaranteed due to certain hardware things and device drivers and stuff like that, so we won't get into all that stuff, which is why, as you see here, I have a bit of a fingers-crossed thing. All right. Cool. So that's what we have.

So as it turns out, Francois Chollet, who wrote the book, the textbook, he actually made this data available in a Pandas dataframe. So we read the CSV file into this dataframe right there. And then it's 303 rows, 14 columns. And you can see here, we'll take a look at the first few rows. And these are all the rows-- age, gender, cholesterol, blah, blah, blah, blah, blah. And then this is the target variable right there.

And one of the first things I always do when I'm working with a binary classification problem is to quickly check whether the positive and negative classes are balanced or not. And so what you can do is you can just quickly check to see what percent of the data points is 0 versus 1.

And you can see here, 72.6% of the patients don't have heart disease. That's a good thing, of course. And then 27.4 have heart disease. So it's not 50/50 or roughly 50/50. It's a little thin.

So by the way, quick question. What is a good baseline model for this problem? Suppose you couldn't use anything, any complicated thing. What's a good baseline model? Let's go to this other side.

AUDIENCE: Yes. Just predict zero every time.

RAMA RAMAKRISHNAN: Why would you do that?

AUDIENCE: It would give you a 72.6% accuracy.

RAMA RAMAKRISHNAN: Exactly. Because 72.6% is the higher class, higher class of the higher percentage, you just predict it. You'll be right on those 72.6% of the cases, you'll be wrong on the rest, which means that your accuracy of this model is going to be 72.6%. And so any fancy model we build, it's got to do better than this. Otherwise, it's not worth its weight in layers.

All right. So we'll come back to this later. So the first thing we want to do is we want to preprocess it, because this data set has both categorical variables and numeric variables. And so it's usually convenient just to group them into two different groups. So I have listed all the categorical variables here and the numerics here.

And then we have the preprocessing here. We have to take the categorical variables, and we have to one-hot encode them. And the reason is that unlike, say, a decision tree model, a neural network cannot handle categorical inputs directly. It can only handle numeric inputs, which means that we have to numericalize every categorical thing that comes in.

And there are many ways to do it, but the standard way to do it is one-hot encoding. And for the numeric variables, we need to normalize them. And I'll come to that in a second. So Pandas has this get_dummies function here. And you can just run this thing, and it will just one-hot encode the whole thing.

So once you do that, this is what you have. So you can see here, previously, let's say, thal had three values-- fixed, normal, reversible, or something. And then you go to the one-hot encoded version. And now we can see here, thal_fixed, thal_normal, thal_reversible. That's three columns. That's the one-hot encoding in action.

Now, the other thing to remember is that neural networks work best when the numeric inputs you send them are all in a relatively small range. They shouldn't have a wide range of variation. And so the standard practice is to standardize the numerical variables. By standardize, I mean typically subtract the mean, divide by the standard deviation.

We should do that. But before we do so, we should split the data into a training set and a test set. And why do we want to split into a test set? Because at the very end, once we have built the model and done all the things we want to do with it, we finally want to take out the test set and evaluate it once so that we get this true measure of how it's going to perform in the wild after you deploy it.

So you want to divide it, let's say, 80% training and 20% test set. So the question is, why should we do the splitting now before we do the normalization? Why can't we just do the normalization and then do splitting? All right.

AUDIENCE: Because then your validation set is also somewhat dependent on your test set results, as well as the mean of the test set.

RAMA RAMAKRISHNAN: Correct. Because the test set has now essentially been influenced by the training set. The modeling process-- part of the modeling process is the splitting. And the splitting also-- the standardization-- if the standardization, which is part of the process, uses information about the test set, well, the test set is not really kept away from anything, is it?

That's why we want to split it, lock away the test set somewhere, and then proceed with the modeling. Again, this is like machine learning 101, which is why I'm going through it pretty fast.

So we do this sampling function, take 20% of the data, and make it the test set. And the remaining is going to be the training set. And when we do that, you can see the training set is now 242 rows, while the test is 61 rows. And any of these data frames, you'll know the shape attribute gives you the dimensions of the number of rows and the columns. That's what we're doing here.

And now that we have done that, we have done the split, we can calculate the mean and the standard deviation. So I calculate the mean here. I calculate the standard deviation. These are all the means. And once I do that, I just do each column minus the mean, divide the standard deviation.

And then once I do that, I save them in the train and the test dataframes. And you can see here now all the numbers are all very smallish-- 0, 1, minus 1, around that range. And that's kind of ideal for neural network training. OK?

All right. So at this point, the data is entirely numeric. And then we are almost ready to feed it into Keras. And the way you do it is you take a NumPy array-- you take a Pandas dataframe, and then you convert it into a NumPy array. And then Keras is happy to take it, happy to receive it.

So we use this thing called to_numpy, which I think is as descriptive as it gets in programming. And then you save it as train and test. Now, train and test are two NumPy arrays with exactly the same information, and now we can feed it into Keras.

All right. Now I guess there's one other thing we need to do, which is that in this dataframe, train and test are independent variables. All the features, as well as the target, the 0, 1 target, they're all in this. And we need to now take it and just take the dependent variable, the 0, 1 column, and split it out, and keep the x and the y separately.

That's the whole point of it. Because you need to feed the x, do the prediction, and then compare it to the actual y, and calculate the loss, and so on and so forth. So the target column is our y variable. And it's column number 6 from the left. If you count it, you can see it.

So we delete it from the train and test. And now we have 242 rows and 29 columns, 29 features. You will recall from the network that we made way back, it had 29 inputs, 29 nodes in the input layer. And that's where the 29 is coming from.

And so now we just select the sixth column, which is the target, and make it the y variable, train_y and test_y. And that is, of course, a vector, which is 242 long in the training set and 61 long in the thing. So at this point, all we have done is, to be honest, boring preprocessing. We haven't actually gotten to the action yet.

Finally, let's do something. And we start with a single hidden layer. Since it's a binary classification problem, we'll use sigmoids, as we saw earlier. And this is the model we created in class, last class. This is the model we created.

The only difference between that model and this model is that I've actually given names to these layers. And this name thing is totally optional. If you want to give a name, give a name. It's just a little easier to interpret later on. It's just cosmetic. But I've just put it here.

And once you build the model, you should immediately run the model.summary command because it gives you a nice overview of the model. For each layer, it tells you what the layer is, it tells you what's coming into the layer, meaning the shape of the tensor that's coming in and what's going out, and how many parameters the layer has. And it turns out, this layer has-- sorry, this network has 497 parameters.

And I have told you repeatedly, the first few times, just hand-calculate the number of parameters to make sure it verifies. So we should just make sure that it is, in fact, 497. So let's hand-calculate it. And you do basically-- it's basically what's going on here. 29 inputs times 16. All the arrows. 29 times 16 arrows.

And then you have a bias of another 16. That's why you have this expression. And then the next one is 16 times 1 plus 1 bias for the output sigmoid. And you get to 497. OK. Just make sure you follow this later on when you work with the Colab. We did this in class last week.

And you can visualize the network graphically as well by using the plot_model function. So we do that here. And let's say, it gives you the same information, but in a slightly easier form to consume. And when we work with larger networks, starting on Wednesday, you will see that being able to visualize the topology of the network is actually quite handy.

OK. We finally come to actually trying to train this thing. And so what loss function should we use? We need to use binary_crossentropy, right there. What optimizer to use? Well, as I mentioned earlier, we will use Adam. Adam. All right. Adam.

And then the final thing is, you can ask Keras to report out whatever metrics you care about. These metrics are not going to be used in any optimization. It's just reporting it to you. And the most common thing people report out for binary classification is accuracy. So we'll just go with that metric.

And so what we do is we tell Keras, take the model we just built and compile it with this choice of optimizer, this choice of loss function, and these metrics. And this compilation step, what it does is, essentially, Keras will take this information and take the model you've built, and it will reorganize the model in such a way that the parallel computing, distribution of computing across many servers, and so on, that's what's happening in the compile step.

Organizing it so that-- reorganizing the model so that it becomes amenable to parallelization and distribution, that's what's going on. That's why you actually have to do something called a compile step. And once we do that, we are finally ready to train the model. And to do that, we have to decide what the batch size is that we're going to use.

Remember, we're using some flavor of SGD, which means we have to choose what is the batch size. And typically what people do is that 32 is a good default for batch size. If you're just getting started with something, just use 32. And there's a whole bunch of literature on what the right batch size should be for the number of data points you have, the size of the network, and so on and so forth.

My philosophy is start with 32. And you can always try 32, 64, 128. It's kind of like-- oftentimes what people tell me, researchers tell me, is that just use the biggest batch size that doesn't make your machine die. If you can fit into memory, it's probably good. Just try the bigger sizes. We'll just start with 32. It's a tiny problem. It's not a big deal.

And then we also have to decide, how many epochs through the data do we want to go through? How many epochs? And usually, 20 to 30 epochs is a good starting point. And then, because this is a tiny problem, just for kicks, I just had to run it for 300 epochs, just to see if any overfitting is going to happen.

And then whether we want to use a validation set. Of course, we want to use a validation set. So we will use 20% of the data points as a validation set so that we can look for overfitting and underfitting.

All right. So with these decisions made, we finally-- we use the model.fit command. model.fit is what actually trains the neural network. And you have to tell it what the x tensor is. You have to tell it what the dependent variable y tensor is. We need to tell it how many epochs to do this, what the batch size to use.

verbose=1 just means just put a lot of descriptive output as you do this thing. And then validation_split means, take 20% of the training data and set it aside as your validation data set. Don't use it for training because I want to measure overfitting using that.

So that's it. So you do that thing. It will run for 300 epochs. And this is the reason why I decided to just not actually run it in class. And so you keep on doing it. It gives you a lot of output. And finally, we reach the end.

OK. Now, let's take a moment to understand what's being reported. So I'll just take this one line here. So these two-- There is a pair of lines for each epoch. And then here it's telling you-- it actually uses-- in this 300th epoch, it used 7 batches. 7 out of 7 batches. So it used 7 batches.

And you will recall, from the math we did in the class, that it's actually 7 batches, where the first six batches are 32, and the last batch is just a couple of examples. But we have 7 batches. This is the 193 by 32 rounded up. So that's why you have 7 here.

And then it tells you how long it took it for that. And then this is the loss value. This is the binary cross-entropy loss value on the training set on that particular batch that it calculated. This is the accuracy that you asked it to report out. 98.5% accuracy on that batch.

And then, at the end of this epoch, using whatever weights were available in that network, you'd actually calculate the loss on the validation set, which is the 20% the data we have set aside. And then this is the accuracy on the validation set.

So that's what each of these numbers mean. Now, looking at this wall of numbers is kind of painful. So usually you just plot it. And the way you do that is if you notice here-- OK, I'm not going to go back here. So I said history equals model.fit, blah, blah, blah, blah, blah. And that history object has a lot of information that we can use for plotting and diagnostics and so on.

And that history thing, history object, has another object called history, history.history, which is a dictionary with all these values. And that's what we're going to plot. Was there a question here? Yeah.

AUDIENCE: So you prompted it to keep 20% aside for validation. But didn't we already do a test set? So that's going to be a secondary validation.

RAMA RAMAKRISHNAN: Right. So basically, we have a training and then a validation and a test. The role of the validation set is to figure out things like early stopping. Should we stop here? Should we go back? And as you will see later on, if we use hyperparameters, we will try different values of the hyperparameters and figure out-- use the validation set to figure out which one is the best one.

But once we are done with all that, we will finally have a model. At that point, we open the safe, take out the test set, and use it just once with your final, final model. Not because you want to improve the model, but because you want to have a realistic idea of how it will do when you actually deploy it out in the real world. Yeah.

AUDIENCE: Can we use-- instead of accuracy, can be used other metrics to evaluate whether to--

RAMA RAMAKRISHNAN: Absolutely.

AUDIENCE: --actually [INAUDIBLE] on the confusion matrix, let's say?

RAMA RAMAKRISHNAN: Yeah. You can do whatever you want. You can use-- like I said, it's not used for training. So there is no mathematical implication of what you choose. You can choose error rates, accuracy, F1 F beta. You can do whatever you want. And Keras, as you will see, has this dizzying list of possible metrics you can use for reporting.

The key thing to remember is that you're just reporting these metrics. You're not actually using them for any training. Yeah.

AUDIENCE: My question is with respect to validation. Like, we've got a training data set. So when we take out 20% of data from validation, are we taking out from the training set or--

RAMA RAMAKRISHNAN: Correct.

AUDIENCE: From there, that level? Or we go to each batch and take out 20% from each?

RAMA RAMAKRISHNAN: We're taking out from the training set.

AUDIENCE: So it means the batch size-- the number of data would be available for calculating the batch size would reduce?

RAMA RAMAKRISHNAN: Correct. And in fact, once we take out the validation set, whatever remaining is 193.

AUDIENCE: And then we divide that into batches. And then every epoch, the validation and the data gets different.

RAMA RAMAKRISHNAN: No. Once you take out the validation set at the very beginning, you keep it aside, and then you only evaluate at the end of each epoch what your loss and accuracy is on that validation set.

AUDIENCE: So you would have cross-validation.

RAMA RAMAKRISHNAN: No, no, we're not doing any of that stuff. We're just taking it out once, and we're just evaluating the end of every epoch. OK. So yeah.

AUDIENCE: So I know we both asked similar questions- but--

RAMA RAMAKRISHNAN: Use the microphone, please.

AUDIENCE: I know both of us asked similar questions, but just to reconfirm. So here, my training model is giving me, say, a loss of 0.0860. My validation model is giving me 0.660. That means I have already crossed the int. So when I have to actually test the model, that is the midpoint which I did, that will be the model which will get deployed in production.

RAMA RAMAKRISHNAN: Correct. And as to, OK, what do we do to get that model-- do we actually have to go back to the beginning and run it for a few epochs, or can we do something smarter than that?-- we'll get to that. Yeah.

AUDIENCE: Isn't that validation set different for each epoch, or is it the same?

RAMA RAMAKRISHNAN: It's the same. So what you do is you have a training set. Before you do any training, you take out 20%. Keep it aside. You take whatever is left over. You divide that into minibatches and then start running it through each epochs. But at the end of each epoch, you just evaluate the quality of that resulting model using the validation set.

AUDIENCE: What's different between each epoch? Is it just the weight--

RAMA RAMAKRISHNAN: The weights have changed.

AUDIENCE: It's the division into the different--

RAMA RAMAKRISHNAN: No. So the difference in each epoch is the weights have changed. So after every minibatch, the weights have changed. At the end of one epoch, you've gone through all the data points you ever had in the training set. And then you come back to the beginning, and you do it again.

AUDIENCE: How do you identify the sweet spot?

RAMA RAMAKRISHNAN: It's coming. Yeah. All right. So I'm going to keep going. So we have this here. And so you just-- I mean, there's a little bit of Matplotlib code. So what we do is we just plot the training loss and the validation loss as a function of the number of epochs.

And as you can see here, the training loss is these things here. And it's steadily going down, as you would expect. The validation loss goes down here. And then, at some point, it kind of flattens out, and then maybe gently starts to rise.

So do you think there's overfitting? There seems to be some level of overfitting here. But the thing you have to always remember is that the binary cross-entropy loss is a loss function that is convenient for you because it sort of captures the thing you want to capture, the discrepancy, but also because it's mathematically convenient.

But what you may actually care about in practice is something like accuracy. So I always-- that's why you're reporting out the accuracy when we do these things. So you should also plot the accuracy to see what's going on. And really, you should look at the accuracy and figure out overfitting and underfitting and all that stuff. So let's just do that.

So I have here overfitting. So this is how it looks like for accuracy. Accuracy, of course, as the model gets-- as you do more and more epochs, hopefully it gets better and better for training. So you can see here accuracy actually climbs all the way up to the mid 90s right there. The low 90s here. The validation gets to this point after, I don't know, 50 epochs maybe. And then it kind of flattens out. And then, strangely, it climbs up again a bit later.

Now, the fact that the accuracy actually got better at the very end suggests that maybe we can live with this overfitting. It's OK. It's not the end of the world. So you can certainly-- what you can do is you can go back and say, you know what? No, I'm going to be a purist about this.

Around 50 epochs or so. I think that's when it actually flattened out for loss. So you can just go back and just restart the model and run it only for 50 epochs, not 300. And then stop and just use that model for everything from that point on. Or you can say, you know what? It's OK. I can live with this thing.

And so that's what we're going to do here. Let me just stop for a second. There was the question. Yeah.

AUDIENCE: Originally, when we were starting out, we were saying 20 to 30 epochs, but we're going to be doing 300. 50 is over 20 to 30. So when it comes to validation of if you've run enough epochs, are you doing a derivative calculation?

RAMA RAMAKRISHNAN: Oh, I see. No, that's a great question. So the question is, I said start with 20 and 30 epochs as a rule of thumb. Here, I'm just going with 300. And because I'm going with 300, I can actually see some potential evidence of overfitting. But if I had done only 20 to 30, maybe I wouldn't have even seen that.

What happens next? Is that the question? Great question. So what you should do is when you look at these curves, if, at the end of 30 epochs, you find that the validation loss continues to drop, then you know maybe there is more room for it to drop. So you continue from that point on.

The thing about Keras is that you can actually run the fit command at that point, and it will continue where it left off. It won't go to the beginning again. So you can run 10. OK. The validation is still getting better and better. OK. Run for another 10. It's getting better and better. Run for another 10. Getting better and better. Run for another 10. Oh, it starts to climb up again. OK, now I'm going to back off. That's what you do.

All right. Now, all this manual stuff, I'm going through it just because-- to build intuition. There are these things called callbacks in Keras, which we'll get to later on, in which you can actually tell it, hey, when the validation loss stops improving, stop everything, or when it stops improving, save that model for me somewhere. So they don't have to go back and rerun everything. It will have saved it for you, and you can just pick it up and use it. Yeah.

AUDIENCE: What's the intuition behind the accuracy continuing to improve when the loss is getting higher?

RAMA RAMAKRISHNAN: Because accuracy and loss are related, but they're not the same thing. So it's a really good question. It's also kind of a profound question because accuracy is a very discrete measure. So if a particular point, we're predicting its probability to be, say, 0.49, we're going to say, OK, that's a 0, no heart disease. But if it goes to 0.51, we're going to be like, oh, that's heart disease.

So when you go from 0.49 to 0.51, the binary cross-entropy loss will change very, very slightly. Well, the accuracy will go from 0 to 1. Dramatic jump. So it's very jumpy and discrete. And that's why it tends to be a proxy but a crude proxy for loss. So that's part of the reason. And I can talk more offline. OK. So-- yeah.

AUDIENCE: You mentioned that if you are curious, you could stop at epoch 50 in this case and run it and stop it there. I was wondering, if you could see the history of the model, take the weights at epoch 50, and include it in your model, if it would be roughly the same, or it would be certain differences?

RAMA RAMAKRISHNAN: You could try it. Yeah, you should just try it. Because what happens is that, ultimately, what we care about is how it performs on the validation set. Here, it appears to perform better on the validation set if you stop at 50, but only for the loss. For accuracy, actually, if you wait till the very end, it gets better.

So my thrust tends to be, what is the measure that's closest to the real-world deployment? It's accuracy. So I tend to go with accuracy. Binary cross-entropy is a beautiful proxy, but an imperfect proxy for the thing we actually care about in the real world, which is error rate and accuracy. That's why I tend to want both. And if accuracy is telling me one thing, I kind of tend to believe that.

All right. So here is what we have. So once we do all this we have a model. And we know we want to evaluate to see, OK, if we actually deploy it, how good is it going to be? So you use this thing called the model.evaluate function. So you take the model.evaluate function. Now you use the test and the test_x and the test_y data set, which we split at the very, very beginning and never used from that point on.

We run it. And when I ran it last night, it came up with an 83.6% accuracy for the model. And remember, our baseline model, which just predicts everybody's a 0, is going to have a 72.6% accuracy. And this little neural network gives you 83.6%, which is pretty good. So it's actually-- phew, it's beating the model, the baseline model, which is nice.

And I guess there is something here about the fact that we did a bunch of preprocessing outside Keras, and then we send stuff into Keras. You can actually do all this preprocessing inside Keras automatically. And there are layers for that. And I have linked to a bunch of stuff here.

So that's it as far as this model is concerned. I know we went through it really fast, but please go through it afterwards and make sure you understand every single line. Change each of these lines, rerun it, see how the output changes. That's how you build some intuition. OK. All right. Computer vision. As I do.

AUDIENCE: Just one question. Is there a way to build the model just to have less false positives or less false negative? Or we don't know that?

RAMA RAMAKRISHNAN: Oh, yeah, you can do that. But there are-- so you can report on all those things very easily. But there are more complex loss functions which will take the asymmetry between the false positive/false negative into account. Yeah. So the short answer-- it's possible. Yeah.

All right. So first, let's just talk about, how do you represent an image digitally? And so these are our grayscale images represented, black and white images. So the basic idea is very simple. Every picture you have, every location in that picture is a pixel. And a pixel basically has a light intensity-- the amount of light at that location.

And that light level is measured from 0, no light, to blinding white light, which is 255. And so all the numbers here-- if you take this 5, for example, you can see a lot of no light. All the black regions, those are all zeros. And then wherever there is white light, there is a number. And more the amount of light, the closer it gets to 255.

In fact, if you just step back and squint at this, you can actually see the 5. So that's it. That's how black and white images are represented. Very simple. Now-- yeah.

AUDIENCE: Can you see [INAUDIBLE]--

RAMA RAMAKRISHNAN: Microphone, please.

AUDIENCE: Just when you say amount of light, what's the unit that's being measured? What do you mean?

RAMA RAMAKRISHNAN: So here, basically what we have is the computer takes whatever-- so when you send an analog-- you take an analog picture. There is a process by which you take that analog picture and read it in, and it gets mapped to a scale between 0 and 255. That's it. That's all.

So you can think of it as like a relative scale, a normalized scale, between 0 and 255. And so it just roughly maps to the amount of light in that location. The exact lumens to the number mapping, I don't know how they do it. My guess is there are a dizzying number of variations on that. But for our purposes, just think of it as, it's a normalized scale which runs from 0 to 255.

All right. So if you look at-- so that's what's happening. Every is a number between 0 to 255. Boom, boom, boom. So if you have a color image, each pixel of a color image is represented by three numbers, and these numbers measure the intensity of red light, blue light, and green light. Because red, blue, and green, if you mix them in the right proportion, you can get whatever you want.

And so each light density is still a number between 0 and 255, and that's what you have, which means that now you have three tables of numbers instead of one table of numbers. And by the way, just some lingo here, in the deep-learning world, these colors, RGB, Red, Blue, Green, are sometimes referred to as channels.

All right. So this is what we have here. This is a picture of Killian Court. And then if you take that little thing here, the red table, the green table, and the blue table. So for this picture, these three tables is a tensor of rank what? Good. All right. Any questions on this?

So the key task in computer vision-- obviously, the important thing is image classification. The most basic task, if you will, when you're working with images is you have an image and you want to take whatever-- you take the image and figure out, OK, you have a list of possible objects the image could contain. And you're figuring out, OK, which of these possible objects exists in that image?

The Dog/Cat classification is the canonical example that we all know and love. And that's what we will solve later today and on Wednesday. But there are many other tasks that you need to be aware of. So then you actually not just classify an image, but you also localize where in the image is it.

It's not just enough to say sheep. You want to figure out, where is the sheep? And that's called localization. And the way you do localization is you put this little box around it. And then you output not just whether it's a sheep, yes or no, but the coordinates of this box.

The top left and the bottom right, for example. If you put the coordinates, you can actually draw a box around it. So you output the numbers, the coordinates, of where this box is in the picture. This is called localization.

Now, this is object detection where you may have lots of objects going on, and you want to pick up every one of them, and you want to localize it. This is object detection. So here, we have gone in there and said, OK, sheep 1, sheep 2, sheep 3. And each of these sheep has a little box around it.

By the way, self-driving cars, the camera vision system is constantly scanning what's coming in through the cameras and doing object detection constantly, many times a second. Pedestrian box, zebra crossing box, doggy box, stroller box, and so on and so forth.

And then we have this thing called semantic segmentation, where we take every pixel in the picture and classify every pixel. We're not classifying the whole picture. We're classifying every pixel. So we are saying, OK, all these gray pixels, road. All these pixels are sheep, and all these pixels are grass.

Every pixel is being classified. So we are taking an image and, instead of giving one classification, for every pixel, we are solving a multi-class classification problem. Every pixel is classified.

And just when you think it can't get more complicated than this, we have something called instance segmentation, where not only are we classifying every pixel, we are distinguishing between the different sheep. So every pixel is classified, and different instances of the same category need to be identified.

So these are all some of the most-- I won't say popular, most prevalently and useful-- most prevalent and useful categories of image-processing problems that are amenable to a deep-learning system. All right.

So let's go to image classification. And we're going to work with this application called fashion-mnist. So the idea here is that you have 70,000 images of clothing items across 10 categories, like boots and sweaters and T-shirts. You get the idea. 10 categories of clothing.

We have 70,000 images like this. And then we'll build a network from scratch to classify all these things, with pretty high accuracy. So these classes-- by the way, this is a very balanced data set. So 10% of the data is sweaters, 10% is boots, and so on and so forth. So a naive baseline model would give you what accuracy?

10%. Exactly. So we need to build something that's better than 10%. And I'm glad to report that a simple neural network can actually get you close to 90% on this [INAUDIBLE]. All right.

So this is the simple network that we have. The input in this case is a 28 by 28 picture. It's a 28 by 28 picture. And so far, we have been feeding vectors into our neural network. Now we have a picture, which is 28 by 28. It's a tensor of rank 2. It's a table of numbers. What do we do? How do we feed that in?

It's a-- no. Each image is a table of numbers. Let's just take a single image. What do we do? What do we do with this table? Convert it into a vector. Exactly. And that's called flattening. So we take this table of numbers, and we flatten it into a vector.

And so what we do is-- let me just-- OK. So we have 28 by 28. So what we can do is we can take each row. We can take this row and then write it like that. We take the second row. Oops. I'm going to write it like that.

Third row is here. I'm going to write it like that. You get the idea. So you take each row, just rotate it, and stack it all up. And string them up. It becomes one long vector. So this is called flattening. So that's how you take this thing and make it into one long vector.

So when you do that, 28 by 28 is? What is it? 784. So we get 7-- so we get a vector. This is the flattened input. And you get 784. It's a vector that's 784 long after the flattening. We have not done anything complicated yet. We've literally taken the numbers and just reorganized them in a different way.

And once we do that, now we are back in our familiar neural network territory. We know how to work with vectors. So we just need to pass it through a hidden layer. And this hidden layer, we're going to use ReLU neurons. And I tried a few different values. And it turns out that 256 neurons does a really good job.

And so I'm going to use 256 neurons here. And then we need to now think about what the output layer should be. Now we run into a problem because the output layer-- before we saw for the heart disease example it's just 0 or 1. Here, there are 10 possible outputs. It could be a boot, a sweater, a shirt, and so on and so forth. 10 possible categories.

So we need some way to handle something with more than one binary output, many possible outputs. So the way we do that-- by the way, pay attention to this, because this is actually how GPT-4 works. So what we do is, here's what we have. We know how to output 10 numbers. If you want to output 10 numbers, no problem. We can easily output 10 numbers by just using a linear activation.

We also know how to output 10 probabilities. Each one just needs to be a sigmoid. But here, we can't use 10 sigmoids as the output. Why is that? Why can't we use 10 sigmoids?

AUDIENCE: Because the probabilities wouldn't sum to 1.

RAMA RAMAKRISHNAN: Correct. So here, when the output comes, we need to figure out, OK, is it a boot, a sweater, a shirt, and so on and so forth? There is only one right answer. Which means that we need to actually figure out which of these 10 is the right answer, which means that we need to produce probabilities, but they have to add up to 1 because only one of them can be true.

So that's the key thing. They have to add up to 1. That's the wrinkle. If not for that, we can just use 10 sigmoids. And the way we do that is using something called the softmax function or the softmax layer. And the idea is actually very simple. We have these 10 outputs in the very final layer, which is just linear activations. And then we take each one of these numbers, and then run it through the exponential function, and then divide by the total.

So when you do that, two things happen. The first one is when you take these numbers and run it-- say you take a1 and do e raised to a1, you now get a positive number. And now you have a positive number divided by the sum of a bunch of positive numbers.

And they're all-- you can see here, you can confirm visually that they will add up to 1 because you're literally taking each number, dividing by the total. So they will add up to 1. There's no other option. So this is called the softmax function, which means that you can take any set of 10 numbers that's coming out of the network and convert them into probabilities that add up to 1.

And so, by the way, the GPT-4 reference, when you actually put a prompt into GPT-4 and it starts giving you the output, every word it's emitting-- it's actually a token, but we'll get to that later. Imagine it's a word. Every word it's emitting, it's doing a 52,000-way softmax.

Think of it as every word in the language is a possible output. So it's a vector, which is 52,000 long, but it's actually a softmax. And it just picks the most probable word and emits that. So this notion of a softmax is actually very powerful. But we'll come back to that later.

So to summarize, if you have a single number, you can use a simple output layer, a single probability, a sigmoid. You have lots of numbers. Just have a stack of these things. And when you have a lot of numbers that have to add up to 1 that have to be probabilities, use softmax. OK. So. Yeah?

AUDIENCE: Why do we choose probabilities instead of just number when we know it's going to be only 1?

RAMA RAMAKRISHNAN: Sorry?

AUDIENCE: When we know it's only going to be 1?

RAMA RAMAKRISHNAN: Because you can't force the network to give you 1's or 0's. It's going to produce what it's going to produce. You can't force it to be exactly 1 or 0. It will give you some number. The best you can do is to tame that number so that it comes into a range that you like, like between 0 and 1.

So here, very quickly, we have a binary classification example like yes or no, this is the one-hot-encoded version, 1 or 0. This is what we saw in the heart disease example. When you have something like this example, fashion-mnist, where you have all these different possibilities, then you can encode it in one of two ways.

You can encode it just using integers, like 0 to 9. This is called the sparse encoded version. Or you can do a one-hot-encoded version of the output. You can have a one-hot-encoded version of the output.

And depending on how your data comes in to you, comes into your Colab, just pay attention to this. And depending on what it is, you have to pick the right Keras loss function. So if our data comes like a 1 and 0 thing, which is exactly what we had in the heart disease example, we use binary_crossentropy.

If your data comes in this form where it's sparse encoded, you use sparse_categorical_crossentropy. And then if it comes in this form, you use categorical_crossentropy. These are all equivalent things. It just depends on the data that you get, how it happens to be encoded by the people who sent it to you. And if they send it this way, use this loss function. If they send it that way, use that loss function.

Now, as it turns out, in our example here, the data is actually coming in this form. So we'll use this thing called the sparse_categorical_crossentropy. And categorical_crossentropy is a generalization of binary_crossentropy, which I'm not going to get into the mathematical details, but the intuition is basically roughly the same.

So this is what we have. If this is your output layer, use mean squared error. If this is your output layer, use binary cross-entropy. And if you still have a stack of these numbers, you can still use mean squared error. And if your output is a softmax, use categorical cross-entropy or sparse categorical cross-entropy.

OK. So let's actually run this in Colab. So this is what we have. Can folks see this OK? All right. So this is the data set we saw earlier. Here. As usual, we load TensorFlow and Keras. We load our usual three packages. And then we set the random_seed for reproducibility.

And it turns out that the fashion-mnist data is actually available in Keras. You don't have to go find it somewhere and bring it in. It's actually available in Keras. It's one of the standard data sets. We luck out. So we just actually load the data using this load_data command.

And then, if you do that, and conveniently for us, Keras has not only made the data available, it has already split it into a training and test set. So we don't have to do the splitting. And the reason they do that-- why would they do that? They do that so that different people who are building algorithms for that particular data set can all be evaluated using the same test set.

Otherwise, if I split it one way and say, hey, look how well I did that, well, I don't know. How did you split it? That's the reason. And you can see here that we have-- the input data is a tensor of rank 3. And basically, another way to think about a tensor of rank 3 is just a list of rank 2 tensors.

So here you have 60,000 images. 60,000 images. And each image is a 28 by 28 square of numbers. Each image is a 28 by 28 table. And then, of course, the output is just what category it is, a number between 0 and 9. So you just have 60,000 numbers. It's just a vector of 60,000 numbers.

So there are 60,000 in the training set. Oops. And then there are 10,000 in the test set. Same structure, 28 by 28. That's what we have. So if you look at the first 10 rows of the dependent variable y, you get these numbers, 9, 0, 3, 3, like that. There are numbers from 0 to 9.

So if you look at the fashion-mnist GitHub site, this is what it refers to. 0 is a T-shirt, 1 is a trouser, and so on and so forth. And 9 is an ankle boot. All right. So whenever I'm working with multi-class classification problems, I always do a little thing here to help me figure out that 9 corresponds to an ankle boot and so on and so forth.

It just makes it a little easier to work with this stuff. So I create this little list. And then it turns out, if you-- OK, what is the very first data point? What is it? What is its y value? It turns out to be an ankle boot. So you can actually look at the raw data for that image, which is just a 28 by 28 thing. And these are the numbers you have.

You can see all these. 250, 233, lots of zeros, and so on and so forth. And so you can actually look at the first-- visualize the first 25 images. I have a little bit of code here which visualizes that. Just Matplotlib code. And you can see these are all the images. They're kind of smallish.

This, my friends, is an ankle boot. It's like, OK, can the network really make any sense out of this thing? It looks very blurry and, I don't know. Oh, this is actually a better ankle boot. Look at that.

OK. Sorry. I'm getting distracted. So this is what we have here. OK, we're at 9:55. I'm going to stop so you folks are not late for your next class. So we'll continue this journey on Wednesday, and then we'll go on to color images in the next class as well. Thank you, folks. Have a good one.