In this video, we're

finally going to introduce the actual Python library Keras, which we are going

to use in order to build out our neural networks. In this section, we're

going to cover a bit of an overview of the

different Python libraries that are going to be

available in general. Again, we're just

going to be focusing on what our title here

was, which was Keras. We're then going to

show you how to set up a network structure using Keras. Then finally from A to Z, how to actually build

out that model. Once we are able to do that, we'll jump into the

actual code to create our first deep learning network within Python using Keras. What are some of the many

Python libraries that are available to us if we want to do deep

learning in general? Some of the most common libraries used include TensorFlow. TensorFlow is going to

be built by Google, and has now actually

incorporated Keras simplified syntax and to

that TensorFlow package. TensorFlow was originally known, as we talked through the

different packages available, as the more complicated

steep learning curve version of a deep learning framework, but with his

incorporation of Keras, has made it a lot

more accessible. TensorFlow also has

a larger community than most of the other

packages that are available, and is originally built

off of what we'll talk about next,

this package Theano. Now, Theano is essentially dead as development and

has seized back in 2017, but many academic researchers

relied on Theano, and Theano is

considered somewhat of the grandfather of deep

learning frameworks. Then we have PyTorch. PyTorch also has a

large community, and is currently a bit more research-oriented

compared to TensorFlow, which is a bit more into building out those

AI related products. PyTorch is developed by Facebook. Again, also has a

large community, was originally known for being more accessible than

TensorFlow, but again, with that incorporation

of Keras into TensorFlow, it's now also very accessible,

TensorFlow, that is. Then like we said,

Keras is just going to be that high-level library. Very accessible. Like Python when

we say high-level, that means it's very

close to English, and it can run either on

TensorFlow or Theano. With TensorFlow's

incorporation of Keras into the

TensorFlow package, it's most likely going to be running on TensorFlow

moving forward. On this course, we'll

be focusing on running specifically on Keras through

TensorFlow under the hood. That's going to be what

we will be focusing on in the code that we'll

see throughout this course. Let's talk through the

typical command structure when building out our

deep learning frameworks. The first thing that we're

going to want to do is actually build out that

structure of our network. How many layers do we want? How many nodes we

want in each layer? Then we're going to compile

that model that we create. However many layers it is, whatever the specifics

of each layer are, and we're going to learn

more complicated types of frameworks later on. We're then going to want

to compile that model, and when we compile that model, we're going to specify the loss function

that we're going to use at the end of our model. Different metrics that we want

to track, maybe accuracy, maybe loss functions,

things of that sort, as well as the optimizer

that we're going to use. That's also going to include the learning rate that we use in order to specify that optimizer. Now optimizer will be one of those that we

discussed earlier, whether that's atom,

stochastic gradient descent, something with

momentum, and so on. We're then going to fit the

model onto our training data. When we fit that

model, we'll specify the batch size as well

as the number of epochs, the number of times we'll

run through the dataset. Once we do that, we'll be able to predict on new data

once it's fit, once that model is already

fit onto the training data, and then we can

evaluate our results. Now, when we work with Keras, Keras is going to provide two approaches to building

the structure of our model. We'll have the sequential model, which allows for an easy-to-use

linear stack of layers. It's going to be

much simpler than the other version that

we're going to talk about, and more convenient

if the model has a simpler more relatable form that you're already used to. Whether that's going to be the dense networks

that we've talked about so far, or later on, just your typical

convolutional neural nets, recurrent neural nets, and so on. Then there's also

the functional API, and that's going to be a bit

more detailed and complex, but will allow for more

complicated architectures. Now, probably, given that

you're watching this course, you're just learning

this for the first time. Everything that the sequential

model is going to provide for you will probably cover everything that you

need to know so far. In this course, we'll be focusing therefore just on that

sequential model. Now let's talk through how to actually write

out the code in Keras to create this neural

network that we see here. This neural network will take in an input of a data set with

three different features, X\_1, X\_2, and X\_3. It will then have

two hidden layers, each one with four nodes,

as we'll see in a second. We're going to call it units, but they're one and the same. We have four units in each

one of the different layers, they are fully connected

as we've seen so far in each one of our

feedforward neural networks, and then finally

we're going to have that output layer, Y\_1, Y\_2, Y\_3. Here as you see, in the purple hidden layers, we're going to be using the sigmoid activation

function throughout. We can use different activation

functions if we'd like, but in this example we'll

just show you using the sigmoid activation functions. The first thing that you're

going to want to do is import that sequential function and initialize your

model objects. From keras.models, we're importing this

sequential function. Once we have that sequential

function available, we just initialize our model by setting model equal

to sequential. From there, we can add on each one of our different layers with their specific

activation functions to build out the

remainder of our model. Right now we just have initialize the model with

no details involved. We can then add layers

to the model one-by-one. In order to do so, we're going to use these different

types of layers. Here we're just

going to stick with dense layers which are those

fully-connected layers, and later on we're going to learn a bit more complex layers such as recurrent neural nets and convolutional neural nets, and we'll discuss those later on. But again, we're just going to focus here on dense layers, and then we can also

add on our activations. With those activations, we can specify whether we want sigmoid, ReLU, Leaky ReLU, and so on. The actual code is going to

look like what we have here. We have in that first step, importing the libraries and the model elements

we just discussed. From the keras.models

are sequential function, from keras.layers are dense

in activation function, then we initialize our

model as sequential. Then to add on our first layer, we're going to want to

specify the input dimension. That will just ensure that we have that first step correct, and that we're putting on LEGO

blocks that actually fit. We call model.add, then we're adding on to that empty model, a dense layer, that

fully connected layer, that layer is going to have

four units or four nodes. The input dimension coming in to that hidden layer

with four nodes, there's going to be

three dimensions, and those represent the

three features coming in. We're then going to specify

our activation function. We'll see as we do the code in a second that this

could actually be specified while we add on

our dense layer as well, and we'll see that when

we write out the code. But just to make this

easier to read and give you other syntax in order to add on the

activation function, we added on separately, and we call model.add, and we call activation, what type of

activation do we want, and we pass in sigmoid. Then to add on further layer was to make

a deeper neural network, we're going to have

that input dimension already presumed from

the previous layer, so you don't have

to keep writing out what the input dimension is. You can just call it model.add, we want a fully connected network and the units is equal to four, and then model.add

activation sigmoid, and we have created

our second layer here. Now there's more steps to complete that model

that we just saw, but we will walk through it in greater detail as soon as we get to the code in just a second. Just to recap, in this section, we discussed an overview of the different Python libraries available to build out these

deep learning frameworks. That included TensorFlow,

Theano, and PyTorch. Currently, PyTorch and TensorFlow are the main competitors. We're going to be using

TensorFlow specifically. To be even more specific, we're going to use the Keras

package as now available in TensorFlow and made TensorFlow a bit more accessible

than it ever was. We then discussed with that mine, setting up an actual network

structure using Keras, as well as how to build out

our models using Keras. We discussed that briefly. With that as promised

just a second ago, I said we're going to go a

bit deeper into actually building out those

networks using Keras, and we're going to see that in just the next video.

I'll see you there.

All right, welcome to our notebook here,

that will introduce Keras, this one is a bit of a lab, so hopefully you're able

to walk through some of this on your own. The goal here is going to be to use Keras

to build and train neural networks. We're going to be using

the UCI Pima Diabetes Dataset, which will just allow us

to predict whether or not a certain person has diabetes based

on the attributes that we have here. So we have nine different

features within our data set, one of them is the outcome

variable we're trying to predict. So we're working with eight features,

if you recall when we need to pass in that input, that input here

would be equal to eight. We're also going to start off by

having a random forest in order to just get a baseline value for

what the actual accuracy should be around, and then hopefully we'll try and

improve on that. And we may see as we go through this,

that deep learning may not always be the answer, and we'll see how much

longer it also takes as well, but often obviously will be

the answer sometimes. So don't say just because here it's

not answer that we should never use neural nets. So the first step is going to be import

here in this first cell are going to be libraries that were

already familiar with. And then in this second cell here,

you see that we're importing, and again I mentioned in the lecture that tensor flow

has now incorporated the Keras syntax. In order to get that

specifically from tensor flow, we import from tensorflow.keras

rather than saying from Keras. So from tensorflow.keras.models,

we import the sequential function, then from tensorflow.keras.layers,

we import dense. And we're just going to be working here I

believe with SGD will see later on, but from tensorflow.keras.optimizers we're

going to import a couple of options. And I would say feel free to try working

with some of these other options that we may not use. We're then going to import our actual

file that we're going to be working with, now it's going to be

this diabetes data frame. Wer'e going to name each one of

the columns as specified here, so we just have a list

equal to these names, and we're going to use those name specifically

when we read in our CSV file. We can then get the shape, and as

anticipated there's going to be 9 columns, one of them being that outcome whether or

not they have diabetes and there's 768 rows, so

this isn't a huge data set. Generally speaking, deep learning will work better

when you have a larger data set. We're then going to set x

equal to diabetes\_df.ilog and you see we're taking all of

the rows in just that last column. Everything besides that last column and

then for the y variable were just taking

that last column has diabetes. We're then going to use that

train\_test\_split with a 75 to 25% split, something that we should

already be familiar with. So that we get our x\_train, our x\_test,

our y\_train and our y\_test and then we can train on our train and

test on our test sets. We pull out the mean value for

y in 1 minus y here again, those are just going to be 0s or 1s. So it's going to provide some value

between 0 and 1, and it will show us what proportion of our data set is

already going to be a positive value. So because we see that 35% of the patients

have diabetes whereas 65% do not, we can get an accuracy of 65% by just

predicting that nobody has diabetes. So we've talked about classification and how we need to be careful when

working with something like accuracy. So we'll look at accuracy but what that will also look at

the ROC-AUC metric as well. So we're going to get our

baseline using random forests, so we're going to train a random

forest using 200 trees, so our number of estimators that we see

here is going to be equal to 200. So rf\_model and we initiate our

model RandomForestClassifier, this is a classification problem and then we fit that to our training set,

our x\_train and our y\_train. Now our models fit, and we're going

to want to predict the actual values, were also going to want to predict

the probability outputs for each one of our values, for

each one of our different rows. And we're going to do that so that we can plot our area under the curve

that our ROC curve in the next cell. So we call rf\_model.predict on our

x\_test as well as rf\_model.predict problem in order to get

the different probabilities. And then we can get what are actual

accuracy is by passing in the y\_test and the predicted value and then we can

get the ROC-AUC score by passing in the y\_test and

the predicted probabilities. And here we say that that will output the

predicted probabilities for each one of the classes, we want just for the positive

class, which is why we say one here. So we see that our accuracy is about

77.6 better than just predicting, not for everything right out of 65%,

and our ROC-AUC is going to be 83.6. We're then going to create this function

to plot our ROC curve and we'll use this later on, so we create our function so

that we can use it again later on. We get our false positive rate and our true positive rate as well as

the threshold that we're using by calling ROC curve on the y\_tests and

our predicted values. And that's just going to be whatever

we pass in as our y\_pred, and we should pass in those probabilities, not actual predictions as well as

the actual model that we'll be using. Or the model name of what we're using so

that we can specify this is random forest versus our neural nets,

which will use later. We're then going to just initiate

our figure and our access, and then we're going to plot our false

positive rate versus our true positive, with a black line. And then we're going to

plot just a straight line, which would show us if we were to just

predict randomly about how well we do, so you can see that area

over that line as well. And that's going to be a dash line, that's

going to be 0.5 rather than that full line width of 1, so

we'll see a smaller line there. Keeping our grid and then we can set

our title as well as our x limits, so from essentially 0 to 1 on our x and

y axis. Then once that function has been created, we just call plot\_roc on our y\_test,

and our y\_pred probabilities. Again specifying that we just

want the positive values, and then we're saying that this

is a random forest for rf. And we see here the ROC curve for

RF on that Pima diabetes problem and I see it does better than

random prediction and perhaps we can do a little bit better. It's not a perfect prediction

as we have that ROC-AUC of 0.836 where one is perfect. So that's going to closeout our baseline

values, so you can remember these values as well as this graph that

we have here of 77.6 and 83.6. And with that we are now ready to

build out our first neural net model, all right, I'll see you in the next video.

Welcome back. Hopefully,

at this point you're excited to finally build out

your first neural network. Here we're going to build just a single hidden

layer neural network. We're going to have, again, that input of eight variables

and we're going to have one single hidden

layer with 12 nodes. Now, something that

we didn't touch on for neural networks which we'll touch on in

the next lecture that we do after this notebook, is that it's going to be

important to actually scale your data before building

out your neural networks. The reason behind this will have to do with how

gradient descent works and how it will update certain weights differently

depending on their scale. We'll get into that

in the next lecture. But for now, note that you're

going to want to scale your data before performing

your neural networks. We create our normalizer

with that standard scalar. We then create our X\_train\_norm by calling fit\_transform

on our X\_train. Then we have our

X\_test normalized by just using transform, not

fit\_transform, again, because we want to

ensure that our holdout set is indeed a holdout set and that we're actually

using something that we learned from the

training set that we should have had

available to us. Here we build out

our first model. As we discussed in

the lecture model\_1, we initialize our model. We call it sequential. We're then going to add

on our first layer, which will be 12 units. That's going to be

the default first value, our input shape. We just have to say how many. You don't have to specify

the number of rows, but the number of columns is going to be what's important. Then, like I said during lecture, we can skip a step

that we saw when we walked through the

actual syntax and we can actually include

this activation within this model.add as we

add on that dense. This is still part

of that dense layer, if you look at where these parentheses actually close out. We set our activation

here equal to sigmoid. We showed other options that are available

to us and we could use Adam or ReLU or

LeakyReLU, and so on. Then to close it out, we're just going to be

condensing those 12 nodes into one node to predict some

value between zero and one. So we add on one dense layer

that's fully connected to those 12 nodes and we set the activation

equal to sigmoid again, because we want it to output a value between zero and one. You run that. We've

initialized our model and we can call model\_1.summary() and get some of these nice details about our actual layers and how many weights

there are going to be. If we look here, we see the

total amount of parameters that we need to

train and how many we need to train at each layer. We see that we have

121 total parameters and 108 at the first layer

and 13 at the second layer. Now, I would advise for

you to pause and try to think through why there are 108 parameters and 13 parameters. I'm going to give you a

second here to pause. Assuming you paused and

thought this through, the reason why we have

108 parameters at that first layer is going to be we have

eight input features, and then we have that fully connected to each

one of the 12 nodes. If you think about that, you'd originally think maybe

something like 8 times 12. But we also have that bias term, so it's going to be nine units that are

actually connected. So 9 times 12 is going

to give you your 108. Then to get to the next layer, again, you're going

from 12 down to one, so it's going to be fully

connected to that one plus the bias term so you

have 12 plus that one, and that's going

to be equal to 13, which is why you're going

to have to learn a total of 121 different parameters. We're then going to

compile our actual model. This is going to

be our first time seeing how to actually compile that model using this

specified optimizer, our loss function and the different metrics that

we want to track throughout. We call model\_1.compile

and we say SGD, so using stochastic

gradient descent which we imported earlier. Our learning rate is 0.003 and we can change that learning rate to make it faster or slower. We then have our loss function, which is going to be

binary crossentropy. So that's binary either

0 or 1 in cross entropy. If you wanted something that's

going to be categorical. So across many different

categories then you would use categorical

crossentropy. If you wanted to do something that's going to be continuous, then you can do Mean Squared

Error, which is MSE. Then we say the metrics

that we want to track. Here we want to track accuracy. Then it'll automatically also track the loss

function throughout. We are going to actually

save the run history. We will see how this becomes

useful throughout and that's going to be one of the

outputs from our fit. We are going to fit

to our X train norm, to our y train. What we can actually do is

pass in our validation data, pass in our test set to see

how we are performing on that holdout set as well as

we fit to our training set. Then we set the number of epochs, the number of times we want

to run through our data-set. We set that equal to 200. So it's going to run through

the full data-set 200 times. At each one of the

different steps, it's going to say each one

of the different epochs. How much are we increasing

or decreasing our loss? How much are we

increasing our accuracy, and how we doing on

that validation loss, that holdout set, how

are we doing overall? I'm going to pause the

video as it'll take just a second and I'll see you as

soon as it's done running. Oh, it's done running. It's only 200 epochs we

are able to run through. Then, like we did

for random forest, we're going to generate

two kinds of predictions. One's going to be that

hard prediction and the other one's going to be

the probabilistic score. So we have predict our different

classes model\_1.predict classes which will be

available to us once we fit the model. Then we have are just predict

once we fit the model, again, not being model\_1. We have our different

predictions and we can see for our

different classes, we had there at zeros or ones. For our probabilities,

we have some value between zero and 1 and

that differentiator, as you look at this,

is just going to be whether or not it's

greater than 0.5. With that, we can then

create our ROC AUC curve. We created a function earlier

that allow us to do this, as well as looking at our

different accuracy scores and our ROC AUC scores. Let's see how we did compare

to our baseline model. We see if you recall earlier, we did a little bit worse and

it's hard to tell exactly from the curve but we can

see the ROC AUC is 0.782, whereas before is

somewhere around 0.8 and our accuracy is 0.729. That's going to be our

first neural network model and there may be some

variations due to, we randomize that initialization. So there is some randomness involved in creating

these neural net models. So you may not get the

exact same result, but hopefully, you have

between 75 and 85. Here we did a little bit

worse percent accuracy. Our AUC again is a little bit worse than between 0.8 and 0.9, but you may end up

with a higher value depending on your initialization. When we save that history from

that fitting of the model, what we actually did is we were able to get

this dictionary. So run\_hist\_1. If we just look at this and look at the type that we have, this is the initial

output that we saved and this is going to be that history's object that

Keras makes available to you. That's going to have with

it this history attribute, which is just going

to be a dictionary. As we just saw, that

dictionary has certain keys. Those keys are going to be the actual loss and it's going to be the loss at each one of the different epochs, the accuracy levels at each

epoch, the validation loss. So for that holdout

set what your loss was and then your

validation accuracy. That's only because we specified

that we wanted to track accuracy when we first

created our model, when we compiled it here. Otherwise, accuracy would not be available within

this dictionary. Once we have each

one of these things; the loss, the accuracy,

and validation loss, we can actually plot these out. We initiate our

figure on our axis. Then we call

runhistory.historyloss to get the different

loss values at each one of the different epochs. That's going to be in

order as it trains, so it should get lower and lower. Then we can also get

our validation loss, and we'll plot that in

either red or blue. Red is going to be the loss

on the training function, so that should always be

going down as it gets closer and closer to fitting

exactly to our training set. Then our validation

loss, hopefully, it was getting smaller but

could have possibly increased, meaning that we

overfit our dataset. Here, we see that they're

both still going down, on both the training set

and the validation set. This suggests that

the model might benefit from further training. So running through more epochs. With that in mind,

let's train the model a little bit more and

see what happens. Something to note is that it will pick up

from where it left off. It'll continue to

train given where it left off at these 200 epochs. We're using that same model

1 that's already been compiled and fit once, and we're going to run it

for another 1000 epochs. I will run this. Then again, I am

going to actually pause the video

here and we'll come back once it's done running. As this will take approximately, five times as long since it's five times as many runs

through the dataset. That may have taken a

couple of minutes to run, but now it's run through

all the 1000 epochs. We want to see what kind

of improvement did we get. Recall, as we fit on

that training set, that validation error

should keep going down. That accuracy should

continue to go up. Whereas, for that validation set, for that holdout set, it's possible that we start to overfit and that we tend to actually have that loss function go up or the accuracy go down. We're going to plot. First, we're going to call n, and that's the length of

our original run hist. If you see output here

was called run hist 1B. If we look at our

original run hist, we'll say the length

of that was n, which is our first 200, and then 1B should

be the next 1000. Then we're going to

plot for range n and taking run hist 1 and

get that loss function. Again, just for that

first 200 epochs, and we plot that in red, and this is for the training set. Again, if we just say loss, if we want to see

the holdout set, we say val loss, which

we'll see in just a second. Then for n through n plus m, so from 200-1200, we're going to look at the

additional loss. How much are we able to

improve that loss function, how much is able to decrease

as we did 1000 more epochs? Then we're going to

do the same thing for the validation loss. Before we do red and hot pink, and that's going to

be our train loss, and then blue and light sky-blue, so that we can differentiate between the first run

and that second run. We plot this out, and we see it continued

to decrease after 200. Then we see the validation loss actually also

continued to decrease, but really start to flatten out. The training loss

decreased even further, not at that same

rate as it began to fit and perhaps overfit a

bit to that actual data. That closes out our

first neural net model, playing around with running

it through different epochs, seeing that plot and the

output once we fit that model. In the next video, we're going to try

and play around with different models and see what kind of effect that will

have on overall accuracy, as well as how fast

it will be able to fit. I'll see you there.

Let's close out this video

with exercise 2 over here. We're going to build

another model. This time we're going to

have two hidden layers, each with six nodes. We're going to use the

ReLu activation function for each one of those

different hidden layers, which is generally going

to be best practice, and then sigmoid for

that final layer as we're trying to output

values between zero and one, and if you recall for ReLu, you're not going to necessarily get values between zero and one. We're then going to use a

learning rate of 0.003, and we're going to

train for 1,500 epochs. That's going to take

some time as we size, we did 1,000 epochs. I will do that, pause and

continue as we go through this. Then as we did before, we're going to graph

the trajectory of the loss functions, as well as the accuracy on both the training and test set. Then we'll plot the ROC curve for these different predictions. We're going to initialize our model as a sequential model. We just call sequential. Then we're just going to add on each one of our different layers. We have our first hidden layer, and that's going to be

advanced layer with six nodes, so fully-connected. We pass in for that

first hidden layer, the actual input shape, and that's going to be eight. For the activation this time, rather than saying sigmoid, all we have to do is change

that string to ReLu. For the first layer, then we can do that again

for the second hidden layer, so we just add on

again model.add dense. This time we don't

need the input shape. We said the activation

equal to ReLu once again. Then to get to that final output, we recall that we only want

one node as we're just trying to predict one or zero for each one of the

different values. We therefore have a dense layer fully connected to just one node. We want that activation at a final layer to be

equal to sigmoid, as we want some value again

between zero and one. We're then going to

compile our model, and that's the next step. There we're going to pass in our optimizer as well

as our loss function, as well as the different

metrics we want to track. We're saying we want

Stochastic gradient descent. Feel free on your

own to try Adam, as well as RMSProp which

we imported earlier, as well as playing around with different learning rates perhaps. Then we're going to still use binary cross-entropy

as we're still trying to figure out a

binary zero or one value. Then we're going to also track accuracy as one of the metrics. We're going to save our

history as run\_hist\_ 2, and we get the

output from calling model\_2.fit on our

x\_train and our y\_train, and then again, we can

specify here are what we want our holdout set to be so that we can track

that throughout. With that, we also,

when we fit our model, need to specify how many times we want to run

through our dataset. How many epochs we want? We said in the exercise

prompt that we want 1,500 run-throughs of

the entire dataset. I'm going to run this, and you see we start to get each one of the different epochs, and we see that the

accuracy increases, the loss decreases and at

least in the beginning, the validation loss should

continue to decrease. I'm going to pause here, and we'll come back

once it's run through those 1,500 different epochs. Now we have run through our

1,500 epochs as we see here. Just a quick reminder, if we call run\_hist\_2, which we defined when we set

it equal to the fit output., we have our dictionary of

keys with our different keys, which is the loss, the accuracy, the validation loss, and

the validation accuracy. Now, last time we didn't do this, but here we're

actually going to plot the accuracy as well. We're going to create subplots

and in that first subplot, so we call plt.figure. Then on that figure, we add on the subplot and

we're going to say that it's a one-by-two

subplot and we want to look at the first one. On that ax, on that bounding box, we're going to plot the loss, as well as the validation loss in red and blue respectively. Then in the next subplot, so we call add subplot and we say we want

that second subplot, we're going to call the accuracy as well as that validation accuracy

in red and blue as well. We're going to include

a legend on each, so you won't have to remember

which one was which. You run this, and here we see that on that training set obviously

continue to go down, but on that validation

set going through 1,500 epochs and those two layers, we definitely overfit

our dataset as we see that the validation loss actually starts to increase

around that 800 epochs point. We see a bit of an inflection. Then we can see that accuracy jumping up and down as

it tests throughout. We see that fairly increase throughout as we fit our

model closer and closer. Then again, we see a bit of a decrease around

800-1,000 epochs, and that's not going to

correlate perfectly. This is just according

to our loss function and then our overall

accuracy when we check it. We see that jumps up and down. But really [inaudible] , definitely pass out

of 1,000 epochs mark. Then we can use what

we did before in order to predict both the classes as well as the probabilities. Check for our accuracy as well as our ROC AUC score once we get those outputs using

the predictive class as well as the predicted

probabilities. Then we can also plot our ROC using that function that

we defined earlier. You run this and we see

a bit higher accuracy, a bit higher of ROC AUC, and then we see

the curve as well. Now, again, there's a

bit of randomization so we don't necessarily get

the exact same answer. Maybe there's not that

much improvement and especially with

the amount of time it took to train this model, there probably wasn't

enough improvement from our random force. Keep that in mind that

sometimes it's not always going to be

the best solution. There is this talk within the data science community where you just throw neural

nets and everything, now that's not best practice. I want to ensure as

you watch this video, that you keep that

in mind as well. But they will be very powerful throughout and that's why

we're learning it here. Well, that closes out our video here with the

introduction to Keras. Feel free to play around with this model

that we have here. You can add on more layers, increase the amount of nodes, decrease amount or nodes, change the activation functions,

change your optimizers. You play around

with each of these and see how the model runs. I would say don't do each

one for 1,500 epochs, you're probably overfitting

and it'll take too long. But it is worth playing around, getting familiar with

what you can play around with within this Keras framework. All right. I'll see

you back at lecture.

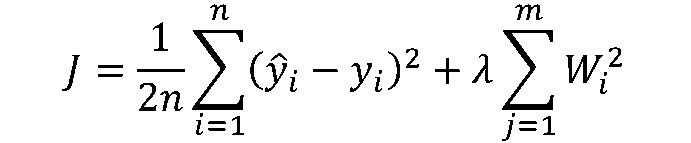
# Summary/Review

### **Deep Learning and Regularization**

Technically, a deep Neural Network has 2 or more hidden layers (often, many more). Deep Learning involves Machine Learning with deep Neural Networks. However, the term Deep Learning is often used to broadly describe a subset of Machine Learning approaches that use [deep Neural Networks to uncover otherwise-unobservable relationships in the data, often as an alternative to manual feature engineering. Deep Learning approaches are common in Supervised, Unsupervised, and Semi-supervised Machine Learning.](https://www.coursera.org/learn/deep-learning-reinforcement-learning/supplement/BFNil/summary-review)

These are some common ways to prevent overfitting and regularize neural networks:

* [Regularization penalty in cost function - This option explicitly adds a penalty to the loss function](https://www.coursera.org/learn/deep-learning-reinforcement-learning/supplement/BFNil/summary-review)



* Dropout - This is a mechanism in which at each training iteration (batch) we randomly remove a subset of neurons. This prevents a neural network from relying too much on individual pathways, making it more robust. At test time the weight of the neuron is rescaled to reflect the percentage of the time it was active.
* Early stopping - This is another heuristic approach to regularization that refers to choosing some rules to determine if the training should stop.

Example:

Check the validation log-loss every 10 epochs.

If it is higher than it was last time, stop and use the previous model.

* Optimizers - This approaches are based on the idea of tweaking and improving the weights using other methods instead of gradient descent.

## **Details of Neural Networks**

Training Neural Networks is sensitive to how to compute the derivative of each weight and how to reach convergence. Important concepts that are involved at this step:

Batching methods, which includes techniques like full-batch, mini-batch, and stochastic gradient descent, get the derivative for a set of points

Data shuffling, which aids convergence by making sure data is presented in a different order every epoch.

## **Keras**

Keras is a high-level library that can run on either TensorFlow or Theano. It simplifies the syntax, and allows multiple backend tools, though it is most commonly used with TensorFlow.

This is a common approach to train a deep learning model using Keras:

1. Compile the model, specifying your loss function, metrics, and optimizer.
2. Fit the model on your training data (specifying batch size, number of epochs).
3. Predict on new data.
4. Evaluate your results.

Below is the syntax to create a sequential model in Keras.

First, import the Sequential function and initialize your model object:

**from keras.models import Sequential**

**model = Sequential()**

Then add layers to the model one by one:

**from keras.layers import Dense, Activation**

**# Import libraries, model elements**

**from** keras**.**models **import** Sequential

from keras**.**layers **import** Dense**,** Activation

model **=** Sequential**()**

**# For the first layer, specify the input dimension**

model**.**add**(**Dense**(**units**=**4**,** input\_dim**=**3**))**

**# Specify activation function**

model**.**add**(**Activation**(**'sigmoid'**))**

**# For subsequent layers, the input dimension is presumed from the previous layer** model**.**add**(**Dense**(**units**=**4**))**

model**.**add**(**Activation**(**'sigmoid'**))**