Welcome to our lab

here on autoencoders. In this lab, we're

going to be going over dimensionality reduction

techniques on the MNIST dataset, which is those handwritten digits that we'd seen in

earlier notebooks. We're going to perform

dimensionality reduction using first PCA as a baseline, which we learned in

earlier courses. Then as just

discussed in lecture, autoencoders and

variational autoencoders. With each one of these models, we're going to use the

appropriate scoring metrics so that we can compare the

performance across each. A quick reminder of what

the MNIST dataset is, it's going to be handwritten

digits between 0 and 9, we'll have 70,000 different

handwritten digits, all in black and white. When we run this traditionally we're

going to split this into 60,000 training images and

then 10,000 validation images. This is such a common

dataset that's actually built into Keras. So we pull it out

from keras.dataset. We pull out the

MNIST functionality, and with MNIST we

can call load data. Once we call that method, we get the output of

our training set, x\_train and y\_ train and our

test set, x\_test and y\_test. We can actually look at

quickly the shape here. We're working with

28 by 28 images. If you recall, because

it's black and white, this won't be three dimensional. If it had RGB or some

color scheme to it, then it would be three dimensions

such as 28 by 28 by 3. But here it's all

black and white. So our pixels will all be on the gray scale and

that'll be 28 by 28. If we look at just

a single image, we can see that each one of these different pixels are

going to be some number between 0 and 255 representing the light or darkness of each one of these

different pixels. This takes a lot of space, so I'm just going

to clear that out. Now, we want to make

sure that they're all between 0 and 1. So we're just going to

divide here by 255. Then if you think about it, max number is 255,

we divide by 255, the max number is now one. We're going to use now PCA

as a baseline with which we can compare against our deep-learning models,

which we'll do later on. For PCA, we're going

to have to treat each image like a row of data. We're going to have to flatten

out those 28 by 28 images. In order to do that,

all we have to do is call that x\_train.reshape. We saw that the original

shape was 60,000 by 28 by 28. We keep that 60,000 because we still have

60,000 different examples. Then we just take a product

of that 28 by 28 here. That's.shape one through

is going to be that 28, and then the second

28, np.prouduct is the product of the two. That will end up

with 60,000 by 784. Then same for the test set, we'll have 10,000 by 784, and we see those results here. Now, just a quick one sentence

reminder on how PCA works. PCA will do a matrix

decomposition of this data that we're working with to

find the eigenvalues. These eigenvalues

will end up being the principle

components of our data, or those latent

features that describe the maximum amount of

variance in the data. We've been talking about

these latent features, that lower dimensional space

that hopefully represents the important portions

of each one of these different pieces of

data that we're working with. So just to ensure

that it's scaled, we already got it

between 0 and 1, but we will just run

the MinMaxScalar here. We call MinMaxScalar, which we imported from

sklearn.preprocessing. We fit it on our x\_train\_flat, that flattened data

that we just produced. Then we call transform

on that x\_train\_flat. Now, we have our x\_train\_scaled. Now, the function that

we're going to use for PCA, it's going to take in some data and that will end up being this x\_train\_scaled

to start at least. We say the number of components, how much do we want to

reduce our dataset by? That's going to be

something that we determine in the preset. So we can say we only

want two dimensions, five dimensions, so on. Going into the actual function, are going to call PCA. That model needs the

hyper parameter of how many components

do we actually want? We say that we want the number of components that we defined

here above in the function. Then we have our PCA model,

that's been initiated. We then fit that model to

our data set by calling pca.fit on that X data that we passed here

into our function. Now we have this fit PCA, we have that model

fit to our data. Then we can print out here

quickly after we run this, how much of the overall

variance was explained with the number of components that we pass in

through our function? So we say variance explained

with say two components. Then we just print out the

actual amount explained. That's going to be done by

calling that fit model, getting the explained

variance ratio, which we'll show in just a second what that

actually looks like. But that quickly is

going to be the amount of variance explained by each one of our

different components. We take that sum and we

see the overall amount explains where the maximum value, summing those altogether

should be equal to one. We're then going to return

our fit model so that we can use that once

our function has run, as well as our transformed data. So that's going to

be our data set reduced down to our new

number of components, that new number of dimensions. So we're going to do this

with 784 dimensions. So that's going to

be all the data. When I run this,

recall it's going to print out the amount

of variance explained. I would want you to

think to yourself how much of the variance

should be explained. We run this, and with all 784 components being

taken into account, all of the features being

taken into account, we see that a 100 percent of the variance was explained,

which makes sense. Now, just to be

clear on what this explained ratio attribute

is of our model. If I run this, this is going to be

an array that says, for each principal component, what is the marginal amount

of variance explained? The first component explain

this much of the variance. The second one this much, I believe it's around

0.09 and 0.07. We can see that the

length of this, given that our PCA model took all the components, will be 784. If we actually plot this out, we can take that

cumulative sum across all 784 components and that's

what this cumsum does, for this entire array

that we just printed out. We can see how much

of the variance is explained as we add

on more components. We can see that we need

about 250 components to explain about 90 percent

of our variance there. Now for visualization purposes, let's try reducing it down the number of

features down to two. If we reduce the number

of features down to two, then perhaps what

those two features we can see whether or not we're able to group together where the ones lie in

those two features, where the twos lie with

just those two features, so on and so forth. So you run this and

we get our output of both the model as well

as our transformed data. Now I'm going to get to

this in just a second. That's just an example of how to explain what we have here. What we want to do is

plot out each one of these numerical values that

we have zero through nine. We have those labels

available to us. For each one we

want to plot out in these new two-dimensional space that we have where each

one of those points lie. Just to ensure that we don't have too large of a scatter plot, we're only taking the first

250 examples of each, remember our X train

has 60,000 examples. That would be quite a

dense scatter plot. Then for numbers ranging

between zero and nine, including zero and nine. If we say range 10

doesn't include 10, we're going to create this mask, which is just saying, does Y train equal that number

where we currently are? So we can say here, let's just our Y

train equals zero. We can see whether or not that exact example in our full array. We'll just put that out. Sixty thousand. It's going to be for each one

of the different examples. Is it labeled a zero? We're then going

to mask our data. This MNIST\_data \_2 again is

the output from our function. So it's that

two-dimensional data. We're only taking the rows from that two-dimensional data where our y-train is equal to the current number

in our for loop. Then we're going to take

just that first column. We only have two columns, only two features, and then we're only taking the

first 250 examples. Then we're going to

do the same thing to get our y-data or our second axis by saying, again, I only want the rows where we have it equal to the

Y-label specified. So starting off at zero, is it equal to 0, and then

we want that second column. Again, only the

first 250 examples. We then plot that scatter plot of the X-data and the Y-data, we label it, we'll call

that legend later on. Once we run this, we can see for each one of the different values

whether or not they create clumps in

this two-dimensional space representing that 0, 1, 2 and so on. We can see, for example,

the ones here in orange are already disentangled and grouped together on their own. We can look at the nines

which are light blue here, and the fours which are purple, and we can see that those are fairly close to one another, which makes sense given the way that fours and nines are drawn, and you can continue to explore

where groups are able to separate themselves out or where there seems to be

some type of overlap. But we can already see that

these latent features within PCA are learning somewhat how to disentangle our features and perhaps a neural network could help even further

and doing this. Now we want to score our PCA. Again, we're going to

want to come up with some actual function

to decide whether or not we are improving how

well we are performing in regards to

working with the PCA versus working with

our network models. The number that we're going to be using, the latent features, the amount of latent

feature we're going to be working with, here's

going to be 64. We call that MNIST PCA function

that we defined earlier. We call 64 dimensions,

we run this, we now have our model, besides that model also

the actual transform data. We're then going to take our X-test flat that

we defined earlier, and scale it so it's on the

same scaling that we used. This S was for

transforming above. Here are X-train, so we're using that same S to

transform our X-tests. We're then going to use that

PCA 64 that we just fit on our training set to transform

our X-tests scales. That's going to give us

X-tests flat in 64 dimensions. Then we're going to reconstruct that same image back to the

original dimensionality, that 784 dimensions that

we are working with. We do that by calling PCA 64, which is our model

inverse transform on that X-test flat 64

that we just produced. On that 64 dimension data that we just produced by

calling PCA 64.transform. This is that reconstruction

that we're trying to do. We're reconstructing

our original image by reducing the

number of dimensions, and then going back to

that original amount of dimensions that

we are working with. We can look at the shape here, and as expected, given

that we have the test set, there's 10,000 different samples, and reconstructed there back to that original

dimensionality of 784. We're then just

going to call this true and reconstruct it so it's clear when we

call this into our model. That's going to be our X-test scaled versus our

reconstructed data. We're then going to

come up with a mean squared error of

that reconstruction. So we just say true minus reconstructed for all

the different pixels, and we just average out the total error

that we have there. We can call that on

our now true and reconstructed that we

just defined above. We see that we have an

average mean squared error, about 90.5 when you use

64 components for PCA. That's going to be the baseline

that we're working with. We see a mean squared

error of 90.6. That closes out our motivation building out that baseline. In the next video, we're going to start to

work with autoencoders, a simple autoencoder

to see if we can do better than this baseline performance that

we currently have, a 90.6. I'll see you there.

Welcome back to our lab

here on autoencoders. If you recall in the last video, we used PCA in order to reduce the number of

dimensions down to 64, and then reconstructed our

image and then saw how far off our reconstructed image

was from our original image. Now we're going to

do the same thing, except using autoencoders

and using neural nets. Now so far, whenever we

built out our neural nets, we've been using Keras and from Keras we've been using

this sequential API, where we just add on layers and that's a bit of a simpler way of building

out your neural nets. What we're going to introduce

here is the functional API. That's going to be

fairly simple as well, and we'll walk through

the steps here, in regards to actually building out these complex architectures. What it will allow for, is for more flexibility in

building out your models. If you think back to our convolutional

neural net discussion, and we didn't have

a notebook there. But if you wanted

to build some of those more complex architectures, such as Inception or ResNet. You'd have to actually use this functional API in

order to build out layers, such as with Inception, where you're

concatenating a bunch of different types of

layers together, or ResNet where you want to bring along portions of the

layer to further layers, you'll have to use something

like the functional API. So it is worth getting a hang of as we talked

through it here. So we're going to import

from tensorflow.keras, this input and this dense layer. We'll see how those are

used in just a second. As well as importing model, which will be the key

function in regards to the functional API within Keras. So the goal that we have when we build out our autoencoder, will be to build out

three different models. Will have that full autoencoder, and that's taking that inputs. Remember the inputs

and the outputs would be the same but taking that image and then ultimately reconstructing that

image at the back end. So that'll be the

encoder and decoder, where we deconstruct and

reconstruct these images. We'll have the encoder portion, which is just the portion

that will take inputs, and try to bring them

into that latent space. Then the decoder which will take that latent space and

try to reconstruct it. So we're setting our

encoding dimension to 64 as we did with PCA. So we're going to have

that latent space in 64 dimensions. We have to define when we

use the functional API, what our input is going to be. So this is just creating

a blank tensor, and when we say a tensor, and that's where the word

tensorflow comes from. All we're talking

about is an array and a certain amount

of dimensions. So we're going to be using

probably here two dimensions. One dimension is the

number of samples. The other dimension, which we want to define the shape of, is going to be how many actual features

we're going to have. If we kept it at 28 by

28, then for shape, you'd probably put in 28 by 28, and then you'd leave

out the 60,000 still. But we're initiating

this blank tensor and we're going to

need this in order to define what the

flow of our inputs to outputs actually are

within the functional API. Then the reason why it's

called a functional API, why we have the term

function involved, is they become clear right here. We create this dense

layer and this is similar to the dense

layer we saw before, or we call dense. We say what's going to be

the dimension of that dense? How many hidden nodes

are they going to be, and what type of activation

are we're going to use? This portion right here

is actually a function, and that function will be

able to take a certain input. All we have to do is

say, as an input, we want these inputs that we

defined here, that tensor. Then we only have a

simple model here. So we're going, and this is just the encoder model

the encoder portion. In order to bring

together all the steps, we call model, and we call

the inputs and the outputs. As long as the inputs

and outputs match up, even if we added on a bunch

more layers in between here, which we'll see later on. As long as the first value is able to reach

that final value, given our functional

API and how we define to each one of

the different steps, the model will be able to

bring those all together. So we have our encoder model, which is this model

that starts with the inputs and ends with the outputs from this

encoded portion. We then are going to

have our decoder model, which again we're going to

have our input defined, and this time the input

for the decoder model. If we think about the

idea that it's taking that latent space and

reconstructing our image, should be the dimension

of that latent space. So it can be taking in

vectors with dimension of 64, that encoding DIM

that we have here. We're then going to pass

that through a dense layer. We call that dense layer and we want that

dense layer to be reconstructing our image

backup to 784 dimensions. We use the activation

of sigmoid and what we pass into that dense layer

is in this parentheses, as you would with a function, we pass in the encoded inputs. Then, our decoder model

will just be again, that model and then saying what the input is and

what the output is. Then, to define the full model. First, we are going to say what our outputs are going to

be for that full model. If you recall, our decoder model as currently constructed, just takes in this input, which is just a blink tensor. Instead, we're going to define the input as the

encoder model inputs. It's going to move from inputs, pass out the encoder

model output. It's actually going to take

that input that we defined, since we had that

within the function. The output of the encoder model, will then become the input

for our decoder model. Then, that decoder

model will by defaults, because the way that we

constructed the decoder model, output that final dense layer that reconstructs

our image. Then, our full model

will be the inputs, which are defined up here,

those initial inputs. Then, those final outputs

which we just defined. We just created all

the steps needed, runs from these inputs

through the encoder model, then through the decoder model, and then outputs

this reconstruction. You run this. We now have our

full model available to us. We set our model, inputs equals inputs,

outputs equals outputs. It's already what we did here. Then, the steps from there

in regards to compiling and fitting the model are the same as with the sequential API. We call it compile, we define our optimizer, our loss-function, and then what metrics

we want to track. We're going to track accuracy, and then we're going

to run this just for one epoch here, on

our x\_train\_flat. Then, recall that our outputs are also going to

be x\_train\_flat, our input and output are

going to be the same. We fit the model, running through just one epoch, this will take just a second. Afterwards, and also

we have the option here to actually just

look at the summary. We'll do full model.summary, which we'll look at right

after this is done. In 3, 2, 1, look at the summary and we

can see that it passes through that

input of dimension 784, down to dimensions of 64, that being our latent space and then reconstructs that image. Now, the way that Keras works is, because we built out the

smaller intermediate models, but actually train

them along the way. That encoder model, that we fit into this full model

that we defined up here, has actually been

fit to the data. We can actually encode our image and output that

64-dimensional space. We run this and we see we

took our x\_tests\_flat, which was originally 10,000 by 784 and reduce down the

dimensions, down to 64. We can look at that

and see that we now have those values which represent the different pixel values or that 64-dimensional

version of the encoded image. Now that we have this available, our goal is to see what the reconstruction

error actually is. We want to use the

trained autoencoder to generate reconstructed

images and then compute the pixel

wise distance between that reconstructed image

and the original image, and see how it did compare

to our baseline PCA. We have our full model, which will both encode

and decode our model. We can just pass in x\_test\_flat

into our full model. If we call full model.predict, on x\_test\_flat, it's going to try to bring that down

to the latent space. So encode it and then

decode it again, into what should be

exactly the same. If it was able to

do it perfectly. All the decoded images will, when we run full model.predict, both encode it and decode it. It's doing that

reconstructing step here and we get our

decoded images. Then, we can run that

MSE reconstruction that we defined above, to see what the actual error

is on our decoded images, compared to our

original x\_test\_flat. You run this and we see that

it did significantly worse. If you recall, we had around

95 for the mean squared error and now we're

at 346 of that. We could have ran for

more epochs as well, rather than just one epoch

to better fit the model, but even with five epochs, you'll see that it's

still does a bit worse. Hence, in the next video, we'll see how instead of

just using one hidden layer, perhaps we make our network a bit deeper and still run

for a bit more epochs. Hopefully, from there we start

to actually do better and perform better than what

we saw with the PCA model. I'll see you in the next video.

Welcome back to our Autoencoder notebook. As we saw in the last video, we built our

first autoencoder with the encoder and decoder portions of our model and

then with the reconstruction, we saw a fairly high error. And one of the reasons that we had this

poor model was that we aren't really doing any deep learning. So what we're going to do here is first, we're going to start off by

adding on some extra layers. We're also going to run it for

a few more epochs and see if we can get a lower reconstruction

area than we did with PCA and see how much we're able to

actually improve on that error. So again, our final encoding

dimension will be 64, but this time we're going to be

including a hidden dimension of 256. And this thing will work very

similarly to what we just did. So this is another time to look

through your practice through the functional API for Karas. We're starting off again with the input

and we just say the shape and that's going to just initiate what type

of tensor we want to pass through. And then we're going to have

first that hidden dimension. Which is just going to be a dense so fully

connected layer and we're going to pass into that fully connected layer with

that hidden dimension of 256 our inputs. So we're just passing our inputs

that will produce some output. We can pass that output through

to another dense layer, so this is the hidden layer, and

then this is encoded layer and this will take in that encoder

hidden within the function. And this is the function itself. It's going to be another dense layer. Now reducing it down to 64 nodes. And then to just create the model,

we just say model and we're going from inputs all the way

out to the encoded outputs. You don't have to write out this middle

step, as long as there's a connection between what's being input all

the way through to the output. Within your model, all you're passing through is what

the input is an what the final output is. And then we do our decoder model and

again, our decoder models should start off

with that encoding dimension of 64 as it will be decoding now that latent

space that has 64 dimensions. We're going to have another

hidden dimension, so it'll take that next step up to 256 nodes

and then finally have the 784 nodes. And we can create our model that

takes in the encoded inputs and passes out this reconstruction

that we have here. And that's going to be our decoder

model that just as before, we create our outputs by passing in

the encoder model with the inputs as its input into the decoder model. That will be that final output to allow us

to go all the way from the inputs to that final output that we have here. And then the full model, we can just

say the input which is these inputs, all the way out to these

outputs that we just defined. We can look at that full model summary. We looked at the summary before,

now it doesn't show, it's just showing the different models. But now you see,

there's a lot more parameters that it's learning rather than just

what we had here at 50,000. We're up to 217,000 because we

have that 256 node hidden layer in between the inputs and

our encoding dimension. So again, we have our full model just

going from inputs that we define to the outputs we defined. We can compile it using the optimizer. The loss function that we care about,

which metrics we want to track, and this time we're going to run it for

five epochs. So we have the batch size equal to 32, so we're on the gradient every 32

samples that we go through, and we're going again from x\_train\_flat as

the input to x\_train\_flat as that output. Those two will be the same and in the middle it will be coming

up with that encoding layer. So you run this, and this could be for

five epochs and may take a bit of time, so I'm going to pause the video and we'll come back as soon as

this is done running again. So now our model has been fit

to again our x\_train\_flat and our x\_train\_flat,

both as input and output. We can then call full\_model.predict

on our test set and again, that's just going to try to

deconstruct and reconstruct or bring it down to that latent space, and

then reconstruct that original image. So run that and then again, get that mean

squared error of that reconstruction, passing in that new decoded image as

well as the original x\_test\_flat. And when we run that, we see that we

get a score of 84.3, so it did better than PCA now, now that we've done 5

epochs and done this deeper network. Now let's see if we can

improve even further. We notice that so

far we're only using so many epochs. And we want to see if maybe we introduce

further training as we see the loss continue to go down and

it hadn't plateaued yet, and accuracy continued to go up. See if we can actually get a better

performance as we increase the number of epochs. So this function that we have here will

actually just be putting together all the steps that we have here above. So this you can imagine was

just copied and pasted. So we're not going to go through

that again, into this function and then we just say the number of

epochs that we want to run through. And this is the part that's different,

wherefore I arranged the number of epochs, we keep fitting the model

to the training set. And recall that if we're

not reinitiating the model, then what we're doing when we say only one

epoch here for the entire range of epochs, it's going to pick up the training

where the last one left off. So we'll see the results for

just one epoch, for two epochs for three epochs, and so on. And with that, every single time,

we'll get the decoded images. Get our reconstruction loss. We will append that to this list

that we have initiated up here. And we will say the reconstruction loss

after each number of epochs is so on and so forth to see whether or

not we continue to improve. So we initiate that function and

then we're going to run this now for ten epochs, right? Number of epochs is the only

argument we have available. We run this for ten epochs and

this will take just a bit of time to run. We had defined epochs

take x amount of time. This will take about double the amount, so we'll come back as soon

as that's done running. So now we see the results, after ten

epochs, we're able to, for the most part, continue to reduce that reconstruction

error, that mean squared error. We do see that it is not

monotonically decreasing. At some point it seems to waver, but

you see towards the end of ten epochs, we get it down to the mid 60s compared to

that higher number that we had earlier. So those extra epochs, even though if we look at the accuracy

score, it may not be improving that much. We do see that it continued to decrease

that mean squared error of that reconstruction score. So that closes out our video here

in regards to just working with the autoencoders. In the next video, we'll introduce

variational autoencoders and how we can leverage those. All right, I'll see you there.

Welcome back to our

lab on autoencoders. In this video, we're

going to be working with variational

autoencoder specifically. If we recall,

variational autoencoders are specifically creating

that latent space, that's now going to be a mean and a standard deviation that signifies a normal distribution. From that normal distribution, we'll sample values and then

pass those values through to our decoder to try and

reconstruct our images. Again, just at a high level. The first neural network is going to be that encoder where we're going to predict two vectors for each

one of our images, where these two vectors

will be interpreted as the mean and the

standard deviation. That can be sampled and used to pull out something from

the normal distribution. The second neural network will

be the decoder that takes the results of our encoder to reconstruct our

original image. Then the entire system is

going to be trained using back propagation

at each iteration. If we recall what the

regular autoencoders are input and output

are the same. That's still going

to be the case. We're still going

to try and minimize the amount of air when we're trying to reproduce that image. But now we're also adding on another penalization

term if that values, those two values are not going to be standard

normal values. They're not going to

be zero and one for our mean and for a

standard deviation. That's going to be

that KL divergence that we discussed in lecture. We're going to import many of

the libraries that we need. Many of these we've

seen, some of them are new and I'll touch

on the new ones as we get into each one of different portions

within our code here. Recall that when we're

trying to create our input from the

encoded output. That encoded output again is going to be that

two-dimensional vector, and that vector is going to have a mean and a standard deviation. In order to create that to transform that into

input for the decoder, we take those means, add on the standard deviation, then multiply it by

some random epsilon, where epsilon is going to

be some value sampled from the normal distribution

randomly with a mean of zero and

standard deviation of one. That's how they'll

be some variation in what's going to be

output each time. To create that sample, we're going to create this

sampling function here. We pass in args. Those args are actually going

to be the Mu and log Sigma. It's going to be that

output from the encoder. We're actually going

to add this on later on and we'll see how

we do this in Keras. Add this onto our

actual encoder model at the backend of

our encoder model so that we can produce

a single vector. We have our Mu and log Sigma, those are our args. We unpack them here. We then set the epsilon equal to just a random value

with a mean of zero, standard deviation of one. That's going to be the default when you call random normal. We wanted the same shape as the Mu so that we

can add that on. We'll see that in just a second. Add that onto the Mu plus the Sigma according to this

formula that we have here. We then set Sigma equal

to the log of Sigma, e to the log of Sigma so that

we just have Sigma itself. If we recall the output in order to ensure that it

always is positive, we actually output

a log of Sigma. Now we need to transform

that back into Sigma. Then from there, once we have our Mu and our epsilon

and our Sigma, we can produce random

values by taking Mu plus Sigma and multiplying

that by that epsilon, that randomly sampled value. That's going to be our

sampling function. Now, we're going to create

our actual encoder network. We have our inputs

same as before. We have our dense hidden layer, which is going to

have hidden dim, so that 256 dimension, so there's going to

be some hidden layers will be of a deeper network. It has its inputs, which we define here that

are going to be passed in. Again, we're using

the functional API. Then that x is going to be

used as input into both getting the mean as well as the log variants of our z value. We can use that x and

we wouldn't be able to do this as easily with

a sequential API. That's why we use

a functional API. We can pass in the

x to a dense layer here and a different

dense layer here so that we're outputting

two different values with that input. Then to get that final output

that we're looking for, we're going to call Lambda. That Lambda function

allows you to pass in your own

created function, which we created

above that sampling. Lambda sampling and the

input for that sampling is going to be this list which

is z mean and z log there. That corresponds to our args, which we're able to unpack up

here into Mu and log Sigma. Then at the end of

our encoder model, we pass in our model that

input is going to be input. The output, we're

actually going to output three different values. We're going to output the z mean so that we can track that. We're going to output

the z log there. Then we're going to output

that actual z value, that actual sample value as well. Now we've set up

the encoder model. Then we're going to build

out our decoder model. That's going to take an

input of the shape of our latent dimension,

those two dimensions. We're then going to expand it. Similar to how we did

with the auto-encoder, where we first shrink it down with a encoder and

then we expand it with the decoder here, we're

doing that again. We're expanding that out up to the hidden dimension first, up to 256, passing in

those latent inputs. Then finally, that

final output will be the dense 784 node layer, so that'll be the

same dimensionality as our original image. Then we can just say

that the model is going to take in

those latent inputs, then pass out the

outputs we defined, by passing that into the model. Then to get the full model, we just do decoder model and we pass in the

encoder inputs, but this time, we only want

as input that third value. Recall that we actually

are outputting in the encoder model

three different values. What we really want

to pass through into our decoder network

is just that third value, which is why we specify too here. Then once we have our outputs, we can create our full model

which goes from those inputs defined all the way up here in our encoder model to the outputs that we

just defined up here. Now just to take a quick dive in, we can look at our model. We're going to look here at each one of the different

layers that are involved. Within that layer, layers are defined a

little differently than probably what we are used

to in regards to working with our neural networks

that have multiple layers, but rather, the layers are the models that

are being used. We have the encoder

input and that's just going to have no weights

that are being learned. It's going to be

of dimension 784. None is just however many

samples we want to pass in. Then in our actual encoder layer, outside, just the input, we have the input being

784 and the output, as we discussed, being three

two-dimensional vectors. One being the means, one being the Sigmas, and one

being those sampled values. Then within this encoder, still in layer two, our first weights are going

to be our dense layer, which gets us 784 by 256. Our next weights are going

to be the bias term, which is going to be

256 different weights. We're then going to get

our actual mean values, which is 256 by 2. So those are the weights needed

there. Our weights four. Now, when it says

Weights 1, Weights 2, Weights 3, Weights 4, that's

not the number of weights. That's just representing weights. Number 1, weights

number 2, and so on. The same way we did, layer

number 1 and layer number 2. Then we have our mean bias and that's just going

to be two values. Then we're going to

have similar for the log variance with 256 and 2. Then in Layer 3, we have our

decoder starting off with the input of two, output of 784. We have all the intermediate

weights similar that buildup back to that

784 that we discussed. Now again, if we

want to start our go ahead and actually compile this model that we built out, we need to actually reproduce that loss

function that we need. I'm going to pause here now that we've built out the

variational autoencoder. The next video, we'll

talk through how to actually build out

this loss function appropriately so

that we can compile our model that we just

built. I'll see you there.

So as discussed at the

end of the last video, we're going to need a

different loss function for our variational autoencoder. The second part of

that loss function, so we had the first one for

the reconstruction error, which is the same as

the autoencoders. The second portion, which is specific to variational

autoencoders, is what we have here, and we discussed this in lecture, how we predict log Sigma

because predicting Sigma directly could result in a negative value and doesn't make sense to have a

negative variance. Then the fact that the cost

function has two components, both of which penalize

us for having results that deviate from that

standard normal deviation. So if log of Sigma

is far from zero, we can use this portion E\_x minus x plus 1 being minimized

at x equals 0, and that's equivalent

to what we have here in the first portion of

the KL divergence. Then the second part simply

penalizes the Mu value, that mean value from

being far away from zero. Then again, the other part of the loss function

is just going to be that reconstruction error. So our exercise here

was to actually create this loss function so that

we can pass it through and optimize using this

loss function, minimizing this loss

function that we have. So we do this first by starting with the

reconstruction loss. We're going to just

multiply it by 784 so that we're not working

with the average value, but rather the full value rather than an average

over all the pixels. So we're just using this

binary cross-entropy. So far, we've been parsing in that string as our loss function. But if you look at our

imports that we had earlier, we actually pulled in

binary cross-entropy, and that's available

from Keras.losses. So we pull that in and

we're able to say we want that binary cross-entropy for the inputs and outputs, and we define that as

our reconstruction loss. So that's one portion

of our loss function. The other portion

is just going to be this formula that

we have here above. We're going to pass

in the Z log there, which is going to be

one of our outputs that we specified in our encoder, as well as the mean

value that we have here. Sorry, we have the Z

log they are there, and then finally that

square of the mean value. Then we're just going to sum that kl\_loss on that final axis, and then the total

loss is going to be the mean of the reconstruction

loss plus that kl\_loss. So that reconstruction

loss that we defined here, plus the kl\_loss that we

just defined down below, which is this function that we discussed both in lecture

and here in the notebook. So we're going to

run those cells, and then once we have

that total\_vae\_loss, we can add onto

our model by using this add\_loss functionality

within Keras. We can call add\_loss and

pass in this total\_vae\_loss, and then we can just

compile and specify our optimizers as well as the

metrics you want to track. Once we do that, we can

look a bit at our summary. Gets a bit intense with

everything that we're doing, but we don't have to worry

too much about it just here. Then we will fit that

to our x\_train\_flat. So this is training

our full model. The number of epochs we

defined above as just one. I'm going to pause the video here as it'll take about

10 seconds to run, and we'll come back as soon

as that's done running. So that should have been a quick second to run or a

quick 10 seconds, and now we want to look back at the reconstruction error for our new model that

we're working with, the variational

autoencoder model. I want you to think whether

that reconstruction should be higher or lower than the original autoencoder without maybe reading what

we have here below. I'm going to run this here, and we're going to

see that we have a much higher

reconstruction error. The reason being that

this latent space is built more for interpretability, that is, sampling

from a distribution, rather than being perfectly reconstructed to those

original images. As now we're sampling rather than taking the direct latent

space that we learned. Now I want to plot out the

latent space that we had. So we have our models, we're going to create a tuple that's just

the encoder model and the decoder model so that we can pass that into our

function later, and then our data is also

going to be this tuple, which is x\_test\_flat

and the related y\_test. This is pretty large.

I'm going to summarize. I'm not going to go through

every line of code here, but I am going to summarize what the data is actually or what the functions are

actually trying to do. We have our encoder

and decoder as well as our x\_tests and y\_test. The first portion of our

function is going to be to actually plot out in two-dimensional space

as we did with PCA, each one of our

different numbers. That's going to be

the first portion. In order to do that, we need to get our z value. We're going to get each of

the predicted z values. With those z values, we're going to also get

their related numbers. We're setting the color

and our scatter plot according to that y\_tests

that they're related to, and then we have our

two dimensions of z\_0 and z\_1 in that

two-dimensional space. Those z's are going to be those random samples that we pull out according to what each

one of those digits are. Then on the lower portion, so we're going to actually

plot out two plots here. We're going to

create a grid space ranging from negative

four to four. If you see that our limits here is four passing

into the function, or whatever limits tend to be that we get passed

into our function, here it's negative four to four. We're going to plot

a bunch of values. Here n is equal to 30, so 30 different values

between negative four and four on both the x and y-axis. We'll see for these

different values of x and y, what is going to be and

recall we are with our z's we originally plotted

out in two-dimensions, which numbers they

are closest to, and we'll see this one's

the picture comes out. But now, we're actually going to generate numbers

using the decoder. Let's say our first value is negative four, negative four. Then for negative

four negative four, we're going to pass that

in by using that xi, yi. We're going to pass

that into our decoder, and see what kind of sample

it actually generates. You can think of the value being zero and two or zero and three. Remember, these are

supposed to be samples from something close to a

normal distribution. Now that function

has been initiated, I forgot to run this, we need to make sure that

we have all that defined, and then we're going to

actually plot the results, and this should make what I

just discussed a bit clear. Here, we have the

different groupings. We see for our purple here

which is related to our ones, those tend to be in

the top-left corner which is negative four on the z\_0 for our first dimension and z\_1 being positive four. Then when we look at

the numbers generated, those tend to be ones. Whereas, you see here on the bottom that darker

purple is the zeros, and here it's

generating those zeros. Top right, we have the light green which is associated

with the sevens, and we see that that's predicting or generating

sevens as well, so we can see this

generative process. Then finally in Eexercise 6, we want to train the

variational auto-encoder as well as the autoencoder

for 10 epochs each. Then plot the reconstruction

mean squared error as a function of

the total number of epochs for each one

of these models. See which one seems to

have more potential to continuously learn as it's

given more computing time. We have here, it's going to

start running for 10 epochs. I'm going to continue running. I'm going to talk through what we have here because

all we are doing is then running our variational

autoencoder model for 10 epochs. For the autoencoder, we had that function defined

in maybe video 1, I believe it was. I'm going to scroll

all the way up. Maybe it was video 2 because

video 1 was on PCA where we defined this function

here that trains AE, or autoencoder for a

certain number of epochs. We don't have that for the

variational autoencoder, so all we have to do

is for in range 10, fit that model each

time for one epoch. Again, recalling that

every time you run it, it's going to pick up where

the last one left off. Each of these are going

to take some time to actually plot out. We'll come back when that's done, and then we'll actually plot out the loss functions across

each. I'll see you in a bit. Now, we had our

autoencoder as well as our variational autoencoder

run for 10 epochs each. We can look at the plot over time or over the

number of epochs. We see that there tends

to be a plateau for those autoencoders whereas

a variational autoencoders are continuing to go down. You could probably

run this a little bit further if you wanted maybe 10-15 more epochs for the autoencoder is

a really plateau. We see that the auto-encoder

are fitting exactly and eventually are going

to plateau due to the fact that they're

fitting exactly. Whereas the variational autoencoders continue

to decrease. It will take a little

bit more time to get to that same reconstruction rate, probably never getting

to that same exact with contracting rate as we

see with the autoencoder. Now, that closes out

our notebook here. After this, we're going to

get back into the lecture and discuss a different

generative process, specifically GANs and generative adversarial networks.

I'll see you there.

# Summary/Review

**Autoencoders**

Autoencoders are a neural network architecture that forces the learning of a lower dimensional representation of data, commonly images.

Autoencoders are a type of unsupervised deep learning model that use hidden layers to decompose and then recreate their input. They have several applications:

* Dimensionality reduction
* Preprocessing for classification
* Identifying ‘essential’ elements of the input data, and filtering out noise

One of the main motivations is find whether two pictures are similar.

**Autoencoders and PCA**

Autoencoders can be used in cases that are suited for Principal Component Analysis (PCA).

Autoencoders also help to deal with some of these PCA limitations: PCA has learned features that are linear combinations of original features.

Autoencoders can detect complex, nonlinear relationship between original features and best lower dimensional representation.

**Autoencoding process**

The process for autoencoding can be summarized as:

1. Feed image through encoder network
2. Generate the lower dimension embedding
3. Feed embedding through decoder network
4. Generate reconstructed version of the original data
5. Compare the result of the generated vs the original image

Result: A network will learn the lower dimensional space that represents the original data

**Autoencoder applications**

Autoencoders have a wide variety of enterprise applications:

* Dimensionality reduction as preprocessing for classification
* Information retrieval
* Anomaly detection
* Machine translation
* Image-related applications (generation, denoising, processing and compression)
* Drug discovery
* Popularity prediction for social media posts
* Sound and music synthesis
* Recommender systems

**Variational Autoencoders**

Variational autoencoders also generate a latent representation and then use this representation to generate new samples (i.e. images).

These are some important features of variational autoencoders:

* Data are assumed to be represented by a set of normally-distributed latent factors.
* The encoder generates parameters of these distributions, namely µ and σ.
* Images can be generated by sampling from these distributions.

**VAE goals**

The main goal of VAEs: generate images using the decoder

The secondary goal is to have similar images be close together in latent space

**Loss Function of Variational Autoencoders**

The VAE reconstruct the original images from the space of a vector drawn from a standard normal distribution.

The two components of the loss function are:

* A penalty for not reconstructing the image correctly.
* A penalty for generating vectors of parameters µ and σ that are different than 0 and 1, respectively: the parameters of the standard normal distribution.