In this section are going to introduce

our first deep learning model that's going to be used for

unsupervised learning Autoencoders. Now let's discuss the learning goals for

the section. In this section will start off with

a review of non deep learning based techniques for

data representation such as PCA. And how we can use that to condense

our original data set into a smaller representation

of that same data set. After that we will discuss how

Autoencoders Leverage Neural Networks to also come up with lower dimensional

representations of our data. And then finally will discuss

a bit on how to describe those use of trained autoencoders in

order to actually generate images. Now, autoencoders will be our first

time looking at deep learning from an unsupervised learning vantage point. Now the goal of Autoencoders is going

to be to use those hidden layers in our neural networks to find a means of

decomposing and then recreating our data. And we'll see how this

is done in just a bit. And this proves to be powerful for things such as dimensionality reduction

and fighting that cursive dimensionality, as we've seen in prior courses

when we were working with PCA. With that, this dimensionality reduction

can be powerful for pre processing for classification and identifying only the

essential elements of our input data while filtering out the noise

within our data set. Now as motivation, let's say we want to find weather 2

images are similar to one another. So we have two pictures here. Each of the kangaroo and may not want to know whether each of

these images are similar to one another. Now one auction ends. I'll say right off the bat that

we probably don't want to do. Is to look at the pixel wise

distance between these two images. So if we look at the image, the left and we look at the pixels there

in the top right corner. Compared to the pixels in the top

right corner of the right image, we see that these two

are clearly not the same. And we can even see this with our own

eyes that these are very different. If we were just looking at

these particular pixels. And the goal would be in our problem

here is that if we just look at the pixels as a whole, will only be

able to see the placement of the color scheme of the brightness etc and

not the actual content of our image. So the goal would be to find some type

of representation that captures that actual content within the image. So if we think about the image here,

to the left were looking at Brown Kangaroo fur, kangaroo ears,

a kangaroo nose a beige backgrounds. And those are going to be content within

that left image and that right image we have round kangaroo fur kangaroo ears,

kangaroo nose and a green background. So many similarities and

maybe just the difference in background. And the idea would be

that with autoencoders, and we think about what we've

learned with deep learning. How is able to find each of those

features that make up the image with something like autoencoders that

leverages these deep neural networks, we should be able to

capture these essences. Now before getting into the actual

method of working with autoencoders, I'd like to introduce here a business

application for what we just discussed and how autoencoders can be

used in business practice. We think here of electronic

components within a production line. Some defects might be imperceptible for

the human eye or difficult to scale, given if we're looking at images

of each of these chips and the amount of pixels in

each of these images. And we have to look across millions of

chips may be too difficult to scale. We may want to then reduce

the dimensionality of those pixels so we can look at these defects at a lower

dimensionality when we're comparing whether or not they're similar to one

another, whether or not there's a defect. So one approach to this. To being able to detect these differences

at scale would be to use PCA to reduce the dimensionality of our features,

which here are going to be pixels. So that each component is some linear

combination of our principle components. So we start off here with a pixel vector. It's here going to be RGB, so

we have the three channels and for each one of those channels. We had the height and width. We use PCA. And again, we're able to create a linear

combination of our principle components. If you recall from courses past and reduce the number of dimensions

that we are working with. Now just a quick reminder how PCA works

before we get into autoencoders and to motivate autoencoders. PCA works in reducing the mentions of our

original data and the goal within PCA is defined as the dimensions that capture

the most variance from our original data. So as an example,

if we're working with just two dimensions, you can think of this again,

is just two features. The directions of the arrows will

represent the principle components or the directions representing

the most variance within our data. And the length of these arrows correspond

to the amount of variance in the original data that is explained. So we see the diagonal pointing to

the upper right, accounting for more of our overall variance,

the one pointing to our upper left. And we see that each one of these

arrows will actually be composed of a combination of both X1 and X2. And now just saying which

single access accounts for the most variation won't

be what we're doing, but rather some combination of those creating

new axes coming from that X1 and X2. Now there are going to be

limits to working with PCA, and why would want to move to

something like autoencoders. The main thing is that learn features will

have to be some type of linear combination of our original features. When in reality there may be some complex

nonlinear relationship between those original features and the best lower dimensional

representation of those features. And finally, how we define the best

representation can be different depending on what our problem is. So that closes out our video,

just motivating the use of Autoencoders in the next video, we'll pick up and

dive into how autoencoders actually work. All right, I'll see you there.

Autoencoders are going to be a neural network

architecture that will force the learning of some lower-dimensional

representation of our data, and that's commonly

used for images. The way that autoencoders work

step-by-step is that they will have that same value as both input and output

as we see in image. We'll then feed this input

through our encoder network, represented here as that densely

connected nodes in blue. From this encoding step, we're going to be able to reduce the lower-dimensional

embedding of our original data and that's going to be what

we're actually looking for, that lower-dimensional

representation of our data, and that will be here in

the middle of our network. Then finally, that embedding will be fed through the

decoder network, which we have there through

to our final output, which again is going to be

the same as our input values. That decoder portion is

meant to go about creating the reconstructed version

of that original data. Once we have that reconstructed

version of the data, we can go about

computing the loss between that

reconstructed version and that original input and use that to train

our actual network. We take that loss function, as we do with any neural net, and use it to update the

weights within our network, and that's going to be that feedforward and

back-propagation steps that you're already used to. The result will be that

in that middle portion, given that if you've noticed the nodes be shrinking

and so we start with three nodes and then two

nodes and it could be 100-10, whatever it is. As those amounts of nodes shrink, in that middle and then again, it shrinks and then

expands back to that reconstructed version

in the decoder step. In that middle where we

have just two nodes, we're going to have that lower-dimensional

representation of our original data. We can use an autoencoder

to find image similarity, because we feed two images

through the encoder network, and we can calculate

the similarity score of their latent vectors of this lower-dimensional

representation. That allows us to actually see at a version that

we'd be able to scale, because they won't

require as much memory, how similar two images are. We can always use that

decoder portion of the network to map

those vectors from our lower-dimensional space to the full-dimensional

space of our images. The point being that

this allows us a means of compressing and then

decompressing our data. Now, another use of

the decoder model is to actually work as

a generative model. In order to properly do this, we would probably

want to actually work with variational

autoencoders, which we'll discuss

in the next lesson. But even with variational

autoencoders, this isn't going to be commonly done, this generative model, due to the fact that in order

to get reasonable results, some deep convolutional

architecture is generally going

to be required. Even with that,

generally speaking, the results of that

image generation will generally be

inferior to that of GANs, which we'll learn not in the next lesson but in

the lesson afterwards. Autoencoders can have a wide variety of

enterprise applications. They can be powerful for preprocessing and reducing

the dimensionality of our data prior to learning

some classification model, it can be powerful for sending information in a compressed form, as well as retrieving

such information, may use for anomaly detection, as we discussed with

the chip images, can help with machine

translation as we're generally working in very

high-dimensional space, if we're doing

machine translation, can be powerful for

image-related applications such as generating images, denoising or taking fuzzier

images and sharpening them, as well as processing and

compressing as we discussed, and for drug discovery, popularity prediction

of social media posts, and sound and music synthesis. It can help find the

key components that are key to each one of these different domains and help identify those key

components that may be key to the model

in drug discovery, popularity of a social media post or sound and music synthesis. One last note. While

most autoencoders will use deep layers, autoencoders are often

going to be trained on just a single layer each for the encoding

and decoding step. An example of working with a deeper network is going to be using

sparse autoencoders, which essentially allow for

those deeper networks but only certain nodes will be

firing within those networks. This has been used

successfully in things such as recommender systems.

Just to recap. In this section we discussed non-deep learning-based

techniques for data representation and reminded ourselves how PCA will use a linear combination of our

original features to come up with a representation

that maintains as much of the variance as possible from our original dataset. We then discussed how

autoencoders work with the encoder portion coming up with a condensed

version of our data, which can then be reconstructed using

our decoder network. Then finally, we discussed

a bit how trained autoencoders can be used

to generate images. That's especially

true using something called the variational

autoencoders, which we'll discuss in this upcoming video.

I'll see you there.

In the next videos, we'll introduce the concept of a variational autoencoders, which will work similarly to the auto-encoders that

we just discussed, except now that latent space, that hidden space that we're

trying to represent is going to be described by distribution rather

than exact figures. Now, let's go over the learning

goals for this section. In this section, we're

going to cover how variational autoencoders

work and how we can come up with this

new latent space represented by some distribution. Then, we'll discuss variational

autoencoder loss functions and that will provide us with some intuition as to how variational autoencoders

are used and optimized. With variational

autoencoders, we'll still be generating that

latent representation, or again, that compressed

representation of the data that encodes

those similarities. Similarly, we can reconstruct these to generate new samples. Some important features of variational autoencoders

will include: rather than the data being represented by just a

single set of vectors, the values of the data in that latent representation

will now be represented by a set of normally-distributed

latent factors; and now rather than the encoder coming up with a

particular value, instead generates the parameters of our normal distribution, namely the mu and sigma, or the mean and the

standard deviation; then using variational

autoencoders and the fact that we are going to be sampling from a given distribution rather

than some fixed values, we can actually

generate new images. The goal of variational

autoencoders will be to generate images using the

decoder portion of our network. Starting off our encoded

latent vector will now be represented by some

normal distribution. The parameters for that normal distribution

will be learned by the encoder portion

of our network within this variational

autoencoder, and then fed through to our learned decoder portion

to produce the images. A secondary goal that will

come along with that is that similar images will be close together within the latent space. As we'll see in our

notebook later on, if we're looking at

hand-drawn values between zero and nine, the latent space for all the zeros will be

close to one another, for all of the fives will

be close to one another, and so on and so forth. Let's walk through

the steps of how variational

autoencoders will come up with this latent space represented by a

normal distribution. The first step will still be to pass through a network

with some bottleneck, so reducing the number of nodes as we did with the

regular autoencoders. But now at step two, we are going to be learning a mu and a sigma for each value that are meant to represent a normal distribution from

which values can be samples. For example, here, we may end up coming up with

the vectors that we see, which are for the

mu 0.7 and negative 0.6 and then one and 0.6

for the sigma values. In the next step for our

variational autoencoders, we combine these two

values into one vector and add on some white

noise with a mean of zero and a standard

deviation of one. Using our example from before, we can come up with

this vector at the end of 2.01 and 0.54, by adding the mean that mu plus the sigma multiplied

by our noise term. Those are the vectors

that we see here at the bottom that will tell us what is going to be some

sample from our distribution. This randomly sampled

vector is then fed through our decoder

network in step four. We then can produce our

reconstructed image. That's how we walk through this variational autoencoder

with the mu and sigma. In the next video, we're going

to touch on a high level, some of the math that makes the variational

autoencoders work and are specific to this variational

autoencoder in general. All right, I'll see you there.

In this video, I

want to start off by discussing the

loss function of the variational

autoencoder compared to that of what we would do

with just the autoencoder. Our goal will be to reconstruct the original image as we

did in our autoencoders. Now, we know that a

major difference is that the variational autoencoder

will be reconstructing a vector drawn from a

standard normal distribution. With that in mind, we're going to have

two components to our variational

autoencoder loss function. First, we have what we had

for the normal autoencoders, which is just the error

measuring how far off our reconstructed image was

from that original image. The second part of loss function will be a penalty

associated with generating vectors of the parameters Mu and Sigma that are both

different from zero and one. Mu being different than zero and Sigma being

different than one. As our goal will be

to balance between that low reconstruction penalty as he did with autoencoders, while keeping our parameters as close to the standard normal

distribution as possible. This comes down to the

pixel-wise difference between the reconstructed

and original image for that first component. In order to calculate this, we can use a loss function

such as mean squared error to see the distance between the reconstructed image

and the original image. Then again, the second component will be the difference

between the vectors produced by the encoder and the parameters of a standard

normal distribution. How do we go about calculating that second component

of our loss function? For this, autoencoders will use something called KL Divergence between the data that we generated and the standard

normal distribution. Here we have, for example, our Mu values versus

a Mu of zero, and then we're actually going

to take the log of Sigma, and if we think about our Sigma, the ideal Sigma would be one, the log of one would be zero. We're going to be comparing the log of our values

versus zero again, since again, the log of

one is going to be zero. We use the log to ensure

that we end up with strictly positive values as a negative value for variance doesn't

really make any sense. Now the actual KL

Divergence formula will be what we have here, where e\_log of Sigma minus

the log of sigma plus one is going to penalize the

Sigma for a string from one. We can know if Sigma is one, then log of Sigma is zero, and if we replace log of

Sigma with zero here, we would see that this

comes out to 1 minus 1, and thus minimized at

Sigma equal to one. Then for Mu, obviously, Mu squared will be minimized when that Mu

value is equal to zero, because if it goes

a little bit to the right to 0.5

or even up to one, then Mu squared would be one. If it goes a little to

the left to negative one, again, you'll have

a value of one. It'll be minimized when

Mu is equal to zero. In regard to that Sigma portion, we can somewhat see

graphically if we imagine subtracting that orange

line from the blue line. That again, e\_x minus x plus 1, which is similar to our Sigma portion that we discussed in our KL

Divergence formula. That again, is going to be

minimized at x equal to zero. Now, a note on KL

Divergence is not technically necessary to include this component in

our loss function. But the reason that we do

like to include it though, is that it helps generate

a desired latent space, or visually similar

images are going to be close together within

that latent space. As an example, what

we have here is a variational autoencoder trained with two-dimensions in

the Mu and Sigma vectors. Because this is a

generative model, we can scan the latent plane and sample points at

regular intervals and actually generate

corresponding digits given what we sampled for

each one of these points. As long as our sampled

values are close within that

lower-dimensional space, which is represented

by our Mu and Sigma, they will generate

similar images. That's why you see sevens up to the left and some

nines in the middle, none of these were actual

images within our data set, but using our

variational autoencoders we're able to generate

these randomly. Now, just to recap, that will close out our section here on variational autoencoders. We went through the basics of how variational autoencoders work and how they differentiate from

regular autoencoders, and that they produce

a probabilistic means of describing our latent space. With that, we also discussed the loss-function used

and why adding on KL Divergence to measure

the distance from the normal distribution

can be very powerful. Now, let's take a look

at some actual code as to how we can actually build our autoencoders as well as variational autoencoders

of our own. All right, I'll see you there.