This is written using Python and TensorFlow and an API in TensorFlow called keras. Keras makes it really easy to define neural networks. A neural network is basically a set of functions which can learn patterns. Don't worry if there were a lot of new concepts here.

In keras, you use the word dense to define a layer of connected neurons. There's only one dense here. So, there's only one layer and there's only one unit in it, so it's a single neuron. Successive layers are defined in sequence, hence the word sequential.

There are two function roles that you should be aware of though and these are loss functions and optimizers. This code defines them. I like to think about it this way. The neural network has no idea of the relationship between X and Y, so it makes a guess. Say it guesses Y equals 10X minus 10. It will then use the data that it knows about, that's the set of Xs and Ys that we've already seen to measure how good or how bad its guess was. The loss function measures this and then gives the data to the optimizer which figures out the next guess. So the optimizer thinks about how good or how badly the guess was done using the data from the loss function.

As the guesses get better and better, an accuracy approaches 100 percent, the term convergence is used. In this case, the loss is mean squared error and the optimizer is SGD which stands for stochastic gradient descent.

The training takes place in the fit command. Here we're asking the model to figure out how to fit the X values to the Y values. The epochs equals 500 value means that it will go through the training loop 500 times. This training loop is what we described earlier. Make a guess, measure how good or how bad the guesses with the loss function, then use the optimizer and the data to make another guess and repeat this. When the model has finished training, it will then give you back values using the predict method.

Machine Learning depends on having good data to train a system with. In this video you saw a scenario for training a system to recognize fashion images. The data comes from a dataset called Fashion MNIST, and you can learn more about it and explore it in GitHub [here](https://github.com/zalandoresearch/fashion-mnist). In the next video, you’ll see how to load that data and prepare it for training.

Here you saw how the data can be loaded into Python data structures that make it easy to train a neural network. You saw how the image is represented as a 28x28 array of greyscales, and how its label is a number. Using a number is a first step in avoiding bias -- instead of labelling it with words in a specific language and excluding people who don’t speak that language! You can learn more about bias and techniques to avoid it [here](https://developers.google.com/machine-learning/fairness-overview/).

**Sequential**: That defines a SEQUENCE of layers in the neural network

**Flatten**: Remember earlier where our images were a square, when you printed them out? Flatten just takes that square and turns it into a 1 dimensional set.

**Dense**: Adds a layer of neurons

Each layer of neurons need an **activation function** to tell them what to do. There's lots of options, but just use these for now.

**Relu** effectively means "If X>0 return X, else return 0" -- so what it does it it only passes values 0 or greater to the next layer in the network.

**Softmax** takes a set of values, and effectively picks the biggest one, so, for example, if the output of the last layer looks like [0.1, 0.1, 0.05, 0.1, 9.5, 0.1, 0.05, 0.05, 0.05], it saves you from fishing through it looking for the biggest value, and turns it into [0,0,0,0,1,0,0,0,0] -- The goal is to save a lot of coding!

What would happen if you remove the Flatten() layer. Why do you think that's the case?

You get an error about the shape of the data. It may seem vague right now, but it reinforces the rule of thumb that the first layer in your network should be the same shape as your data. Right now our data is 28x28 images, and 28 layers of 28 neurons would be infeasible, so it makes more sense to 'flatten' that 28,28 into a 784x1. Instead of wriitng all the code to handle that ourselves, we add the Flatten() layer at the begining, and when the arrays are loaded into the model later, they'll automatically be flattened for us.

Another rule of thumb -- the number of neurons in the last layer should match the number of classes you are classifying for. In this case it's the digits 0-9, so there are 10 of them, hence you should have 10 neurons in your final layer.

We're asking keras to generate 64 filters for us. These filters are 3 by 3, their activation is relu, which means the negative values will be thrown way, and finally the input shape is as before, the 28 by 28. That extra 1 just means that we are tallying using a single byte for color depth. As we saw before our image is our gray scale, so we just use one byte.

This next line of code will then create a pooling layer. It's max-pooling because we're going to take the maximum value. We're saying it's a two-by-two pool, so for every four pixels, the biggest one will survive as shown earlier