Table of Contents

[**A Primer in Machine Learning:** 2](#_Toc145749962)

[**The “Hello World” of Neural Networks:** 2](#_Toc145749963)

[**An Introduction to Computer Vision:** 4](#_Toc145749964)

[**Writing Code to Load Training Data:** 4](#_Toc145749965)

[**Coding a Computer Vision Neural Network:** 5](#_Toc145749966)

# **A Primer in Machine Learning:**

Coding has been the bread and butter for developers since the dawn of computing. We are used to creating applications by breaking down requirements into composable problems that can then be coded against.



Rules are expressed in a Programming Languages, and data can come from a variety of sources from local variable all the way to databases. Machine Learning rearranges this diagram where we put answers in data in and then we get rules out. So instead of us as developers figuring out the rules, what we will do is we can get a bunch of examples for what we want to see then have the computer figure out the rules.



A “Neural Network” is just a more advanced implementation of machine learning and we call that “Deep Learning”. Fortunately, it is actually very easy to code. So, we are going to jump straight into deep Learning.

# **The “Hello World” of Neural Networks:**

Let’s take a look at a set of numbers, and see if we can determine the pattern between them:

The answer is:

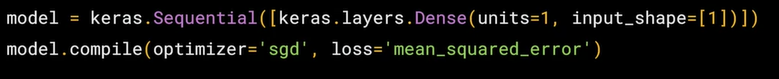
The process of finding this answer in your head is similar to basics of Machine Learning.

Let’s look at it in Code now. This is written using Python and Tensorflow and an API in Tensorflow called “Keras”.

*Note: Neural Network is basically a set of functions which can learn patterns.*



* In Keras, you use the word “Dense” to define a layer of connected neurons.
* Successive layers are defined in sequence (Sequential)
* You define the shape of the input.
* A lot of mathematics needed to use Machine Learning, is implemented in Tensorflow and Keras functions.
* There are two function roles that you should be aware of:
  + **Loss Functions**
  + **Optimizers**

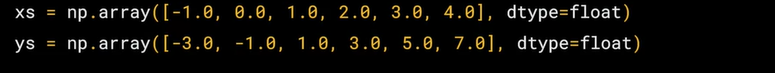


The Neural Network doesn’t have any idea what the pattern is in our set. So, it takes a few guesses. The “loss function” measures how good or how bad its guess was. Then, it gives the data to the optimizer which figures out the next guess.

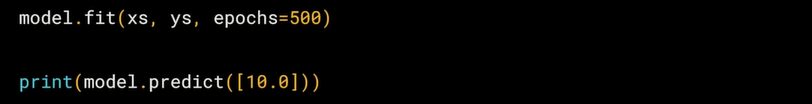
When the guess keeps getting better and better, and reach a 100% accuracy, the term “convergence” is used.

*Note: “sgd” stands for “Stochastic Gradient Descent”.*

* The next step is to represent our Data:



* The “**training**” takes place in the “**fit command**”. Here, we are asking the model how to figure out to fit the “X values” to “Y values”. The “epochs” value is the loop number.
* When the model has finished training, it will then give you back values using the “**predict method**”.
* If you give the predict method the number 10, it will not give you the number 19 you expect. There are two reasons for this:
  + First: You trained the model with very little data. There’s a high probability that , but the neural network is not positive. So, it will figure out a realistic value for Y.
  + Second: When using “neural networks”, as they try to figure out the answers for everything, they deal in “*probability*”. You will see that a lot and you will have to adjust how you handle answers to fit.



# **An Introduction to Computer Vision:**

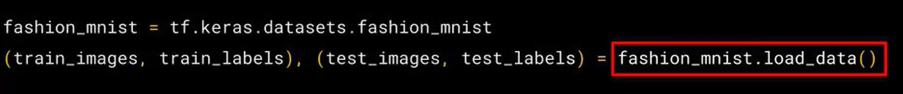
“Computer Vision” is the field of having a computer understand and label what is present in an image.

In this section, we will be working on a dataset called “Fashion MNIST”, which includes 70k images in the size of 28x28, 10 Categories. The images are in gray scale, so the amount of information is reduced.

*Note: When the scale (size) of images are smaller, the computer has less processing to do.*

# **Writing Code to Load Training Data:**

What will handling this look like in code? The last time you had six pairs of numbers, so you could hard code it. Now, you have to load 70k images, so there will be a bit of code to handle that. Fortunately, it is still quite simple because “Fashion-MNIST” is available as a dataset with an API call in Tensorflow.

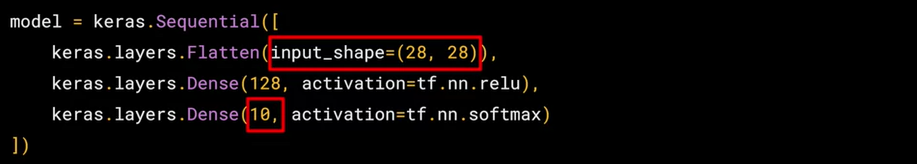


What are “Train images, Train Labels, Test images, Test Labels”? When building a neural network, it is a good strategy to use some of the data to **train** the neural network, and similar data that the model hasn’t seen, to **test** how good the neural network is.

*Note: The computer will initiate a number to each image, such as number 09 for a boot. This is because computers work well with numbers instead of characters and also, it helps us to reduce bias.*

# **Coding a Computer Vision Neural Network:**

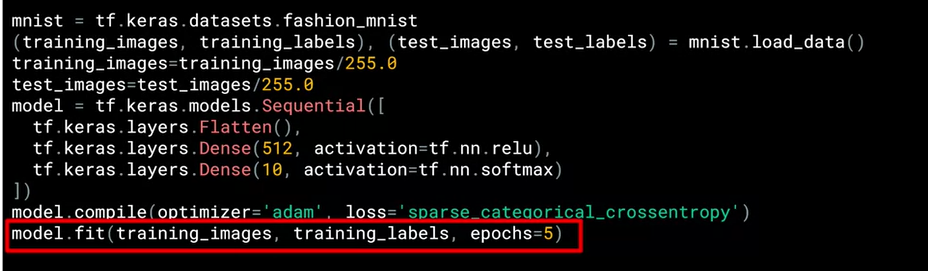
Last time, we had a sequential of one layer. Now, we have three layers in our Neural Network:



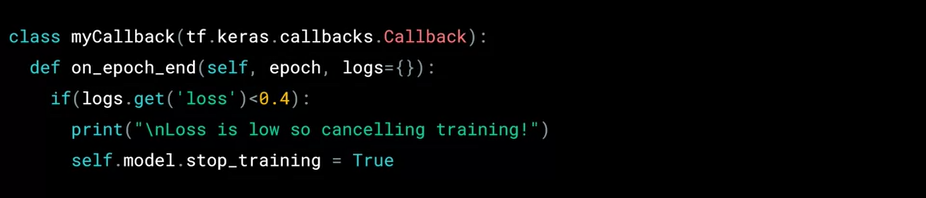
The important things to look at are the first and last layers.

* The last layer has **10 neurons** in it because we have 10 classes or categories in our dataset. These should always match.
* The first layer is a **flatten layer**, with the input shape be 28x28, similar to image sizes in our dataset. **Flatten** takes this 28x28 square and turns it into a simple linear array.
* The interesting things happen in “middle layers” or “hidden layers”, which has **128 neurons** in it:
  + Think of these as variables of a function, maybe call them
  + There exists a rule that incorporates all of these that turns n-values of a picture into a number.
* How can we stop training when we reach at a certain point that we wanted to be? Why should we always add a specific number of epochs?
  + The good news is that the “training loop” does support callbacks. So, in every epoch, you can callback to a code function, having checked the metrics. If they are what you want to say, then you can cancel the training at that point.

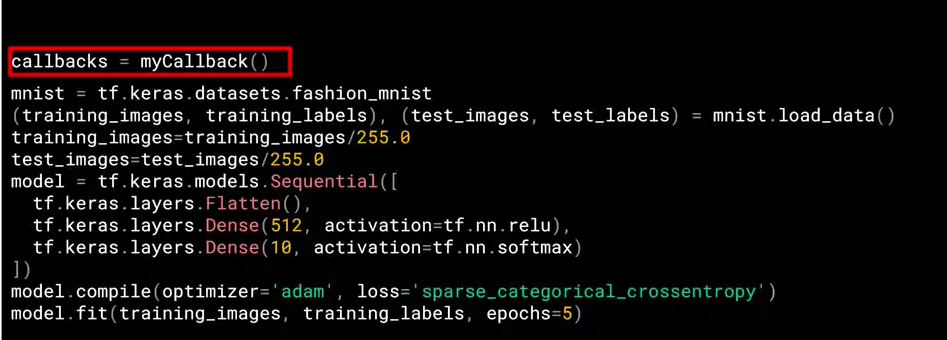
*Note: The “model.fit function” executes the training loop.*

**

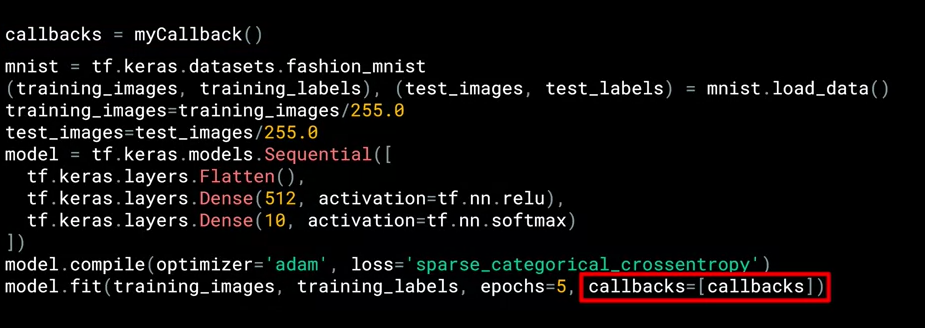
* What we now do is write a **callback** in python:



* It is coded in a separated class, but it doesn’t need to be in a separate file. In this, we will implement a function called **“on\_epoch\_end**” which gets called by the callback whenever the epoch ends. It also sends a “**log object**” which contains lots of great information about the current state of training.
* Now, let’s make two modifications in our previous code:







# **Convolutional Neural Networks:**

In our previous example, when you look at the samples, you realize that there are many wasted space in each image. While there are 784 pixels, it will be interesting to see if there was a way that we could condense the image down to the important features that distinguish what makes it a shoe or something else.

That’s where “Convolutions” come in. If you have ever done any kind if “image processing”, it usually involves a filter and passing that filter over the image in order to change the underlying image.

It does something like this: “For every pixel, take its value and look at the value of its neighbors. If our filter is , then we can take a look at the immediate neighbor, so that you have a corresponding three by three grid. Then to get the new value for the pixel, we simply multiply each neighbor by the corresponding filter.

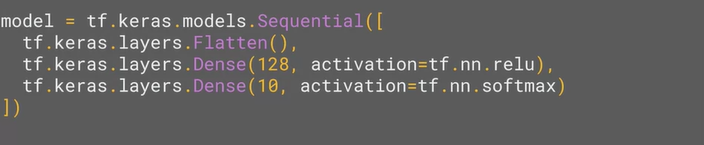
Repeat this process for each neighbor and each corresponding filter value, and would then have the new pixel with the sum of each of the neighbor values multiplied by the corresponding filter value, and that’s a “**convolution**”.

The idea is that some “convolutions” will change the image in such a way that certain features in the image emphasized. This is a very basic introduction to what convolutions do, and when combined with something called “**pooling**”, they can become very powerful.

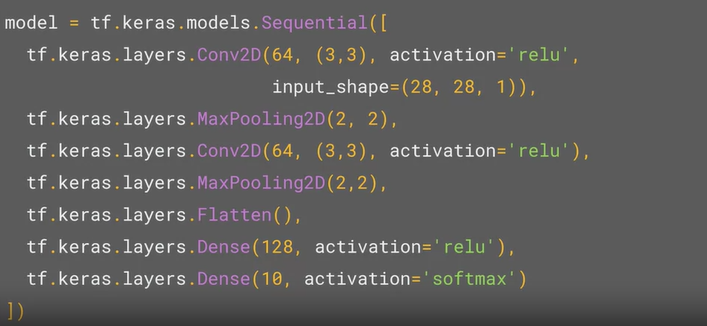
Simply, **pooling** is a way of compressing an image.

## **Implementing Convolutional Layers:**

Here’s our code from earlier example, where we defined a neural network to have an input layer in the shape of our data, and output layer in the shape of the number of categories we wanted to define, and a hidden layer in-between. The “Flatten” takes our square images and turns them into a one-dimensional array.



To add “convolutions” to this, we add these codes:

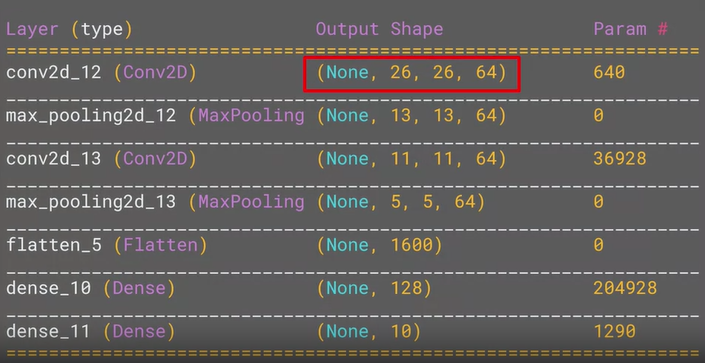


In the first line in Sequential, we are asking “keras” to generate **64 filters** for us. These filters are , their activation is “relu”, which means the negative values will be thrown away, and finally the input shape is as before. That extra 1 just means that we are tallying using a single byte for color depth. (Grey color)

The value “64” for our filter is not random. These filters start with a set of known good filters in a similar way to the pattern fitting you saw earlier, and the ones that work from that set are learned over time.

The second line in Sequential, is used to create a “**pooling layer**”. It is “**max-pooling**” because we are going to take the maximum value. We are saying that it’s a “two-by-two pool”, so for every four pixels, the biggest one will survive. We then add another convolutional layer, and another max-pooling layer so that the network can learn another set of convolutions on top of the existing one, and then again, pool to reduce the size.

*Note: A really useful method on the model is the method. This allows you to inspect the layers and see the journey of the images in the convolutions.*



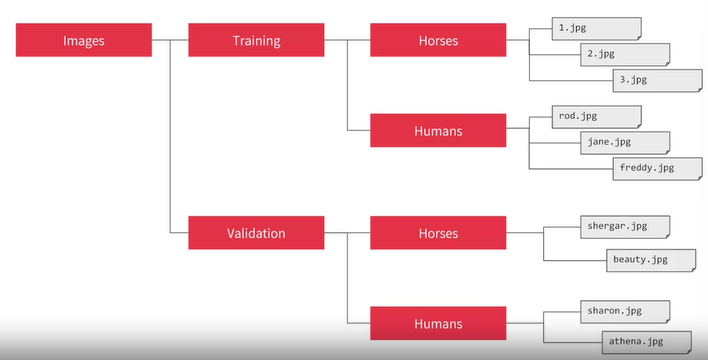
Why is the output here 26 by 26? The key to this is remembering that the filter is “three by three” here. Consider what happens when you start scanning through an image and you are on the top left. There are no neighbors on the upper or left pixel. In the similar sense, the right pixel of the top left pixel, has no neighbors above it. Therefore, the first calculation is this pixel:



## **Understanding ImageDataGenerator:**

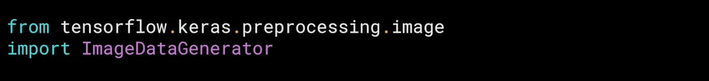
One limitation of our previous examples was that we used very uniform images. What happens when you use larger images and features might be in different locations? For example, how about images have different sizes and different aspect ratios. Or even, in some cases, there might be even multiple subjects. In addition to that, the earlier examples with a fashion data used a built-in dataset. All of the data was handily split into training and test sets for us and labels were available. In many scenarios, that is not going to be the case and we have to do it ourselves.

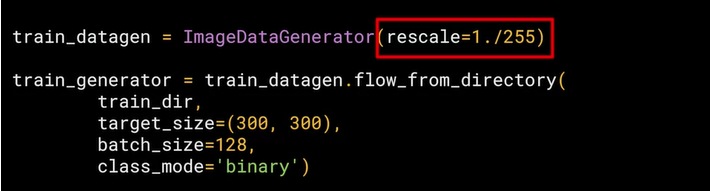
In this lesion, we will take a look at some of the APIs that are available to make that easier for us. In particular, the “image generator in Tensorflow”. One feature of the image generator is that you can point it at a directory and then the sub-directories of that will automatically generate labels for you. For example, consider this directory structure:



Here, you have an image directory and in that, you have sub-directories for training and validation. When you put sub-directories in these for horses and humans and store the requisite images in there, the image generator can create a feeder for those images and auto label them for you.

So, for example, if we point an image generator at the training directory, the labels will be “Horses and Humans” and all of the images in each directory will be loaded and labeled accordingly. Similarly, if we point at the validation directory, the same thing will happen. Let’s see this in code:

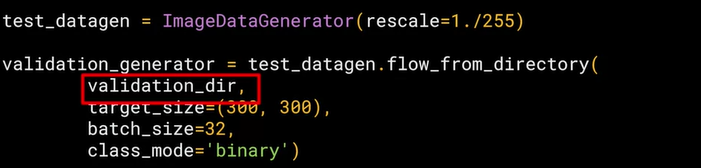




*Note: It is a common mistake that people point the generator at the sub-directory, it will fail in that circumstance. Only point it at the directory that contains sub-directories.*

*Note: The training data will be in batches, which is faster than doing it one by one.*

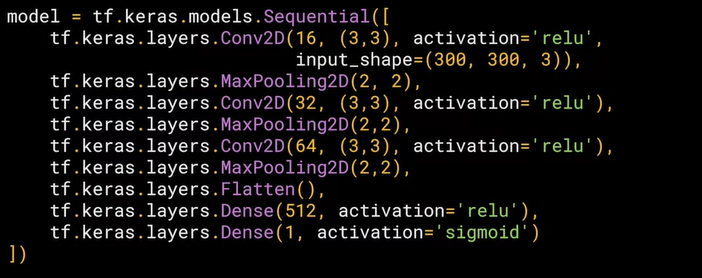
*Note: Since we are just choosing between humans or horses (binary choice), the class mode here is “binary”.*



The validation setup is similar to our training.

## **Defining a ConvNet to use Complex Images:**

Let’s now take a look at the definition of the neural network that we will use to classify “horses versus humans”. It is very similar to our previous examples.

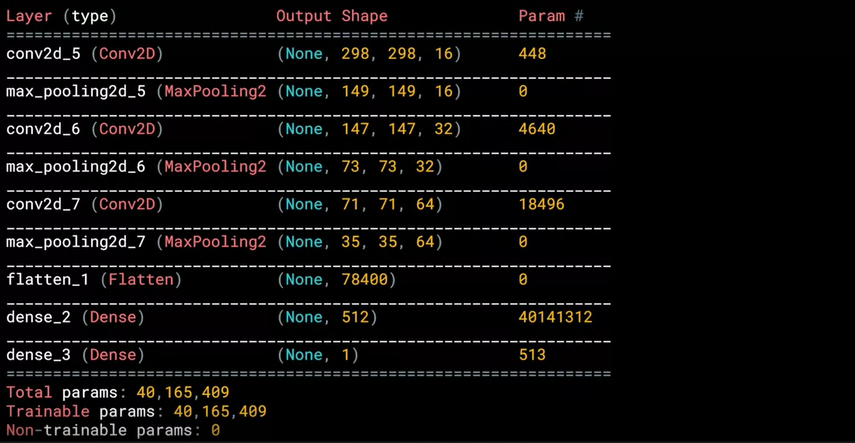




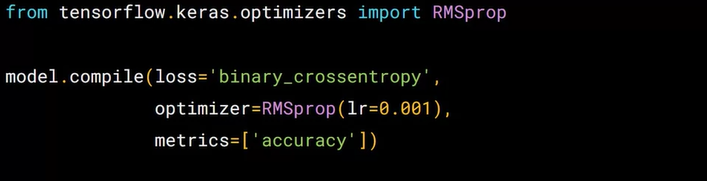
*Note: the number 3 in our input\_shape command, tells that the images have colors (red, blue, green).*

*Note: At our last layer, we used “sigmoid” instead of “softmax” since it works better with binary situations.*

*Note: Our model summary will look something like this:*

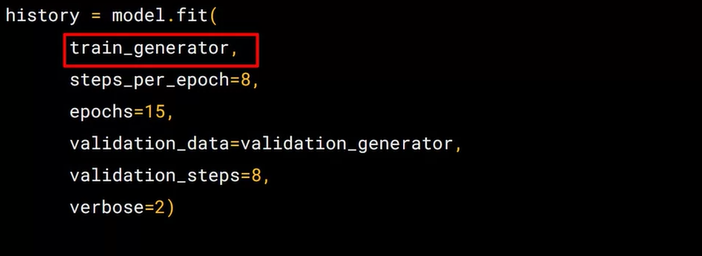
**

Now that you’ve designed the neural network to classify Horses or Humans, the next step is to train it from data that’s on the file system, which can be read by generators.





We could also use the optimizer ‘adam” similar to our previous examples, however, we use “RMSprop” here, where you can adjust the learning rate to experiment with performance.



Let’s take a look at each parameter in our :

* The first parameter is the training generator we set up earlier. This streams the images from the training directory.
* There are 1024 images in the training directory, so we are loading them in 128 at a time. In order to load them all, we need to do 8 batches. So, we use “steps\_per\_epoch” to cover that.
* We set our epochs
* To specify the validation, we use the “validation\_generator” we created earlier.
* Verbose parameter specifies how much to display while training is going on. When set to 2, we will get a little less animation hiding the epoch process.