DeepLearning.AI TensorFlow Developer Professional Certificate

by DeepLearning.AI

Course #1 Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning

Course Site

Made By: Matias Borghi

Table of Contents

Summary

Week 1: A new programming paradigm

Welcome to this course on going from Basics to Mastery of TensorFlow. We're excited you're here! In week 1 you'll get a soft introduction to what Machine Learning and Deep Learning are, and how they offer you a new programming paradigm, giving you a new set of tools to open previously unexplored scenarios. All you need to know is some very basic programming skills, and you'll pick the rest up as you go along. You'll be working with code that works well across both TensorFlow 1.x and the TensorFlow 2.0 alpha. To get started, check out the first video, a conversation between Andrew and Laurence that sets the theme for what you'll study...

Week 2: Introduction to Computer Vision

Welcome to week 2 of the course! In week 1 you learned all about how Machine Learning and Deep Learning is a new programming paradigm. This week you're going to take that to the next level by beginning to solve problems of computer vision with just a few lines of code! Check out this conversation between Laurence and Andrew where they discuss it and introduce you to Computer Vision!

Week 3: Enhancing Vision with Convolutional Neural Networks

Welcome to week 3! In week 2 you saw a basic Neural Network for Computer Vision. It did the job nicely, but it was a little naive in its approach. This week we'll see how to make it better, as discussed by Laurence and Andrew here.

Week 4: Using Real-world Images

Last week you saw how to improve the results from your deep neural network using convolutions. It was a good start, but the data you used was very basic. What happens when your images are larger, or if the features aren't always in the same place? Andrew and Laurence discuss this to prepare you for what you'll learn this week: handling complex images!

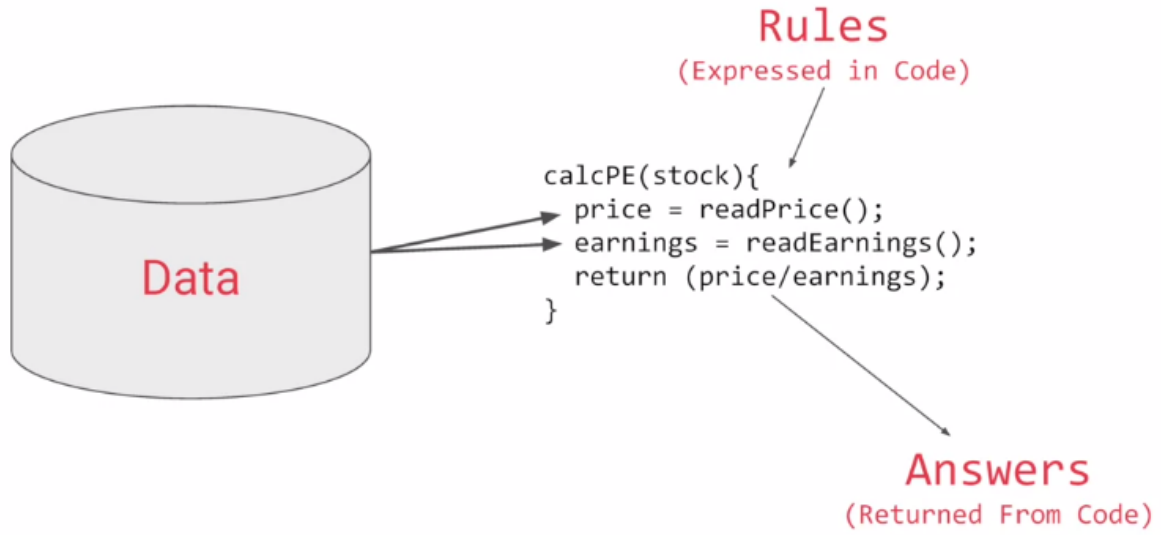
Week 1: A new programming paradigm

A new programming paradigm

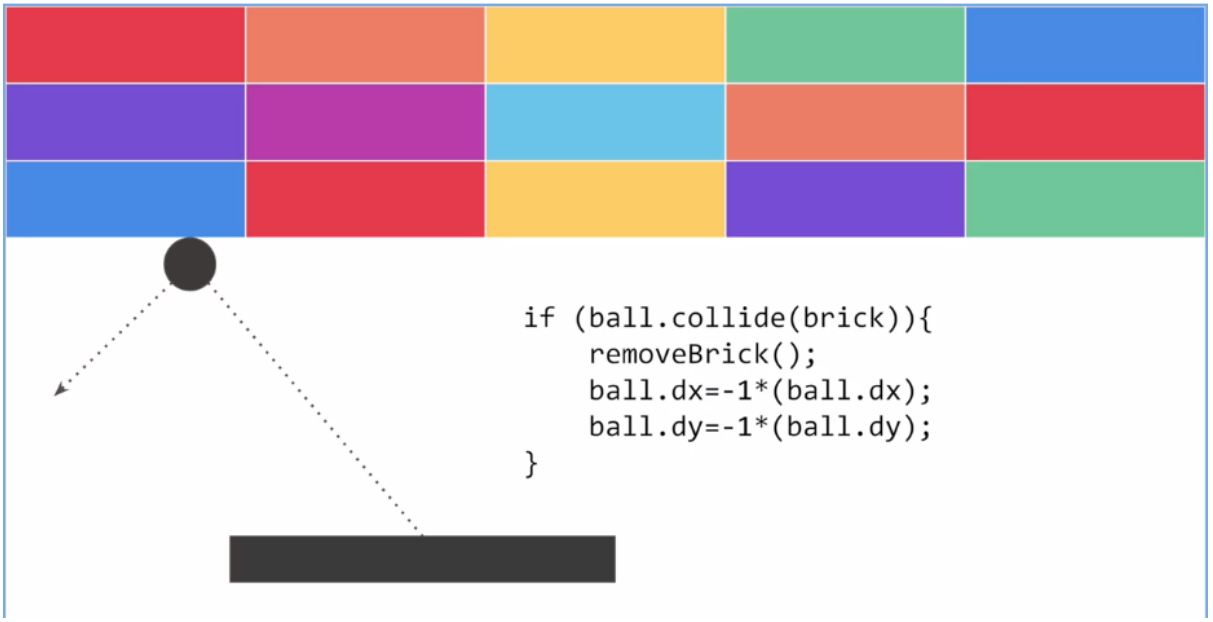
A primer in machine learning

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

So for example, if we have to write an application that figures out a stock analytic, maybe the price divided by the ratio, we can usually write code to get the values from a data source, do the calculation and then return the result.

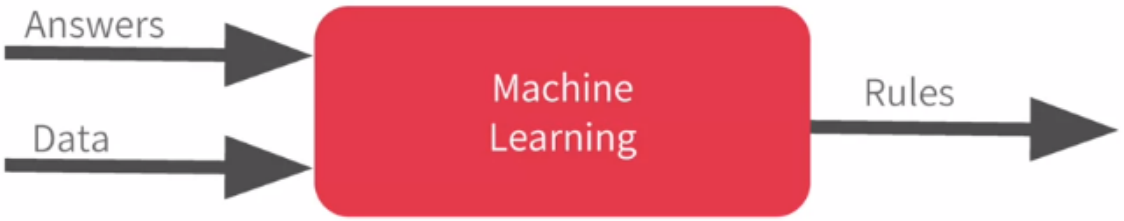


Or if we're writing a game we can usually figure out the rules. For example, if the ball hits the brick then the brick should vanish and the ball should rebound. But if the ball falls off the bottom of the screen then maybe the player loses their life.



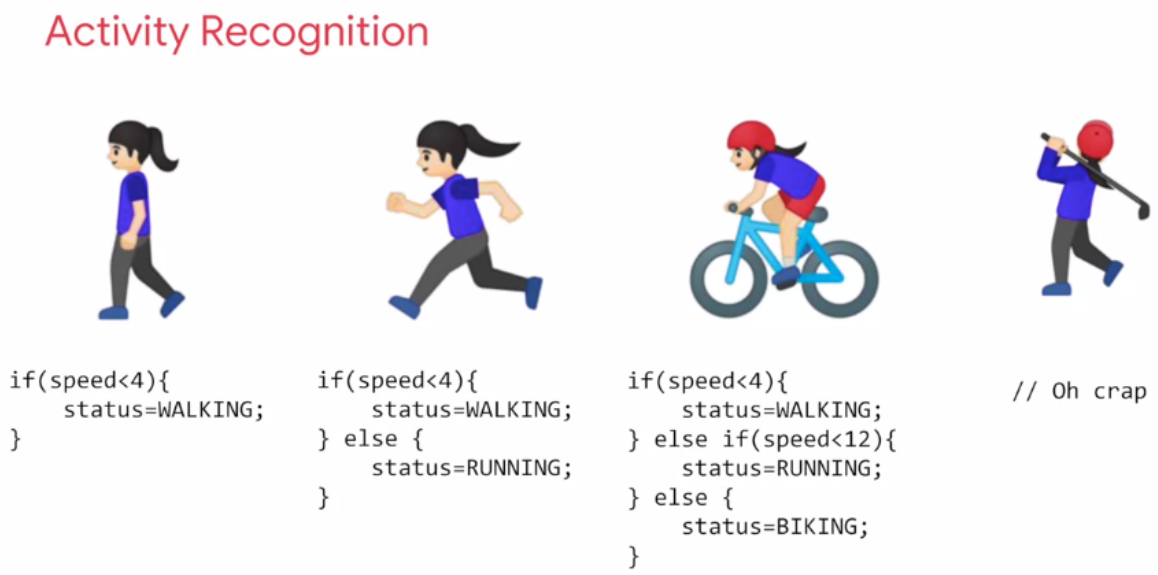
We can represent that with this diagram. Rules and data go in answers come out. Rules are expressed in a programming language and data can come from a variety of sources from local variables all the way up to databases.

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 when should the brick be removed, when should the player's life end, or what's the desired analytic for any other concept, what we will do is we can get a bunch of examples for what we want to see and then have the computer figure out the rules. Now, this is particularly valuable for problems that you can't solve by figuring the rules out for yourself.

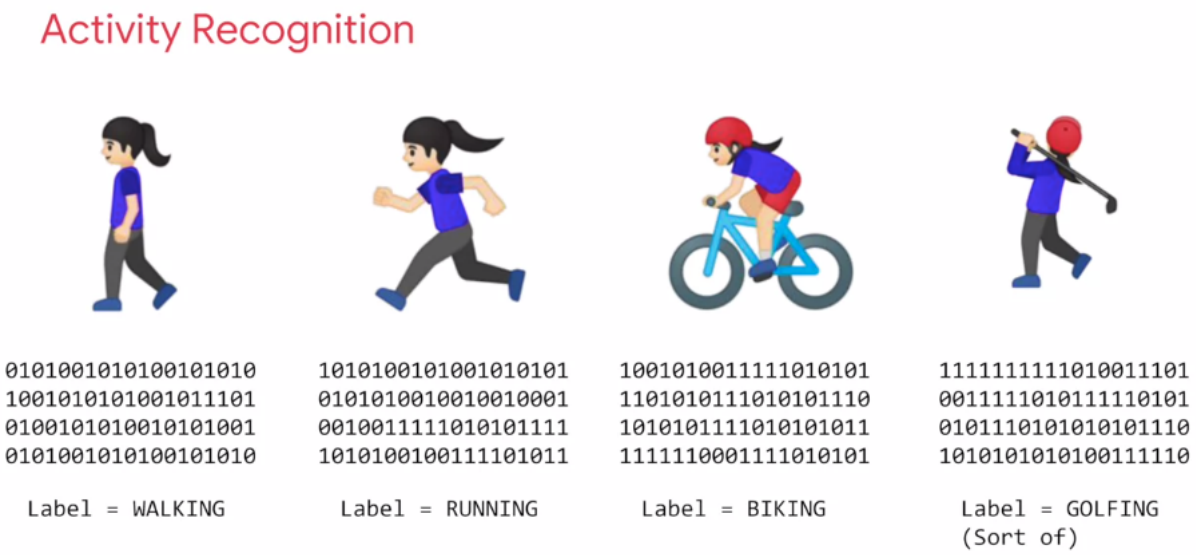


So consider this example, activity recognition. If I'm building a device that detects if somebody is walking and I have data about their speed, I might write code like this and if they're running well that's a faster speed so I could adapt my code to this and if they're biking, well that's not too bad either. I can adapt my code like this. But then I have to do golf recognition too, now my concept becomes broken. But not only that, doing it by speed alone of course is quite naive. We walk and

run at different speeds uphill and downhill and other people walk and run at different speeds to us.



Ultimately machine learning is very similar but we're just flipping the axes. So instead of me trying to express the problem as rules when often that isn't even possible, I'll have to compromise. The new paradigm is that I get lots and lots of examples and then I have labels on those examples and I use the data to say this is what walking looks like, this is what running looks like, this is what biking looks like and yes, even this is what golfing looks like. So, then it becomes answers and data in with rules being inferred by the machine. A machine learning algorithm then figures out the specific patterns in each set of data that determines the distinctiveness of each.



That's what's so powerful and exciting about this programming paradigm. It's more than just a new way of doing the same old thing. It opens up new possibilities that were infeasible to do before. So in the next few minutes, I'm going to show you the basics of creating a neural network which is the workhorse of doing this type of pattern recognition. A neural network is just a slightly

more advanced implementation of machine learning and we call that deep learning. But fortunately it's actually very easy to code. So, we're just going to jump straight into deep learning. We'll start with a simple one and then we'll move on to one that does computer vision in about 10 lines of code. But let's start with a very simple "Hello World" example. So you can see just how everything hangs together.

The 'Hello World' of neural networks

Earlier we mentioned that machine learning is all about a computer learning the patterns that distinguish things. Like for activity recognition, it was the pattern of walking, running and biking that can be learned from various sensors on a device. To show how that works, let's take a look at a set of numbers and see if you can determine the pattern between them. Okay, here are the numbers. There's a formula that maps X to Y. Can you spot it? Take a moment.

```
X = -1, 0, 1, 2, 3, 4
Y = -3, -1, 1, 3, 5, 7
```

Well, the answer is Y equals 2X minus 1. So whenever you see a Y, it's twice the corresponding X minus 1. If you figured it out for yourself, well done, but how did you do that? How would you think you could figure this out? Maybe you can see that the Y increases by 2 every time the X increases by 1. So it probably looks like Y equals 2X plus or minus something. Then when you saw X equals 0 and Y equals minus 1, you thought hey that the something is a minus 1, so the answer might be Y equals 2X minus 1. You probably tried that out with a couple of other values and see that it fits.

Congratulations, you've just done the basics of machine learning in your head. So let's take a look at it in code now.

```
model = keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

Okay, here's our first line of code. 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 are a lot of new concepts here. They will become clear quite quickly as you work through them. The simplest possible neural network is one that has only one neuron in it, and that's what this line of code does. 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. But as I've said, there's

only one. So you have a single neuron. You define the shape of what's input to the neural network in the first and in this case the only layer, and you can see that our input shape is super simple. It's just one value. You've probably seen that for machine learning, you need to know and use a lot of math, calculus, probability and the like. It's really good to understand that as you want to optimize your models but the nice thing for now about TensorFlow and Keras is that a lot of that math is implemented for you in functions. 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 10*X-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. Then the logic is that each guess should be better than the one before. As the guesses get better and better, as 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*. If you want to learn more about these particular functions, as well as the other options that might be better in other scenarios, check out the TensorFlow documentation. But for now we're just going to use this.

Our next step is to represent the known data. These are the Xs and the Ys that you saw earlier. The *np.array* is using a Python library called numpy that makes data representation particularly enlists much easier. So here you can see we have one list for the Xs and another one for the Ys. 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=500 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 are 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. So it hasn't previously seen 10, and what do you think it will return when you pass it a 10? Now you might think it would return 19 because after all Y equals 2X minus 1, and you think it should be 19. But when you try this in the workbook yourself, you'll see that it will return a value very close to 19 but not exactly 19. Now why do you think that would be? Ultimately there are two main reasons. The first is that you trained it using very little data. There's only six points. Those six points are linear but there's no guarantee that for every X, the relationship will be Y=2X-1. There's a very high probability that Y equals 19 for X equals 10, but the neural network isn't positive. So it will figure out a realistic value for Y. That's the second main reason. When using neural networks, as they try to figure out the answers for everything, they deal in probability. You'll see that a lot and you'll have to adjust how you handle answers to fit. Keep that in mind as you work through the code. Okay, enough theory. Now let's get hands-on and write the code that we just saw and then we can run it.

Working through ‘Hello World’ in TensorFlow and Python

Intro To Google Colab

Week 1 Resources

That brings you to the end of what you need to look at for Week 1. If you're eager to learn more, before we go to Week 2, there are some great resources you can check out:

- AI For Everyone is a non-technical course that will help you understand many of the AI technologies we will discuss later in this course, and help you spot opportunities in applying this technology to solve your problems. https://www.deeplearning.ai/ai-for-everyone/
- TensorFlow is available at TensorFlow.org, and video updates from the TensorFlow team are at youtube.com/tensorflow

Play with a neural network right in the browser at http://playground.tensorflow.org. See if you can figure out the parameters to get the neural network to pattern match to the desired groups. The spiral is particularly challenging!

The 'Hello World' notebook that we used in this course is available on GitHub here.

Week 2: Introduction to Computer Vision

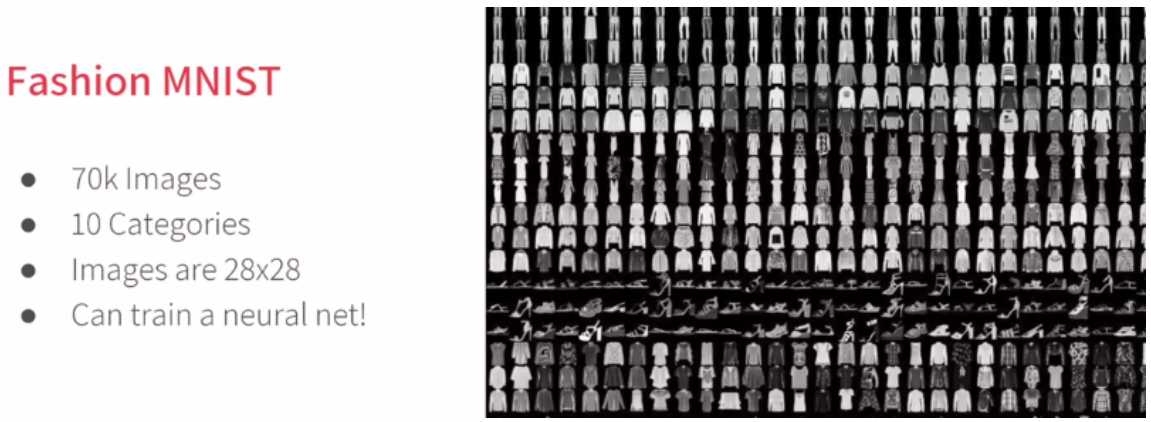
Introduction to Computer Vision

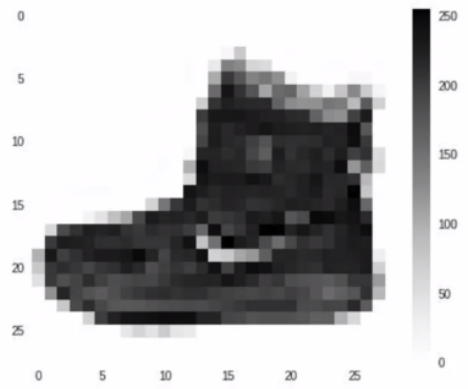
An Introduction to computer vision

In the previous lesson, you learned what the machine learning paradigm is and how you use data and labels and have a computer infer the rules between them for you. You looked at a very simple example where it figured out the relationship between two sets of numbers. Let's now take this to the next level by solving a real problem, computer vision. Computer vision is the field of having a computer understand and label what is present in an image. Consider this slide.

When you look at it, you can interpret what a shirt is or what a shoe is, but how would you program for that? If an extra terrestrial who had never seen clothing walked into the room with you, how would you explain the shoes to him? It's really difficult, if not impossible to do right? And it's the same problem with computer vision. So one way to solve that is to use lots of pictures of clothing and tell the computer what that's a picture of and then have the computer figure out the patterns that give you the difference between a shoe, and a shirt, and a handbag, and a coat. That's what you're going to learn how to do in this section.

Fortunately, there's a data set called Fashion MNIST which gives 70 thousand images spread across 10 different items of clothing.

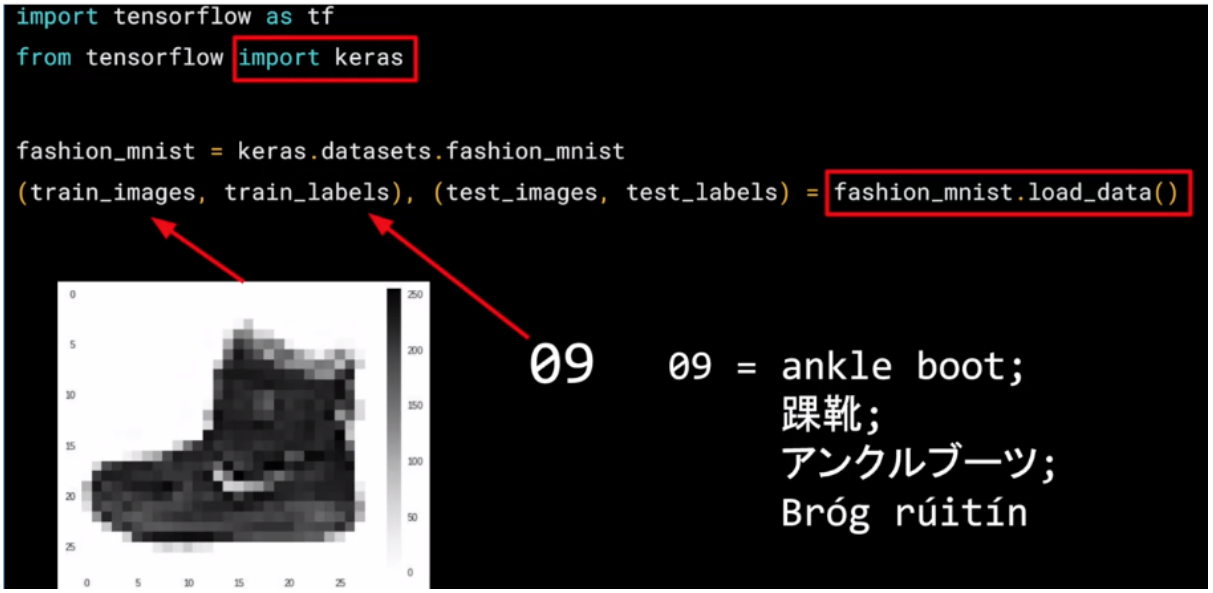




These images have been scaled down to 28 by 28 pixels. Now usually, the smaller the better because the computer has less processing to do. But of course, you need to retain enough information to be sure that the features and the object can still be distinguished. If you look at this slide you can still tell the difference between shirts, shoes, and handbags. So this size does seem to be ideal, and it makes it great for training a neural network. The images are also in grayscale, so the amount of information is also reduced. Each pixel can be represented in values from zero to 255 and so it's only one byte per pixel. With 28 by 28 pixels in an image, only 784 bytes are needed to store the entire image. Despite that, we can still see what's in the image and in this case, it's an ankle boot, right?

Writing code to load training data

So what will handling this look like in code? In the previous lesson, you learned about TensorFlow and Keras, and how to define a super simple neural network with them. In this lesson, you're going to use them to go a little deeper but the overall API should look familiar. The one big difference will be in the data.



The last time you had your six pairs of numbers, so you could hard code it. This time you have to load 70,000 images off the disk, so there'll be a bit of code to handle that. Fortunately, it's still quite simple because Fashion-MNIST is available as a data set with an API call in TensorFlow. We simply declare an object of type MNIST loading it from the Keras database. On this object, if we call the load data method, it will return four lists to us. That's the training data, the training labels, the testing data, and the testing labels. Now, what are these you might ask? Well, when building a neural network like this, it's a nice strategy to use some of your data to train the neural network and similar data that the model hasn't yet seen to test how good it is at recognizing the images. So in the Fashion-MNIST data set, 60,000 of the 70,000 images are used to train the

network, and then 10,000 images, one that it hasn't previously seen, can be used to test just how good or how bad it is performing. So this code will give you those sets. Then, each set has data, the images themselves and labels and that's what the image is actually of. So for example, the training data will contain images like this one, and a label that describes the image like this. **While this image is an ankle boot, the label describing it is the number nine. Now, why do you think that might be? There's two main reasons. First, of course, is that computers do better with numbers than they do with texts. Second, importantly, is that this is something that can help us reduce bias.** If we labeled it as an ankle boot, we would of course be biased towards English speakers. But with it being a numeric label, we can then refer to it in our appropriate language be it English, Chinese, Japanese, or here, even Irish Gaelic.

> 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.

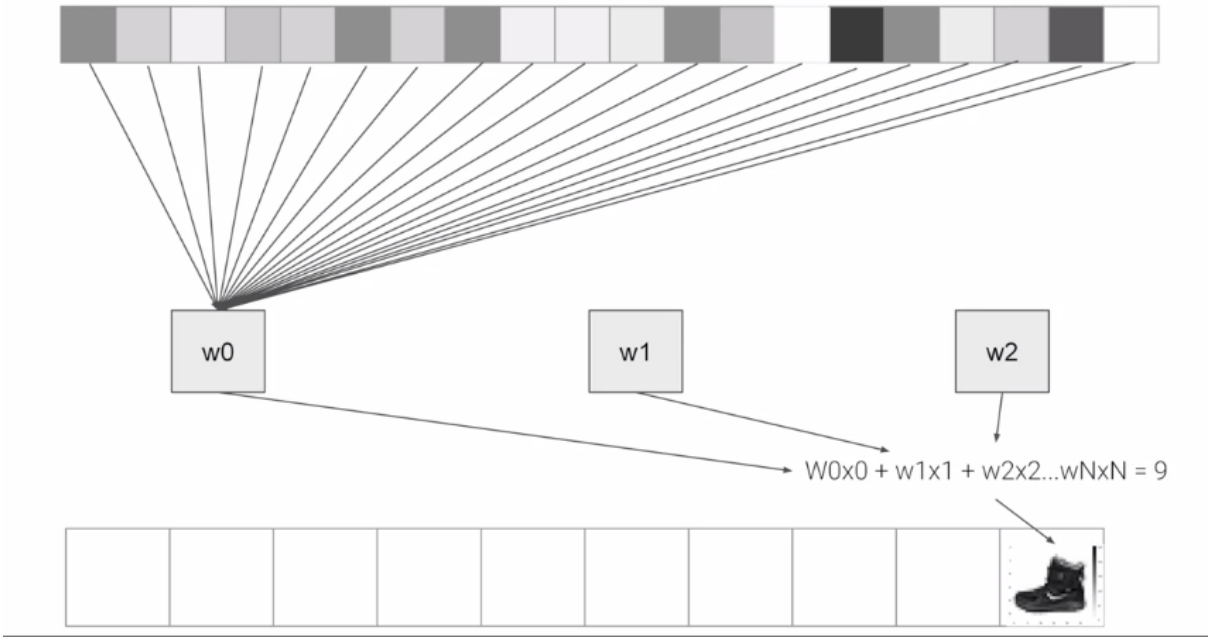
Coding a Computer Vision Neural Network

Okay. So now we will look at the code for the neural network definition. Remember last time we had a sequential with just one layer in it. Now we have three layers. The important things to look at are the first and the last layers.

```
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

The last layer has 10 neurons in it because we have ten classes of clothing in the dataset. They should always match. The first layer is a flatten layer with the input shaping 28 by 28. Now, if you remember our images are 28 by 28, so we're specifying that this is the shape that we should expect the data to be in. Flatten takes this 28 by 28 square and turns it into a simple linear array. The interesting stuff happens in the middle layer, sometimes also called a hidden layer. There are 128 neurons in it, and I'd like you to think about these as variables in a function. Maybe call them x1, x2 x3, etc. Now, there exists a rule that incorporates all of these that turns the 784 values of an ankle boot into the value nine, and similar for all of the other 70,000. It's too complex a function for you to see by mapping the images yourself, but that's what a neural net does. So, for example, if you then say the function was y equals w1 times x1, plus w2 times x2, plus w3 times x3, all the way up to a w128 times x128. By figuring out the values of w, then y will be nine, when you have the input value of the shoe. You'll see that it's doing something very, very similar to what we did earlier when we figured out y equals 2x minus one. In that case the two were the weight of x. So, I'm saying y equals w1 times x1, etc. Now, don't worry if this isn't very clear right now. Over time, you will get the hang of it, seeing that it works, and there's also some tools that will allow you to peek inside to see what's going on. The important thing for now is to get the code working, so you can see a classification scenario for yourself. You can also tune the neural

network by adding, removing and changing layer size to see the impact. You'll do that in the next exercise.



Using Callbacks to control training

A question I often get at this point from programmers in particular when experimenting with different numbers of epochs is, How can I stop training when I reach a point that I want to be at? What do I always have to hard code it to go for certain number of epochs? Well, 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're what you want to say, then you can cancel the training at that point. Let's take a look. Okay, so here's our code for training the neural network to recognize the fashion images. In particular, keep an eye on the model.fit function that executes the training loop. You can see that here. What we'll now do is write a callback in Python. Here's the code.

```
class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('loss')<0.4):
      print("\nLoss is low so cancelling training!")
      self.model.stop_training = True
```

It's implemented as a separate class, but that can be in-line with your other code. It doesn't need to be in a separate file. In it, we'll implement the on_epoch_end function, which gets called by the callback whenever the epoch ends. It also sends a logs object which contains lots of great

information about the current state of training. For example, the current loss is available in the logs, so we can query it for certain amount. For example, here I'm checking if the loss is less than 0.4 and canceling the training itself. Now that we have our callback, let's return to the rest of the code, and there are two modifications that we need to make. First, we instantiate the class that we just created, we do that with this code. Then, in my model.fit, I used the callbacks parameter and pass it this instance of the class.

```
callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5, callbacks=[callbacks])
```

See how to implement Callbacks

Experiment with using Callbacks in this notebook -- work through it to see how they perform!

Week 2 Resources

Here are all the notebook files for this week, hosted on GitHub. You can download and play with them from there!

Beyond Hello, World - A Computer Vision Example

Exploring Callbacks

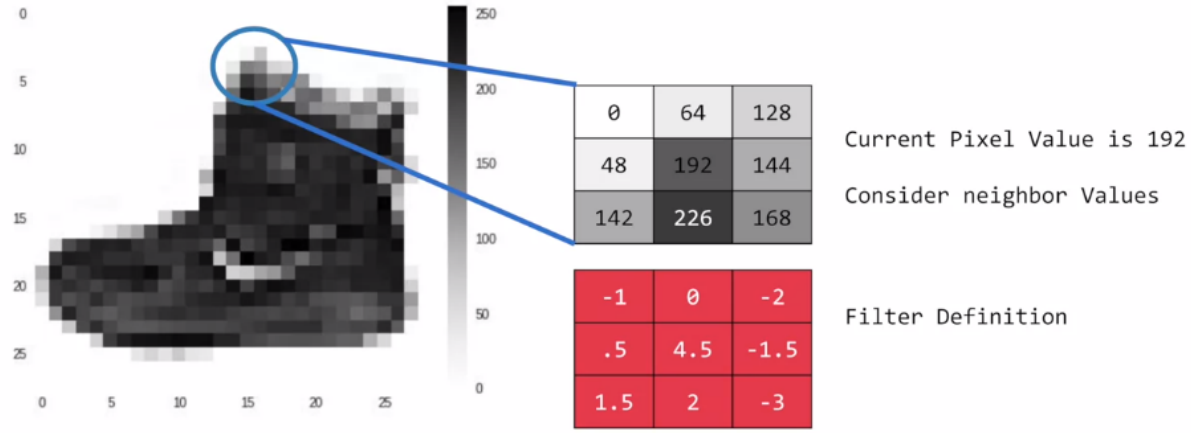
Exercise 2 - Handwriting Recognition - Answer

Week 3: Enhancing Vision with Convolutional Neural Networks

Enhancing Vision with Convolutional Neural Networks

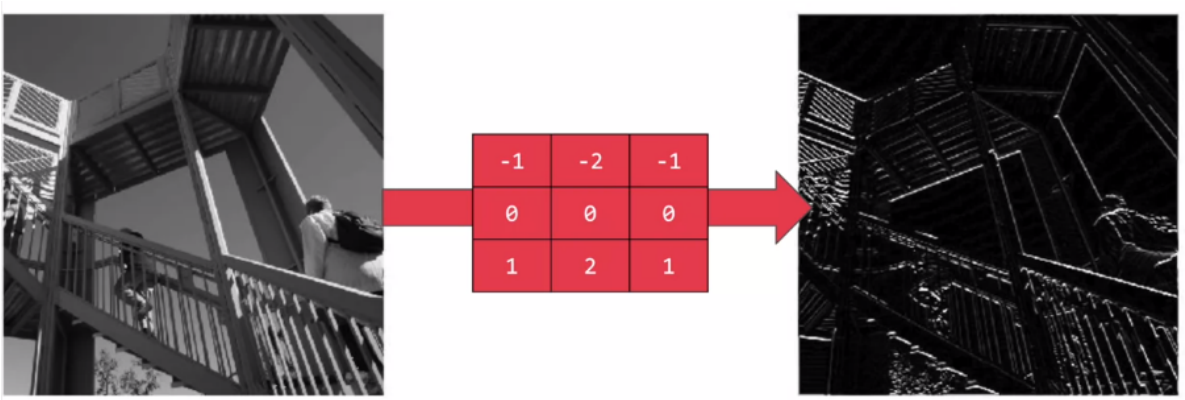
What are convolutions and pooling?

In the previous example, you saw how you could create a neural network called a deep neural network to pattern match a set of images of fashion items to labels. In just a couple of minutes, you're able to train it to classify with pretty high accuracy on the training set, but a little less on the test set. Now, one of the things that you would have seen when you looked at the images is that there's a lot of wasted space in each image. While there are only 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 a handbag, or a shirt. That's where convolutions come in. So, what's convolution? You might ask. Well, if you've ever done any kind of image processing, it usually involves having a filter and passing that filter over the image in order to change the underlying image. The process works a little bit like this. For every pixel, take its value, and take a look at the value of its neighbors. If our filter is three by three, 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 value in the filter. So, for example, in this case, our pixel has the value 192, and its upper left neighbor has the value zero. The upper left value and the filter is negative one, so we multiply zero by negative one. Then we would do the same for the upper neighbor. Its value is 64 and the corresponding filter value was zero, so we'd multiply those out. Repeat this 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.

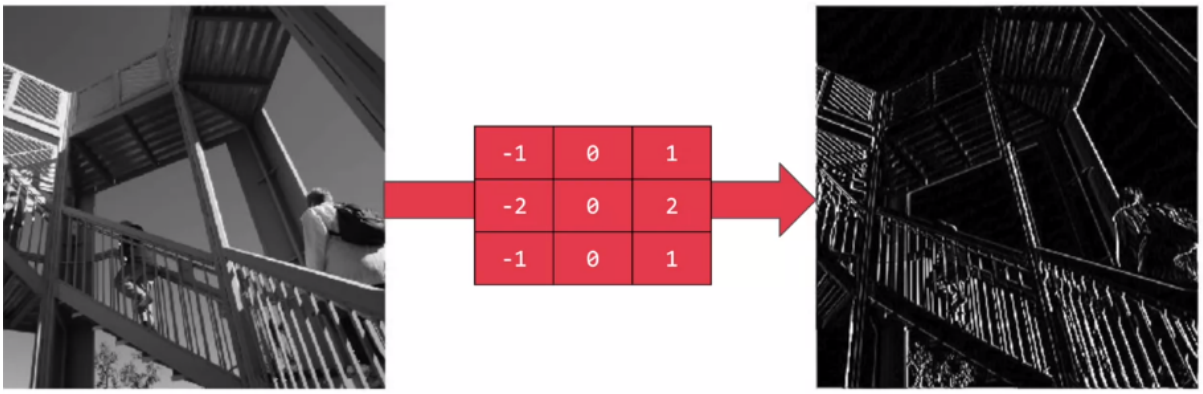



```
CURRENT_PIXEL_VALUE = 192
NEW_PIXEL_VALUE = (-1 * 0) + (0 * 64) + (-2 * 128) +
                  (.5 * 48) + (4.5 * 192) + (-1.5 * 144) +
                  (1.5 * 42) + (2 * 226) + (-3 * 168)
```

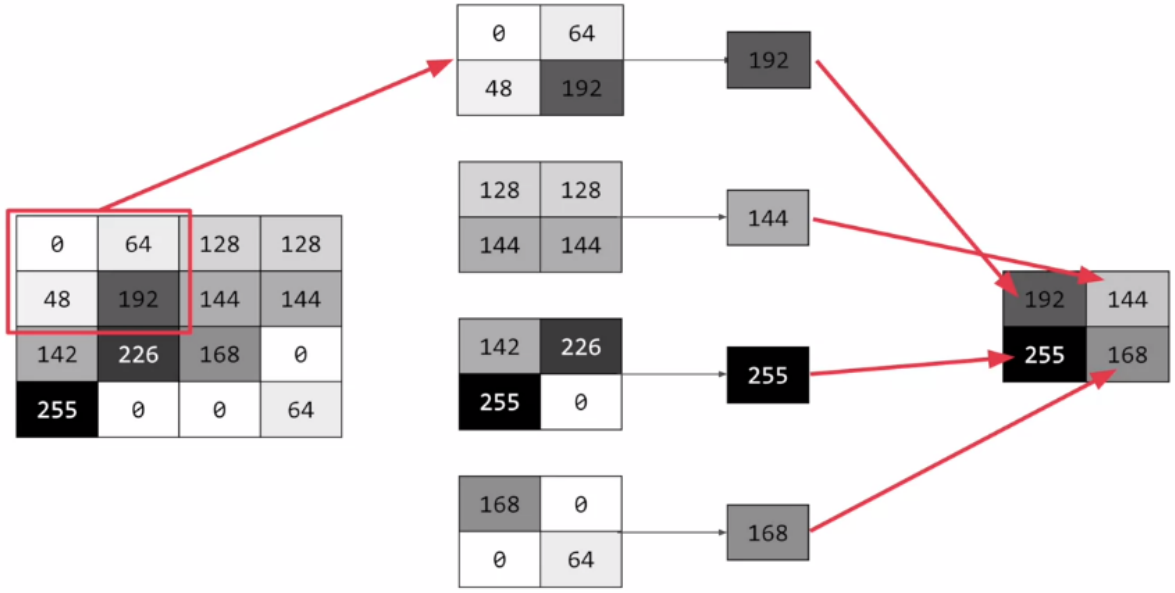
It's really as simple as that. The idea here is that some convolutions will change the image in such a way that certain features in the image get emphasized. So, for example, if you look at this filter, then the **vertical lines in the image really pop out**.



With this filter, **the horizontal lines pop out**.



Now, that's a very basic introduction to what convolutions do, and when combined with something called **pooling**, they can become really powerful. But simply, pooling is a way of compressing an image. A quick and easy way to do this, is to go over the image of four pixels at a time, i.e, the current pixel and its neighbors underneath and to the right of it. Of these four, pick the biggest value and keep just that. So, for example, you can see it here. My 16 pixels on the left are turned into the four pixels on the right, by looking at them in two-by-two grids and picking the biggest value. This will preserve the features that were highlighted by the convolution, while simultaneously quartering the size of the image. We have the horizontal and vertical axes.



Coding convolutions and pooling layers

The concepts introduced in this video are available as Conv2D layers and MaxPooling2D layers in TensorFlow. You'll learn how to implement them in code in the next video…

Implementing convolutional layers

So now let's take a look at convolutions and pooling in code. We don't have to do all the math for filtering and compressing, we simply define convolutional and pooling layers to do the job for us.

So here's our code from the earlier example, where we defined out 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're trying to define, and a hidden layer in the middle. The Flatten takes our square 28 by 28 images and turns them into a one dimensional array.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

To add convolutions to this, you use code like this. You'll see that the last three lines are the same, the Flatten, the Dense hidden layer with 128 neurons, and the Dense output layer with 10 neurons. What's different is what has been added on top of this. Let's take a look at this, line by line.

Here we're specifying the first convolution. 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.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu',
                         input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Now, of course, you might wonder what the 64 filters are. It's a little beyond the scope of this class to define them, but they aren't random. They start with a set of known good filters in a similar way to the pattern fitting that you saw earlier, and the ones that work from that set are learned over time.

For more details on convolutions and how they work, there's a great set of resources here.

Learn more about convolutions

You've seen how to add a convolutional 2d layer to the top of your neural network in the previous video. If you want to see more detail on how they worked, check out the playlist at https://bit.ly/2UGa7uH.Now let's take a look at adding the pooling, and finishing off the convolutions so you can try them out.

Implementing pooling layers

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.

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. So, by the time the image gets to the flatten to go into the dense layers, it's already much smaller. It's being quartered, and then quartered again. So, its content has been greatly

simplified, the goal being that the convolutions will filter it to the features that determine the output.

A really useful method on the model is the *model.summary* method. This allows you to inspect the layers of the model, and see the journey of the image through the convolutions, and here is the output. It's a nice table showing us the layers, and some details about them including the output shape.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               204928
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1290
=================================================================
```

It's important to keep an eye on the output shape column. When you first look at this, it can be a little bit confusing and feel like a bug. After all, **isn't the data 28 by 28, so y is the output, 26 by 26**. The key to this is remembering that the filter is a three by three filter. Consider what happens when you start scanning through an image starting on the top left. So, for example with this image of the dog on the right, you can see zoomed into the pixels at its top left corner. You can't calculate the filter for the pixel in the top left, because it doesn't have any neighbors above it or to its left. In a similar fashion, the next pixel to the right won't work either because it doesn't have any neighbors above it. So, logically, the first pixel that you can do calculations on is this one, because this one of course has all eight neighbors that a three by three filter needs. This when you think about it, means that you can't use a one pixel margin all around the image, so the output of the convolution will be two pixels smaller on x, and two pixels smaller on y. If your filter is five-by-five for similar reasons, your output will be four smaller on x, and four smaller on y. So, that's y with a three by three filter, our output from the 28 by 28 image is now 26 by 26, we've removed that one pixel on x and y, and each of the borders.

So, next is the first of the max-pooling layers. Now, remember we specified it to be two-by-two, thus turning four pixels into one, and having our x and y. So, now our output gets reduced from 26 by 26, to 13 by 13.

The convolutions will then operate on that, and of course, we lose the one pixel margin as before, so we're down to 11 by 11, add another two-by-two max-pooling to have this rounding down, and go down, down to five-by-five images.

So, now our dense neural network is the same as before, but it's being fed with five-by-five images instead of 28 by 28 ones. But remember, **it's not just one compressed five-by-five image instead of the original 28 by 28, there are a number of convolutions per image that we specified, in this case 64**. So, there are 64 new images of five-by-five that have been fed in. Flatten that out and you have 25 pixels times 64, which is 1600. So, you can see that the new flattened layer has 1,600 elements in it, as opposed to the 784 that you had previously. This number is impacted by the parameters that you set when defining the convolutional 2D layers. Later when you experiment, you'll see what the impact of setting what other values for the number of convolutions will be, and in particular, you can see what happens when you're feeding less than 784 over all pixels in. Training should be faster, but is there a sweet spot where it's more accurate? Well, let's switch to the workbook, and we can try it out for ourselves.

Experiment with filters and pools

To try this notebook for yourself, and play with some convolutions, here's the notebook. Let us know if you come up with any interesting filters of your own!

As before, spend a little time playing with this notebook. Try different filters, and research different filter types. There's some fun information about them here: https://lodev.org/cgtutor/filtering.html

Week 3 Resources

We've put the notebooks that you used this week into GitHub so you can download and play with them.

Adding Convolutions to Fashion MNIST

Exploring how Convolutions and Pooling work

Week 4: Using Real-world Images

Using Real-world Images

Explore an impactful, real-world solution

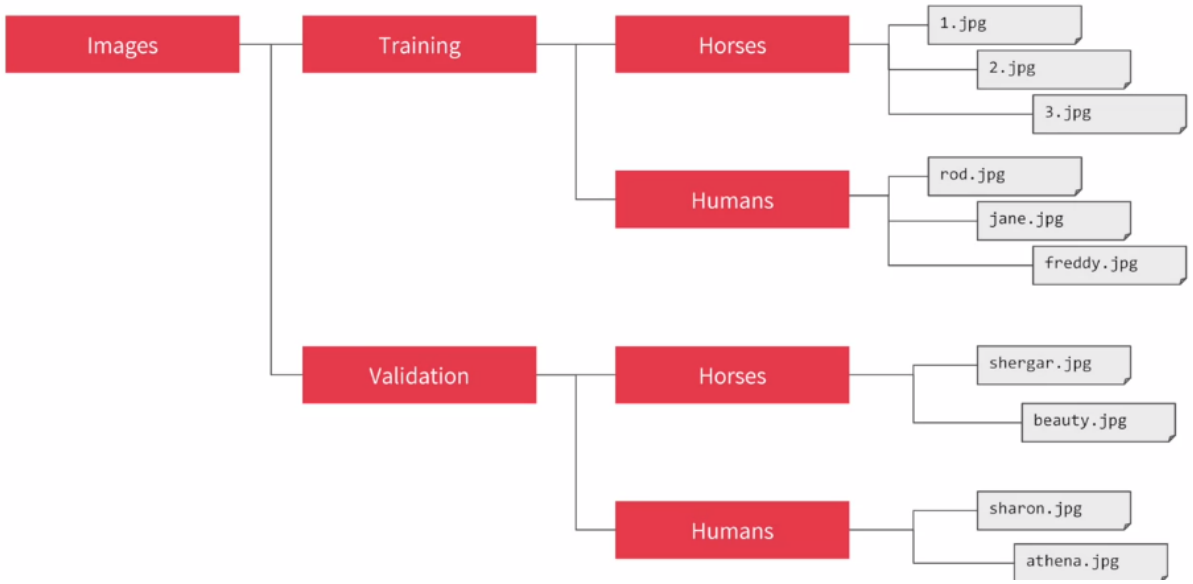
As Andrew and Laurence discussed, the techniques you've learned already can apply to complex images, and you can start solving real scenarios with them. They discussed how it could be used, for example, in disease detection with the Cassava plant, and you can see a video demonstrating that here. Once you've watched that, move onto the next lesson!

Understanding ImageGenerator

To this point, you built an image classifier that worked using a deep neural network and you saw how to improve its performance by adding convolutions. One limitation though was that it used a dataset of very uniform images. Images of clothing that was staged and framed in 28 by 28. **But what happens when you use larger images and where the feature might be in different locations?** For example, how about these images of horses and humans?

They have different sizes and different aspect ratios. The subject can be in different locations. In some cases, there may even be 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 you and labels were available. In many scenarios, that's not going to be the case and you'll have to do it for yourself. So in this lesson, we'll take a look at some of the APIs that are available to make that easier for you. In particular, the **ImageGenerator** 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. So for example, consider this directory structure.



You have an images 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 I 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 I point one at the validation directory, the same thing will happen. So let's take a look at this in code.

```
train_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(300, 300),
        batch_size=128,
        class_mode='binary')
```

The image generator class is available in Keras.preprocessing.image. You can then instantiate an image generator like this. I'm going to pass '*rescale*' to it to normalize the data. You can then call the *flow_from_directory* method on it to get it to load images from that directory and its sub-directories. It's a common mistake that people point the generator at the sub-directory. It will fail in that circumstance. You should always point it at the directory that contains sub-directories that contain your images. The names of the sub-directories will be the labels for your images that are contained within them. So make sure that the directory you're pointing to is the correct one. You put it in the second parameter like this. Now, images might come in all shapes and sizes and unfortunately for training a neural network, the input data all has to be the same size, so the images will need to be resized to make them consistent. The nice thing about this code is that the images are resized for you as they're loaded. So you don't need to preprocess thousands of

images on your file system. But you could have done that if you wanted to. ***The advantage of doing it at runtime like this is that you can then experiment with different sizes without impacting your source data.*** While the horses and humans dataset is already in 300 by 300, when you use other datasets they may not always be uniformly sized. So this is really useful for you. The images will be loaded for training and validation in batches where it's more efficient than doing it one by one. Now, there's a whole science to calculating batch size that's beyond the scope of this course, but you can experiment with different sizes to see the impact on the performance by changing this parameter. Finally, there's the class mode. Now, this is a binary classifier i.e. it picks between two different things; horses and humans, so we specify that here. Other options in particular for more than two things will be explored later in the course. The validation generator should be exactly the same except of course it points at a different directory, the one containing the sub-directories containing the test images.

```
test_datagen = ImageDataGenerator(rescale=1./255)

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(300, 300),
        batch_size=32,
        class_mode='binary')
```

When you go through the workbook shortly, you'll see how to download the images as a zip, and then sort them into training and test sub-directories, and then put horses and humans sub-directories in each. That's just pure Python. It's not TensorFlow or any other deep learning stuff. But it's all explained for you in the notebook.

Defining a ConvNet to use complex images

So let's now take a look at the definition of the neural network that we'll use to classify horses versus humans. It's very similar to what you just used for the fashion items, but there are a few minor differences based on this data, and the fact that we're using generators. So here's the code.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

As you can see, it's sequential as before with convolutions and pooling before we get to the dense layers at the bottom. But let's highlight some of the differences. First of all, you'll notice that there are three sets of convolution pooling layers at the top. This reflects the higher complexity and size of the images. Remember our earlier our 28 by 28.5 to 13 and then five before flattening, well, now we have 300 by 300. So we start at 298 by 298 and then have that etc., etc. until by the end, we're at a 35 by 35 image. We can even add another couple of layers to this if we wanted to get to the same ballpark size as previously, but we'll keep it at three for now. Another thing to pay attention to is the input shape. We resize their images to be 300 by 300 as they were loaded, but they're also color images. So there are three bytes per pixel. One byte for the red, one for green, and one for the blue channel, and that's a common 24-bit color pattern. If you're paying really close attention, you can see that the output layer has also changed. Remember before when you created the output layer, you had one neuron per class, but now there's only one neuron for two classes. That's because we're using a different activation function where sigmoid is great for binary classification, where one class will tend towards zero and the other class tending towards one. You could use two neurons here if you want, and the same softmax function as before, but for binary this is a bit more efficient. If you want you can experiment with the workbook and give it a try yourself.

Now, if we take a look at our model summary, we can see the journey of the image data through the convolutions The 300 by 300 becomes 298 by 298 after the three by three filter, it gets pulled to 149 by 149 which in turn gets reduced to 73 by 73 after the filter that then gets pulled to 35 by 35, this will then get flattened, so 64 convolutions that are 35 squared and shape will get fed into the DNN. If you multiply 35 by 35 by 64, you get 78,400, and that's the shape of the data once it comes out of the convolutions. If we had just fed raw 300 by 300 images without the convolutions, that would be over 900,000 values. So we've already reduced it quite a bit.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 298, 298, 16)      448
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 149, 149, 16)      0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 147, 147, 32)      4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 73, 73, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 71, 71, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 35, 35, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 78400)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               40141312
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 40,165,409
Trainable params: 40,165,409
Non-trainable params: 0
```

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. To do this, you don't use model.fit as earlier, but a new method call: ***model.fit_generator***. In the next video you'll see the details of this.

Training the ConvNet with fit_generator

Okay, we'll now compile the model and, as always, we have a loss function and an optimizer. When classifying the ten items of fashion, you might remember that your loss function was a categorical cross entropy. But because we're doing a binary choice here, let's pick a ***binary_crossentropy*** instead. Also, earlier we used an Adam optimizer. Now, you could do that again, but I thought it would be fun to use the **RMSprop**, where you can adjust the learning rate to experiment with performance.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['acc'])
```

Okay, next up is the training, now, this looks a little different than before when you called model.fit. Because now you call ***model.fit_generator***, and that's because we're using a generator instead of datasets. Remember the image generator from earlier, let's look at each parameter in detail.

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=8,
      verbose=2)
```

The first parameter is the ***training generator*** that you set up earlier. This streams the images from the training directory. Remember the batch size you used when you created it, it was 20, that's important in the next step. There are 1,024 images in the training directory, so we're loading them in 128 at a time. So in order to load them all, we need to do 8 batches. So we set the ***steps_per_epoch*** to cover that.

Here we just set the number of ***epochs*** to train for. This is a bit more complex, so let's use, say, 15 epochs in this case. And now we specify the validation set that comes from the

validation_generator that we also created earlier. It had 256 images, and we wanted to handle them in batches of 32, so we will do 8 steps.And the ***verbose*** parameter specifies how much to display while training is going on. With verbose set to 2, we'll get a little less animation hiding the epoch progress.

Once the model is trained, you will, of course, want to do some prediction on the model. And here's the code to do that, let's look at it piece by piece.

```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

  # predicting images
  path = '/content/' + fn
  img = image.load_img(path, target_size=(300, 300))
  x = image.img_to_array(img)
  x = np.expand_dims(x, axis=0)

  images = np.vstack([x])
  classes = model.predict(images, batch_size=10)
  print(classes[0])
  if classes[0]>0.5:
    print(fn + " is a human")
  else:
    print(fn + " is a horse")
```

So these parts are specific to Colab, they are what gives you the button that you can press to pick one or more images to upload. The image paths then get loaded into this list called uploaded. The loop then iterates through all of the images in that collection. And you can load an image and prepare it to input into the model with this code. Take note to ensure that the dimensions match the input dimensions that you specified when designing the model. You can then call model.predict, passing it the details, and it will return an array of classes. In the case of binary classification, this will only contain one item with a value close to 0 for one class and close to 1 for the other.Later in this course you'll see multi-class classification with Softmax. Where you'll get a list of values with one value for the probability of each class and all of the probabilities adding up to 1.

Training the neural network

Now that you've learned how to download and process the horses and humans dataset, you're ready to train. When you defined the model, you saw that you were using a new loss function called 'Binary Crossentropy', and a new optimizer called RMSProp. If you want to learn more about the type of binary classification we are doing here, check out this great video from Andrew!

Experiment with the horse or human classifier

Now it's your turn. You can find the notebook here. Work through it and get a feel for how the ImageGenerator is pulling the images from the file system and feeding them into the neural network for training. Have some fun with the visualization code at the bottom!

In earlier notebooks you tweaked parameters like epochs, or the number of hidden layers and neurons in them. Give that a try for yourself, and see what the impact is. Spend some time on this.

Once you're done, move to the next video, where you can validate your training against a lot of images!

Get hands-on and use validation

Now you can give it a try for yourself. Here's the notebook that Laurence went through in the video. Have a play with it to see how it trains, and test some images yourself! Once you're done, move onto the next video where you'll compact your data to see the impact on training.

Get Hands-on with compacted images

Try this version of the notebook where Laurence compacted the images. You can see that training times will improve, but that some classifications might be wrong! Experiment with different sizes -- you don't have to use 150x150 for example!

Week 4 Resources

You used a few notebooks this week. For your convenience, or offline use, I've shared them on GitHub. The links are below:

Horses or Humans Convnet

Horses or Humans with Validation

Horses or Humans with Compacting of Images