Projects

Teach a computer to read

Build a machine vision model with TensorFlow to teach a computer to recognise handwritten numbers.

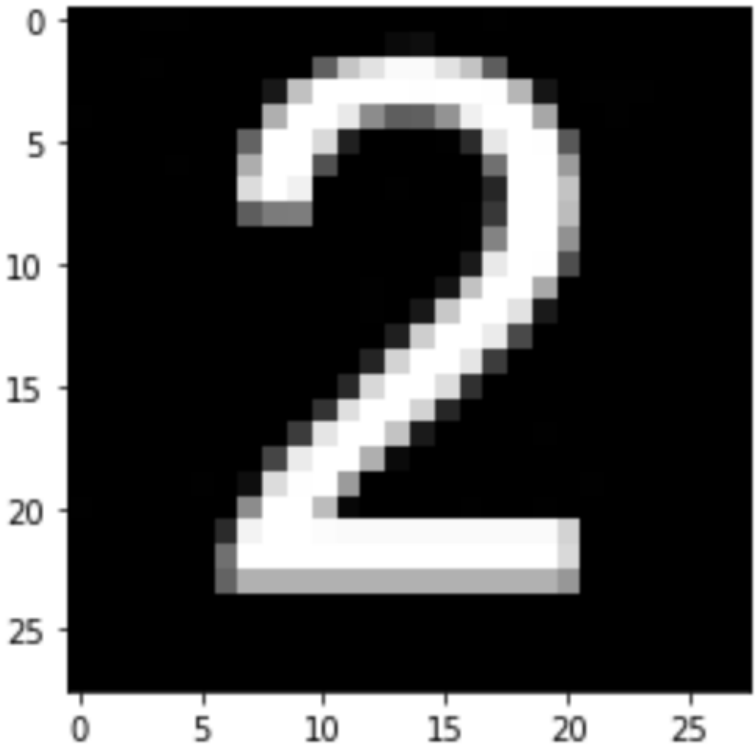
Step 1 Introduction

In this project, you'll train a computer to recognise the numbers between zero and nine. You'll also learn to measure how well your machine vision model performs the job you have trained it to do.

What you will make

You will create a TensorFlow model in a Google Colab notebook that can recognise the numbers between zero and nine from pictures of those numbers.

```
Downloading data from http://dojo.soy/num2
8192/4602 [==================================
Your model predicts this number is 2
```



What you should already know

What you will need

What you will learn

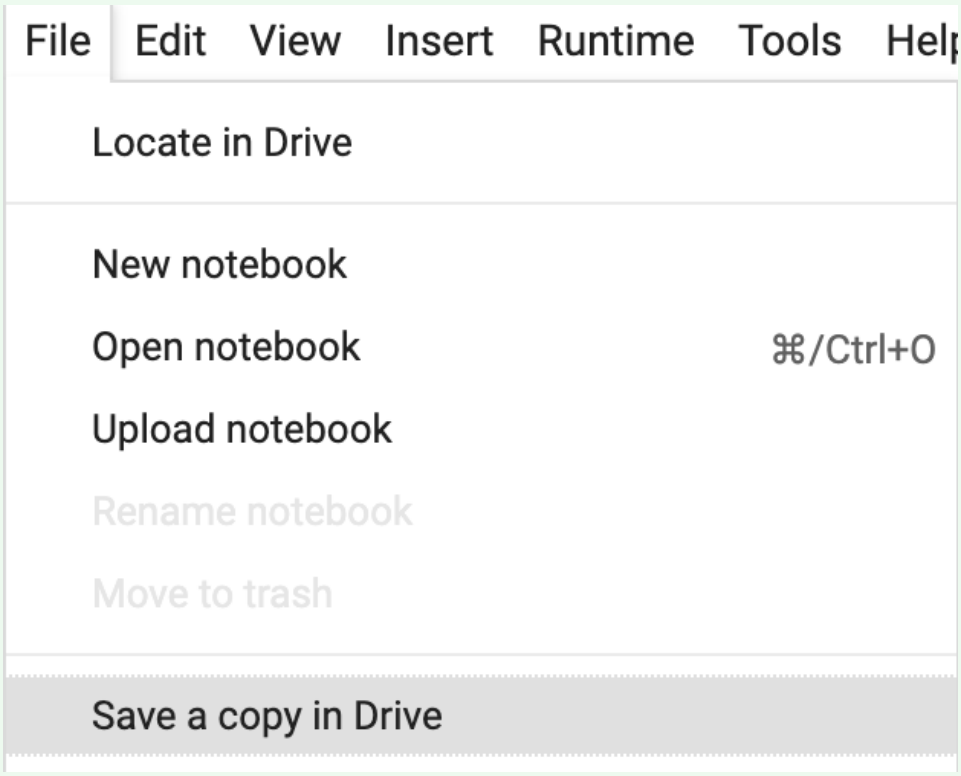
Additional information for educators

Step 2 Get some training data

Before you start to train your model, you need to open the starter Colab notebook for this project and save a copy to your Google Drive.

Open the Google Colab starter notebook (https://colab.research.google.com/drive/10WmCJwhHJhhsLn4truUUS9Y9bLr5X-jO?usp=sharing) for this project in a new tab in your browser.

Before you change anything, make sure you save the notebook to your drive so you can keep your work! Choose `File > Save a copy in Drive` and sign in to your Google account if prompted.



To train a model — the process of having a model learn rules for classifying things from a series of examples — you'll need a lot of example data for it. Since you want the model to identify handwritten numbers from zero to nine, you'll need lots of images of those numbers written by different people; they should be properly labelled so that the model can check its own accuracy as it trains.

Conveniently, TensorFlow already contains a dataset like this, which you can load into your program with a couple of lines of code.

First, you need to load the dataset into a variable. In the first empty cell, create a variable called `numbers` and assign the dataset to it like this:

```
numbers = tf.keras.datasets.mnist
```

`mnist` is just the name of the dataset of numbers you're loading. It stands for 'Modified National Institute of Standards and Technology', which is the name of the group that put it together.

Now that the data is loaded, you need to put it into some variables so you can use it later. Because the `load_data` function returns two tuples, you need to supply the variable names as tuples too.

Using Python tuples

Below the previous line, assign the output of the `load_data` function to variables for training and validation images and labels.

```
(training_images, training_labels), (validation_images,
validation_labels) = numbers.load_data()
```

Save your project

Step 3 Explore and prepare your data

Now that you have your data, you're going to take a look at it and do a little work to convert it from a human-readable form to one that suits your model better.

First, just so you know what the data looks like, you need to print out an image and the associated label. For this, you can use `plt.imshow`, which is a function that was imported as part of `matplotlib.pyplot` up at the top of the notebook. Note that this function usually displays colour images, and the MNIST dataset is made up of greyscale (only black, white, and grey) images, so this is something you'll need to tell the `imshow` function.

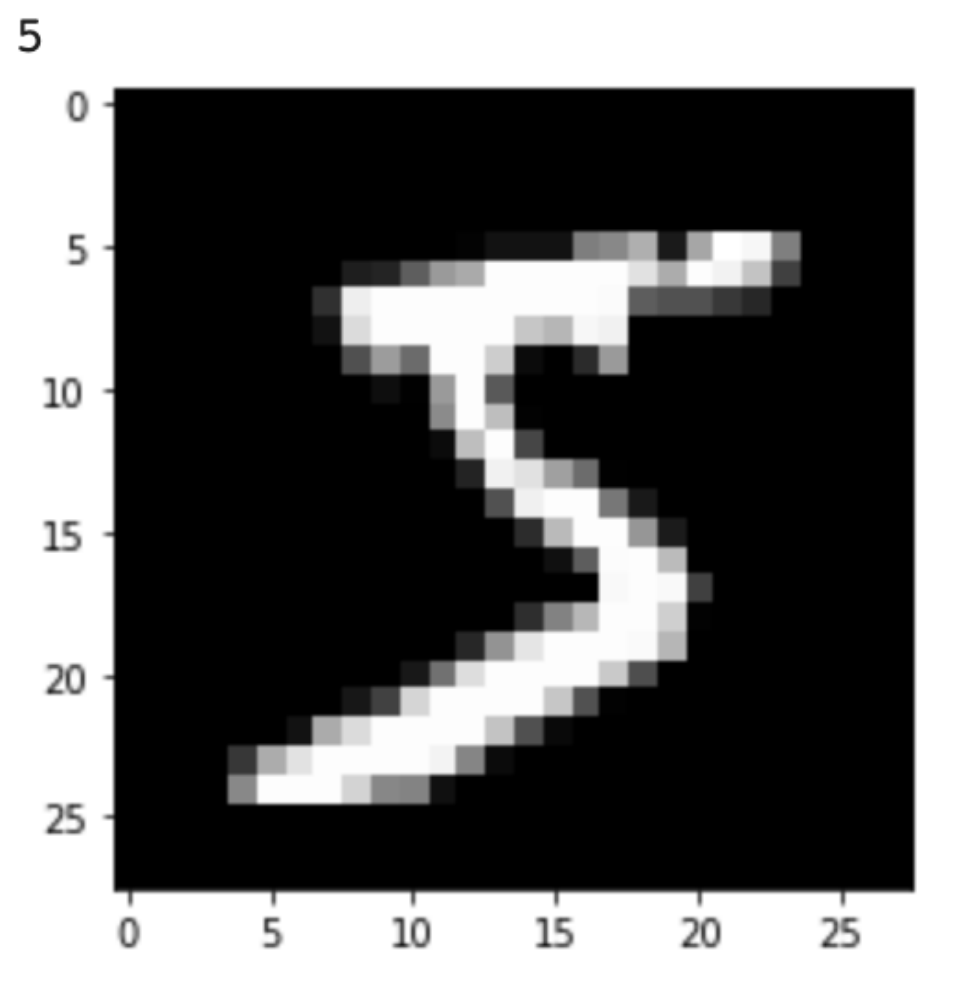
In the next empty cell, call the `plt.imshow` function, and pass it the first training image, as well as the parameter `cmap= 'gray'` to tell it to use the colour map for a greyscale image. Note that the American spelling of 'gray' must be used. Print out the first training label too.

```
plt.imshow(training_images[0], cmap='gray')
print(training_labels[0])
```

Now run the code you've written by going to the `Runtime` menu and choosing `Run all`.

You should see the label, in this case the number five, followed by an image of a five that looks handwritten.

You'll also see the numbers 0 to 25 along the edges of the image. These are added automatically every five pixels by the `plt.imshow` function and they show the size of the image — 28 pixels wide by 28 pixels high.



The images are stored in the `training_images` list as numbers. Since you'll work with those numbers in a moment, it's helpful to understand what they look like.

Just below your code to print out the image label, print the image using the standard Python `print` function.

```
print(training_images[0])
```

Run the code again.

As well as what you saw before, you should see a list of lists of numbers. The numbers are mostly zeros, and never over 255.

```
 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0   0   0   0   0   0   0   0   0]
 0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136
175  26 166 255 247 127   0   0   0   0]
 0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253
225 172 253 242 195  64   0   0   0   0]
```

It can be hard to see how these are your image, but if you copy them into a text editor and clean them up so that each list is on a line of its own, it becomes clearer. To save you having to do that, here's what it looks like:

```
[[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  3 18 18 18 126 136 175 26 166 255 247 127  0  0  0  0]
 [ 0  0  0  0  0  0  0  0 30 36 94 154 170 253 253 253 253 253 225 172 253 242 195 64  0  0  0  0]
 [ 0  0  0  0  0  0  0 49 238 253 253 253 253 253 253 253 253 253 251 93 82 82 56 39  0  0  0  0]
 [ 0  0  0  0  0  0  0 18 219 253 253 253 253 253 198 182 247 241  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0 80 156 107 253 253 205 11  0 43 154  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0 14  1 154 253 90  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0 139 253 190  2  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  11 190 253 70  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0 35 241 225 160 108  1  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0 81 240 253 253 119 25  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0 45 186 253 253 150 27  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 16 93 252 253 187  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 0 249 253 249 64  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 46 130 183 253 253 207  2  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0 39 148 229 253 253 253 250 182  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0 24 114 221 253 253 253 253 201 78  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0 23 66 213 253 253 253 253 198 81  2  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0 18 171 219 253 253 253 253 195 80  9  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0 55 172 226 253 253 253 253 244 133 11  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0 136 253 253 253 212 135 132 16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

There are 28 lists, each with 28 numbers. Each list is a row of pixels, and each number represents a pixel in the image. You can see that the higher the number is, the brighter the corresponding pixel. The zeros appear black, while 255 is the brightest white. Colour images work the same way, but they use three layers of these lists, one for each of red, green, and blue. The combination of those layers can make almost 17 million different colours!

RGB colours

However, machine vision models like this one work best when all of the numbers they handle are between zero and one. Sometimes, this means you have to reshape the data you feed into them.

To reshape your images, you need to divide all of the numbers by 255 to give you values between zero and one. With a normal list, you would have to change each value individually, either in a loop or a list comprehension. However, `numbers.load_data()` did not return normal lists. The function returned numpy arrays, which behave a lot like lists, but have some extra features that are very useful for machine learning work. One of those extra features is that you can do the same piece of maths to every item in the array all at once.

In the next cell, below your plotting and printing, add these two lines to divide every item in the `training_images` and `validation_images` arrays by 255.

```
training_images = training_images / 255.0
validation_images = validation_images / 255.0
```

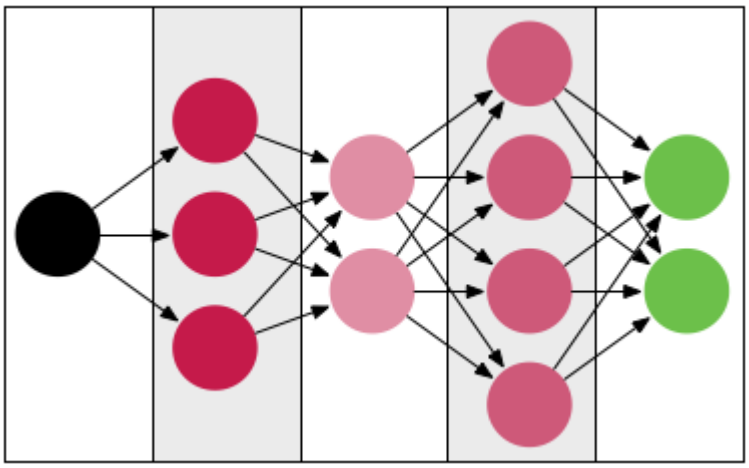
Save your project

Step 4 Build your model

Now that you have images prepared, it's time to build a model to recognise them. The image classifier you'll build is a kind of neural network. Neural networks get their name because they behave a little like our brains, which are made of neurons. However, a neural network that you build in a computer uses nodes instead of neurons. The nodes are organised into layers, with every node in a layer being connected to every node in the next layer.

The first layer takes the input, in this case the image of a number, and the last layer provides the output, in this case the likelihood that the input is each of the numbers from zero to nine. Treat the highest likelihood as the model's prediction, or best guess, as to what the image is.

Each node learns a rule, and produces a measure of the strength of the match from its inputs to the rule it has learned, which it passes on to every node it is connected to as an input. Each node is connected to every node in the next layer.



Your model has four layers:

- The first will take the image in and convert it to a flattened list of numbers, but it won't actually learn any rules. The other three layers will.
- The second layer receives the flattened image as an input and will create simple rules. Maybe these rules detect things like a curve in the top right of the image, or a curve in the bottom left of the image. The second layer will pass its outputs to the third layer as that layer's inputs.
- The third layer might contain a rule that combines both of those nodes as inputs and detects a curve in *both* the top right and bottom left of an image.
- Finally the fourth layer, the output layer, might use the input from that third layer node to make a strong prediction that the number is an eight.

While it's helpful to have some idea of how the network 'thinks', it's not worth spending too much time trying to figure out the sorts of rules it might create. The rules created by machine learning neural networks are usually quite strange, and make little sense to humans, but work very well for machines.

In the next empty cell in the notebook, create your model with one `Flatten` layer, to take a 28 by 28 pixel image and convert it into a single list of 784 (28 times 28) numbers, as well as three `Dense` layers to create rules and provide your outputs. The last layer must have ten nodes, as there are ten numbers. For the others, choose 500 and 300. These are good sizes for this problem, but once you've got the model trained with them you can try others and see what difference it makes.

```
model = tf.keras.Sequential([

tf.keras.layers.Flatten(input_shape=(28,28)),
                                    tf.keras.layers.Dense(500,
activation='relu'),
                                    tf.keras.layers.Dense(300,
activation='relu'),
                                    tf.keras.layers.Dense(10,
activation= 'softmax')
                                    ])
```

The `activation='relu'` you see in the dense layers above is a particular function — called an activation function — that the outputs of the nodes in the layer are passed through before being passed as inputs to the next layer. The function is run for each node, and it decides whether or not the node is used as part of the input to the next layer. Relu is the default activation function, the one you pick when you don't have a reason to pick another. Softmax, the other activation function used here, converts the numbers in the final layer into probabilities that add up to 100 percent. The greatest of these probabilities is the most likely guess for the number shown in the image given to your model.

Save your project

Step 5 Compile your model

Now that you've designed your model, you need to compile it. Compiling a model prepares it for training; it gives it instructions on how to improve itself — which is called the model's optimizer — and on how to measure that improvement using a loss function — a mathematical rule that the model uses to tell how well it's doing in training.

The optimizer works to make the loss as small as possible. It tries out different rules as it passes through groups of images, called batches from the training data. At the start of a batch, the optimizer adjusts the rules based on how well the loss function told the optimizer the model worked in the last batch (the first batch will use random rules). Then it feeds every image into the model with the new rules and checks its losses again. It repeats until it has used all of the training data and then goes through the whole set of training data again, as many times as you tell it to. Each complete pass through the training data is called an epoch. The optimizer also uses the validation data to make sure it is not learning rules that work only on the training data. This problem is known as overfitting, because training a model is sometimes called fitting it.

Programmers don't usually write their own optimizers or loss functions. They use functions created by experts in mathematics and machine learning. Choosing the right ones may take some research, and a little experimentation — for example training the model using two different optimizers and measuring which produces the best result. In the case of this problem, the Adam optimizer is a good choice, as is a loss function called sparse categorical cross entropy loss. Both of these are built into TensorFlow for you.

In the next empty cell, add this code to compile your model and see a summary of its structure.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])

model.summary()
```

The metric you've added is 'accuracy' — this is a percentage measure of how many images the model guessed correctly. The optimizer knows this by checking the image against its label after the model has made a guess. When you train the model, you'll watch this value go up. A score of 1.0 for accuracy is 100 percent, it would mean the model got everything right.

Usually there are some unusual examples in the data, which are called outliers. A well-trained model may still get them wrong, and so would only have an accuracy score between 0.9 and 1.0, but this is not a bad thing. Getting 1.0 would usually suggest that the model has learned some rules that only apply to the training data and so has become overfitted.

Run all the code and look at the model summary.

You should see something like the image below, which shows you each of your four layers.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               392500
_________________________________________________________________
dense_4 (Dense)              (None, 300)               150300
_________________________________________________________________
dense_5 (Dense)              (None, 10)                3010
=================================================================
Total params: 545,810
Trainable params: 545,810
Non-trainable params: 0
_________________________________________________________________
```

The first column is the type of the layer; the second is the outputs it provides to the next layer; and third is the number of parameters for that layer, which controls the model's rules that the optimizer will adjust. At the bottom, you can see the total number of parameters, and how many of them you are training — in the case of this model it's over half a million!

 Save your project

Step 6 Train your model

Now that it's compiled, your model is ready for training. Since you're also going to graph the model's performance once it's done, you want to store the output of the training function, the history of the training, in a variable too.

Create a `history` variable and assign the output of `model.fit()` to it (remember that training is sometimes called fitting). Pass the training and validation data you prepared, along with the size of the batches you want to break the data into, and the number epochs — complete runs through the data — you want the model to do before it finishes training. A batch size of 100 and 10 epochs seem to work well for this model.

```
history = model.fit(training_images,
                    training_labels,
                    batch_size=100,
                    epochs=10,
                    validation_data = (validation_images,
validation_labels)
                    )
```

Below the call to `model.fit()`, add a call to the `plot_accuracy_and_loss()` function provided with the notebook, to produce a graph of how your model improved over time. Pass it the `history` variable you just created.

```
plot_accuracy_and_loss(history)
```

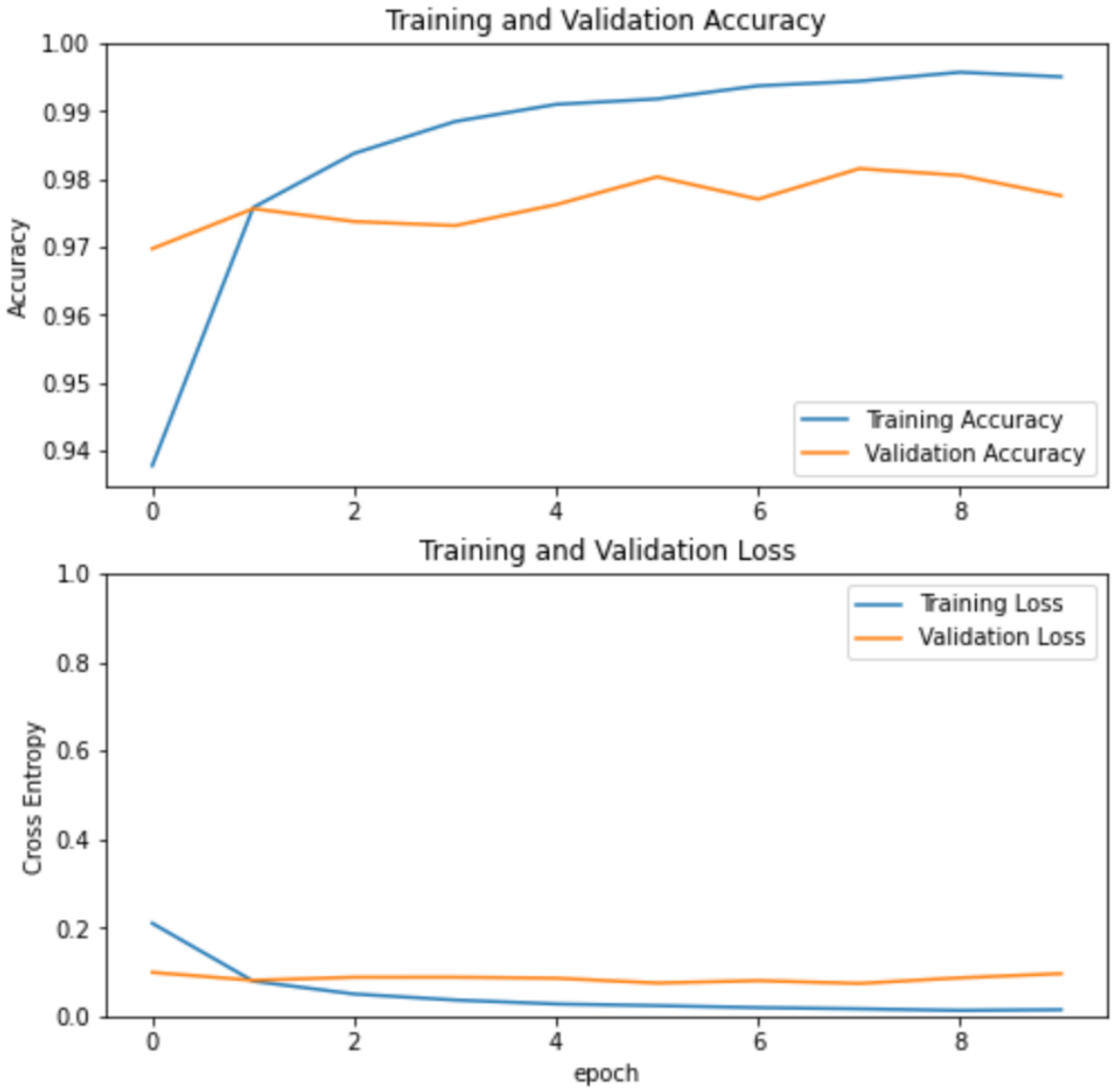
Run all the code.

You will probably have to wait about a minute for training to complete. Watch the loss reduce and the accuracy increase as each batch and epoch completes. There are 600 batches in an epoch, because there are 60,000 images in your data.

Once the training is finished, you should see some output like that in the image below, though your numbers may be slightly different.

```
Epoch 1/10
600/600 [==============================] - 7s 11ms/step - loss: 0.2100 - accuracy: 0.9377 - val_loss: 0.0988 - val_accuracy: 0.9697
Epoch 2/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0790 - accuracy: 0.9758 - val_loss: 0.0809 - val_accuracy: 0.9756
Epoch 3/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0500 - accuracy: 0.9837 - val_loss: 0.0879 - val_accuracy: 0.9737
Epoch 4/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0363 - accuracy: 0.9884 - val_loss: 0.0880 - val_accuracy: 0.9731
Epoch 5/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0275 - accuracy: 0.9910 - val_loss: 0.0856 - val_accuracy: 0.9762
Epoch 6/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0237 - accuracy: 0.9918 - val_loss: 0.0749 - val_accuracy: 0.9803
Epoch 7/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0192 - accuracy: 0.9937 - val_loss: 0.0802 - val_accuracy: 0.9770
Epoch 8/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0165 - accuracy: 0.9944 - val_loss: 0.0739 - val_accuracy: 0.9815
Epoch 9/10
600/600 [==============================] - 6s 11ms/step - loss: 0.0129 - accuracy: 0.9957 - val_loss: 0.0865 - val_accuracy: 0.9805
Epoch 10/10
600/600 [==============================] - 7s 11ms/step - loss: 0.0144 - accuracy: 0.9950 - val_loss: 0.0960 - val_accuracy: 0.9775
```

You should also see a graph that shows the improvement of your model over the course of the training, as accuracy increased and loss decreased.



The left-hand side of the graph shows where the accuracy or loss started at the end of the first epoch, moving right as epochs passed. You can see that the slope of the line was initially very steeply up for accuracy, and steeply down for loss. Over time, the changes became less steep as the model went from making major discoveries about important rules, to making minor improvements on rules it had learned in an earlier epoch.

Step 7 Test your model

Now that you've trained your number recognition model, it's time to see how well it does on some numbers you create!

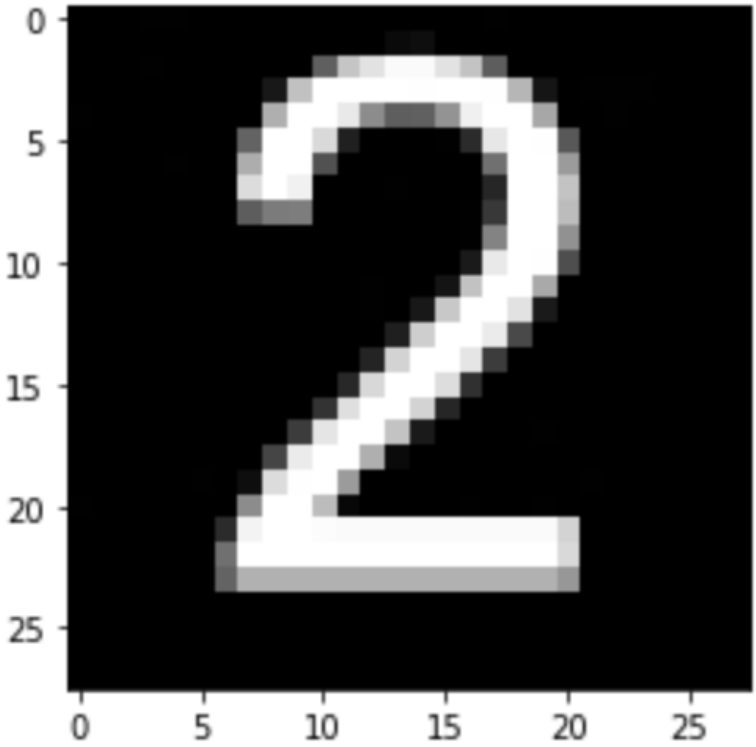
In the last empty cell, add a call to `predict_image` and pass it the URL of the test image at http://dojo.soy/num2 (http://dojo.soy/num2).

```
predict_image('http://dojo.soy/num2')
```

Run just this cell, so your model doesn't train itself again. Click on the ▶ button that appears to the left of it.

You should see something like the image below, which gives your model's prediction and a preview of the image you loaded.

```
Downloading data from http://dojo.soy/num2
8192/4602 [==================================
Your model predicts this number is 2
```



Save your project

Use your favourite image editing program to make a square black `jpg` or `png` file with a single white digit in it. Try different fonts, or draw the number yourself. Since this is a small model that you trained pretty quickly, it will be wrong quite often, but it's possible to build and train much more accurate models when they're needed. If you don't have an image editor, you can use a website like autodraw.com (https://www.autodraw.com/), or download and install GIMP (https://www.gimp.org/downloads/).

Once you've saved your file, you'll need to host it online so Colab can access it. Follow the instructions below to see how to store them in your Google Drive and get the URLs that Colab can use.

Hosting an image in Google Drive

Published by (https://www.raspberrypi.org) under a (https://creativecommons.org/licenses/by-sa/4.0/).

(https://github.com/RaspberryPiLearning/teach-a-computer-to-read)

Published by Raspberry Pi Foundation (https://www.raspberrypi.org) under a Creative Commons license (https://creativecommons.org/licenses/by-sa/4.0/).
View project & license on GitHub (https://github.com/RaspberryPiLearning/teach-a-computer-to-read)