Convolutional Neural Network Tutorial

Leigh Whitehead

27th October 2023

8th UK LArTPC Software and Analysis Workshop

Slack channel: #deep-learning

Introduction

- This tutorial is independent from the previous days
- We will be doing everything in python
  - Python is the most popular language for deep learning and the majority of online resources use python
  - I know python will be alien to some of you…
- I don't have time to teach you python here, but I hope the code I provide is reasonably self-explanatory
  - Structures are mostly similar to C++ but with different syntax
- We will use tensorflow (via keras), but PyTorch is also a popular framework for deep learning

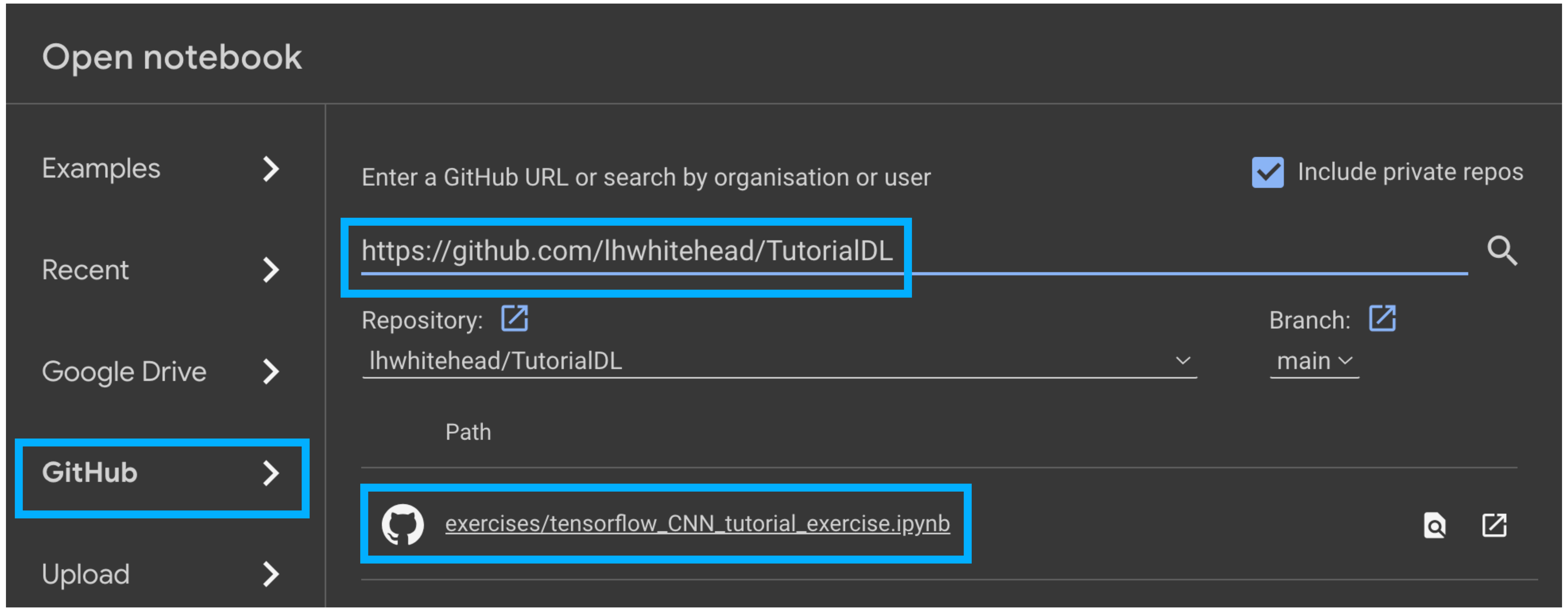
All you need to run this tutorial is a web-browser!

Python notebooks

- Today we will work with python notebooks (also called Jupyter notebooks)
- There are a few advantages for tutorials
  - No environment to set up or packages to install on your machine
  - The code can be interspersed with text and pictures
  - Each small block of code can be executed to show intermediate output
    - Click on a block to edit it
    - Press shift + enter to execute the code
- We will run in a web-browser using Google Colab

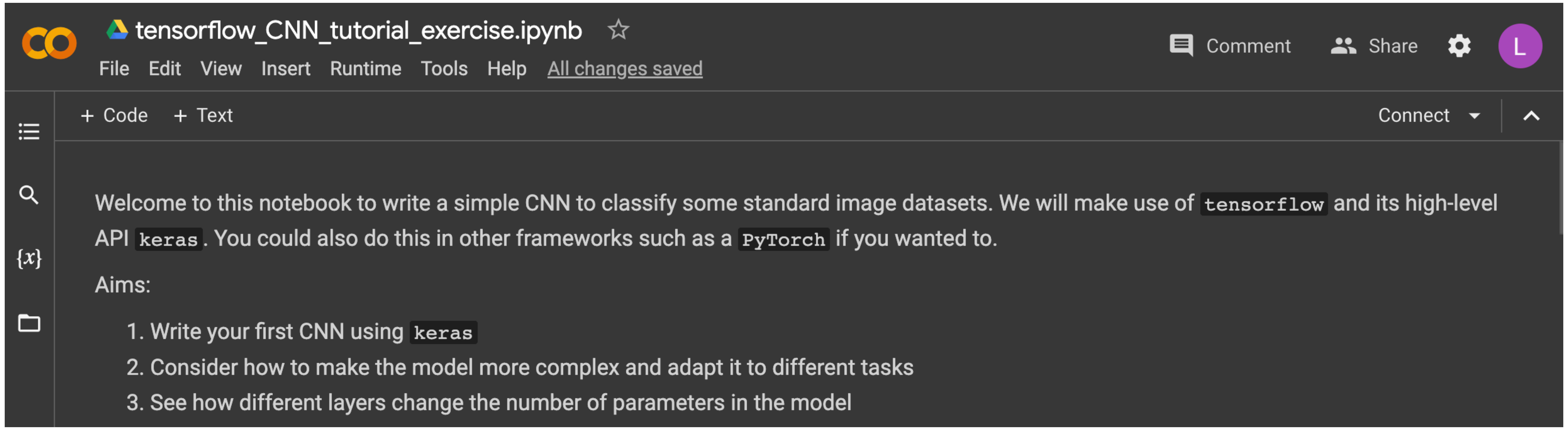
Google Colab

- Load Google Colab: https://colab.research.google.com
  - A popup to load a notebook will appear
  - Click on the GitHub tab
  - Enter this GitHub URL: https://github.com/lhwhitehead/TutorialDL
  - Select the exercise



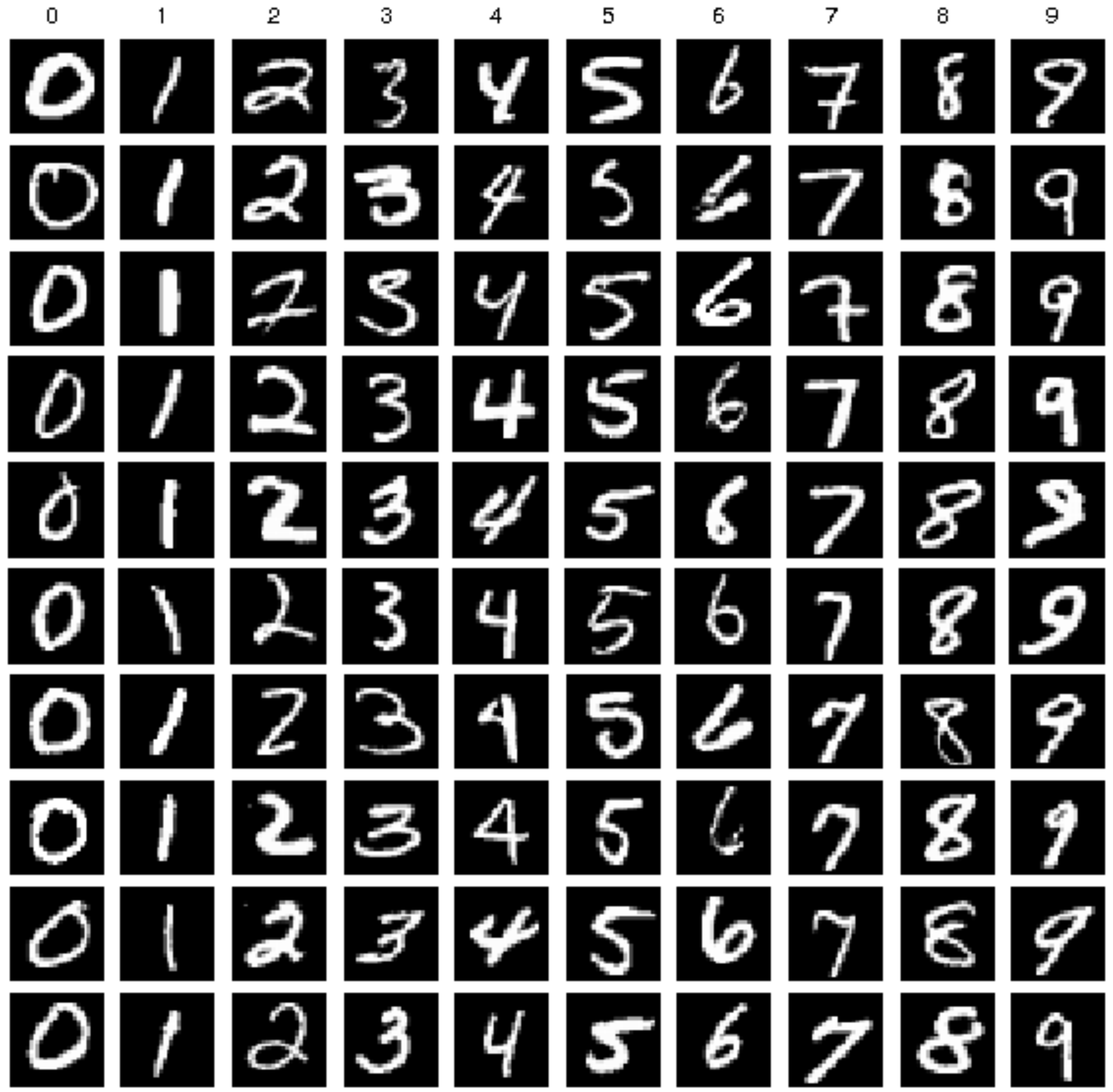
Google Colab

- Load Google Colab: https://colab.research.google.com
  - A popup to load a notebook will appear
  - Click on the GitHub tab
  - Enter this GitHub URL: https://github.com/lhwhitehead/TutorialDL
  - Select the exercise



The Aim

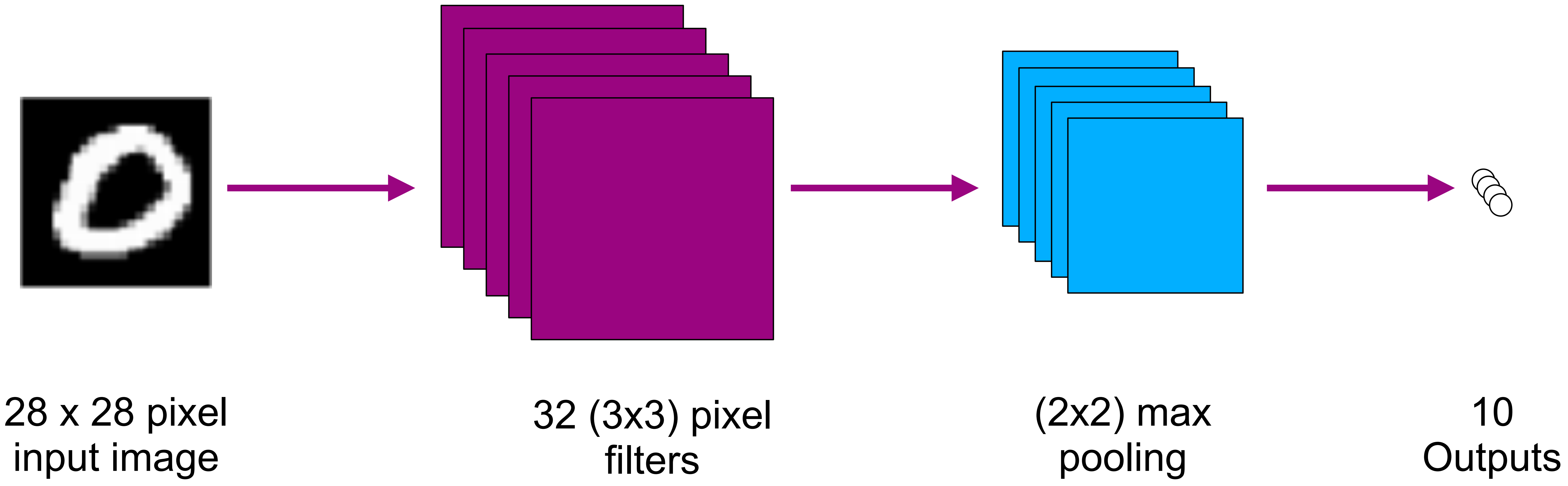
- We don't have time to use a large neutrino dataset to classify neutrinos
  - We will start by using a simple convolutional neural network to classify the MNIST benchmark data set
- MNIST is a collection of 70,000 handwritten digits from 0-9
- Each image is 28 x 28 pixels
- Has a target (truth) from 0-9
- Was a benchmark dataset for CNNs for a number of years



NB: This was the first use-case for a CNN! LeCun, Y., et al., Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541-551, 1989, https://doi.org/10.1162/neco.1989.1.4.541

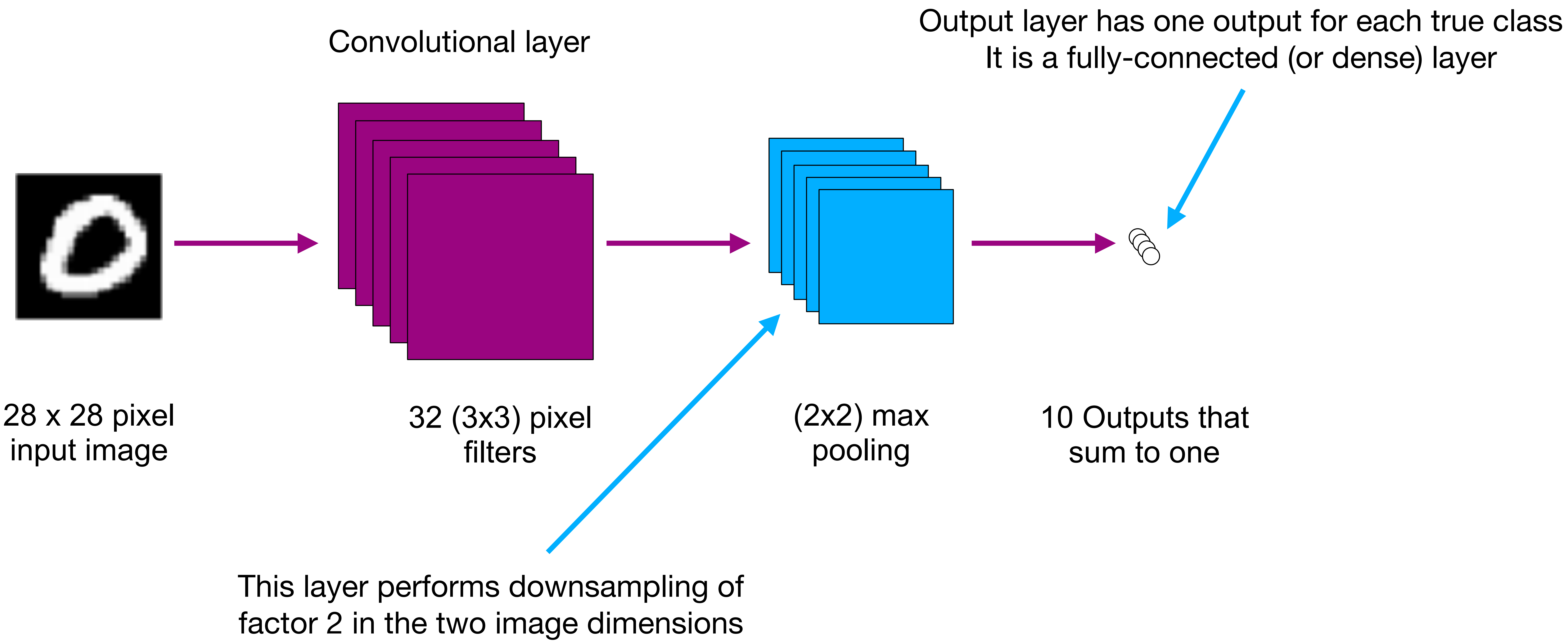
Our Network

- We will start with what is about the simplest CNN we can build



Our Network

- We will start with what is about the simplest CNN we can build



The exercise

- Ok, now we can play with something!
- You will see that the exercise notebook as a number of lines of code that just say `None`
  - These are the parts of the code that you need to fill in
  - I've provided some descriptions, explanations and hints to help you fill in the blanks
  - I'll also cover it in these slides as we go along
- First things first
  - Get your notebook loaded in Google Colab or Binder
  - We'll get started once you've all loaded it up

The exercise

- The first thing we need to do is load the required libraries
  - Run the block of code by selecting the box it is in and pressing shift + enter

```
import tensorflow
from tensorflow import keras
from keras.datasets import mnist, cifar10, cifar100
import numpy as np
import matplotlib.pyplot as plot

print('Tensorflow version:',tensorflow.__version__)
```

```
Tensorflow version: 2.13.0
```

  - You will see it print out the tensor flow version just to show it has done something
  - You might see a warning / error about GPUs… ignore this
  - Can think of these import statements like the #include statements in C++

Function to load some datasets

- The load_dataset function has the code to load the dataset
  - By default it will load the MNIST dataset, but it can also load CIFAR10 and CIFAR100 by providing an argument to the function call

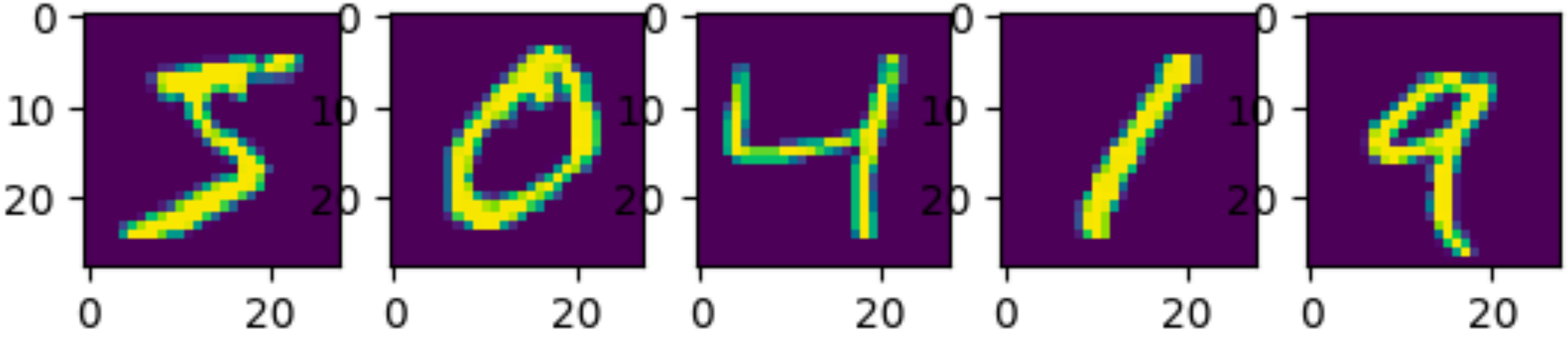
```
def load_dataset(dataset_name='mnist'):
  # MNIST, CIFAR10 and CIFAR100 are standard datasets we can load straight
  # from keras. The data are split between train and test sets automatically
  # - x_train is a numpy array that stores the training images
  # - y_train is a numpy array that stores the true class of the training images
  # - x_train is a numpy array that stores the testing images
  # - y_train is a numpy array that stores the true class of the testing images
  if dataset_name.lower() == 'cifar10':
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()
    n_classes = 10
  elif dataset_name.lower() == 'cifar100':
    (x_train, y_train), (x_test, y_test) = cifar100.load_data()
    n_classes = 100
  elif dataset_name.lower() == 'mnist':
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # MNIST is greyscale so we have to do a trick to add a depth dimension
    x_train = np.expand_dims(x_train, axis=-1)
    x_test = np.expand_dims(x_test, axis=-1)
    n_classes = 10
  else:
    print('Requested dataset does not exist. Please choose from mnist, cifar10 or cifar100')
    return
```

```
  # Let's check the shape of the images for convenience
  print("Shape of x_train =",x_train.shape)
  print("Shape of x_test =",x_test.shape)

  # The y_train and y_test values we loaded also need to be modified.
  # These values store the true classification of the images (0-9) as a single
  # number. We need to convert the single value into an array of length 10
  # corresponding to the number of output classes. Thus values of
  # y = 2 becomes y = [0,0,1,0,0,0,0,0,0,0]
  # y = 8 becomes y = [0,0,0,0,0,0,0,0,1,0]
  y_train = keras.utils.to_categorical(y_train, n_classes)
  y_test = keras.utils.to_categorical(y_test, n_classes)

  print("Shape of y_train =", y_train.shape)
  print("Shape of y_test =", y_test.shape)

  # Let's take a look at a few example images from the training set
  n_plots=5
  fig, ax = plot.subplots(1, n_plots)
  for plot_number in range (0, n_plots):
    ax[plot_number].imshow(x_train[plot_number])

  return (x_train, y_train), (x_test, y_test), n_classes
```

  - Returns numpy arrays of images and truth labels for training and test samples and the number of true classes

Function to load some datasets

- The load_dataset function has the code to load the dataset
  - By default it will load the MNIST dataset, but it can also load CIFAR10 and CIFAR100 by providing an argument to the function call
  - It will also print out the first five images from the dataset



  - These MNIST images are greyscale, but shown with a colour palette here

Function to load some datasets

- The load_dataset function has the code to load the dataset
  - Now we can just call the function to get our dataset:

```
[ ] # Load the input data.
    # x_train is the training data, and y_train the corresponding true labels
    # x_test is the testing data, and y_test the corresponding true labels
    # We don't have a separate validation sample in these keras datasets
    # Num_classes is the number of true classes
    (x_train, y_train), (x_test, y_test), num_classes = load_dataset('mnist')
```

  - The data are stored in x_train and x_test
  - The labels are stored in y_train and y_test

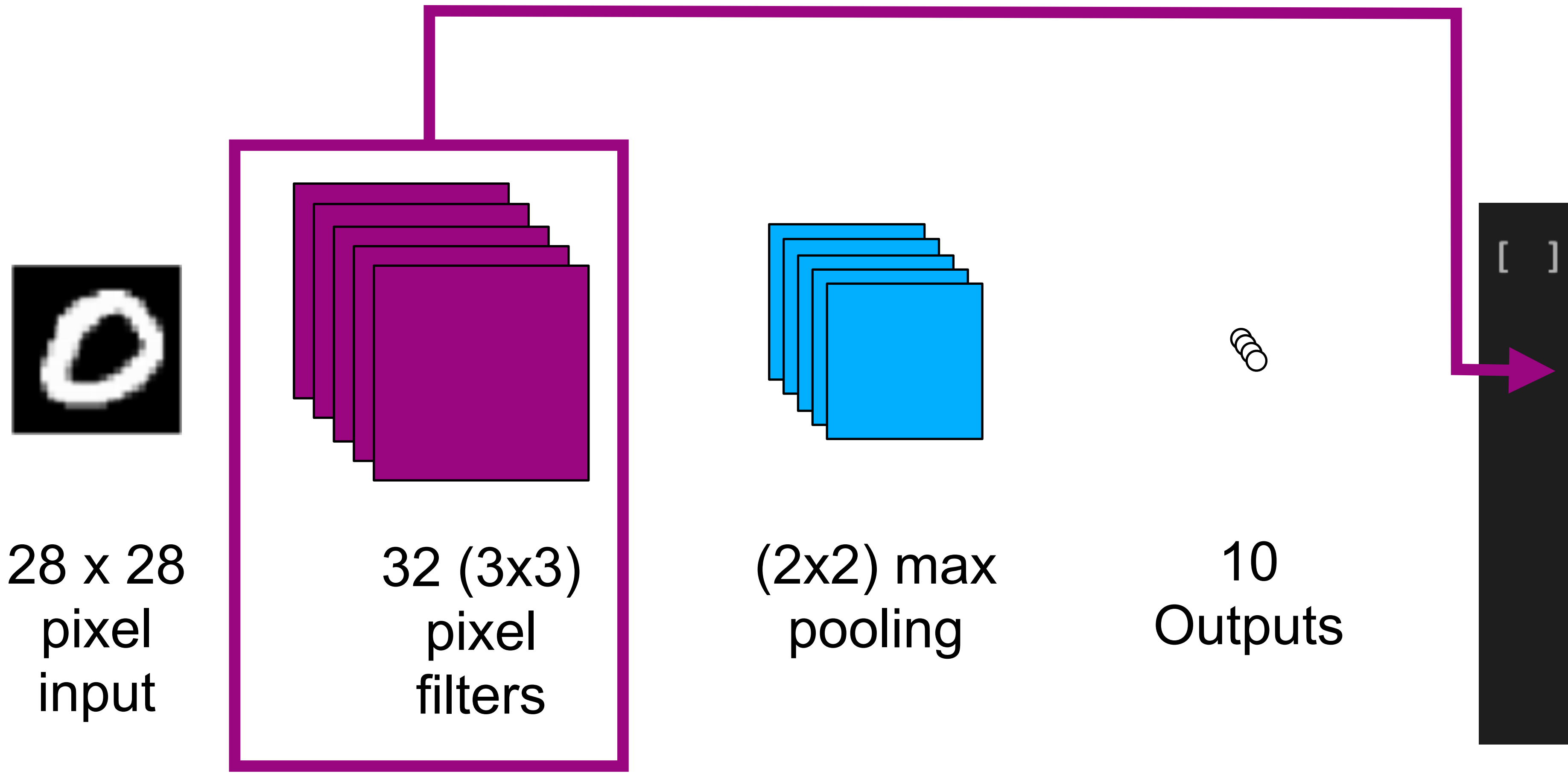
Building the CNN

- This large block of code is used to build our CNN
  - There are lots of blanks to fill in here!
  - I'll give some details in the following slides, but the comments in the notebook should give you all the information that you need

```
[ ]  # Define our simplest model first
     input_layer = keras.layers.Input(x_train[0].shape)
     x = None(input_layer) # Replace None with a 2D convolution with 32 filters of size (3,3) and relu activation
     x = None(x) # Replace None with a MaxPooling2D layer to downsample by a factor of 2 in both dimensions
     x = None(x) # Replace None with a droput layer with a fraction of 0.25
     x = keras.layers.Flatten()(x)
     x = None(x) # Replace None with a final dense output layer with num_classes neurons
     cnn_model = keras.Model(input_layer, x)
     cnn_model.summary()
```

Building the CNN

- Lets remember our network architecture…



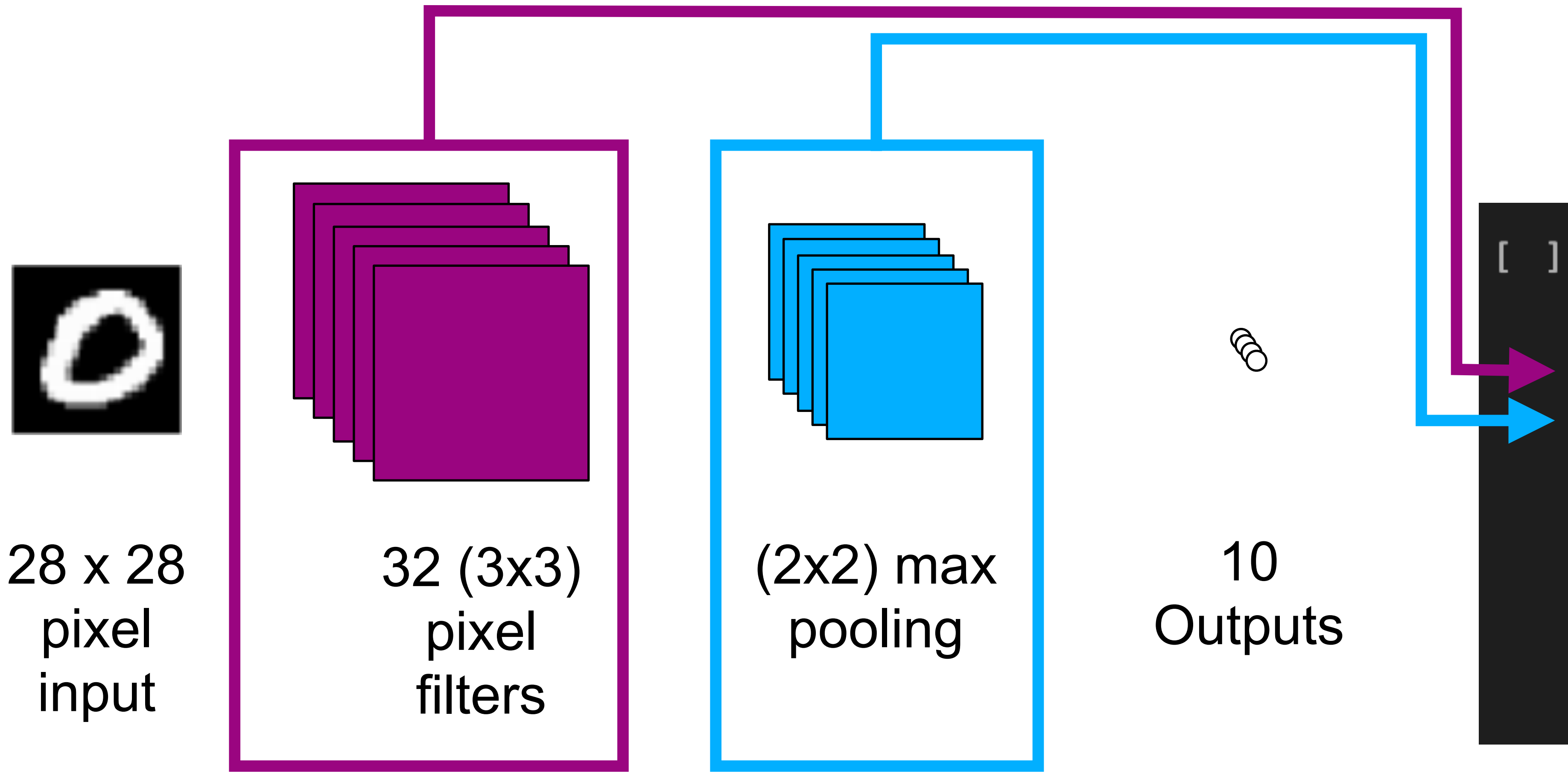

```
[ ] # Define our simplest model first
    input_layer = keras.layers.Input(x_train[0].shape)
    x = None(input_layer) # Replace None with a 2D convolution with 32 filters of size (3,3) and relu activation
    x = None(x) # Replace None with a MaxPooling2D layer to downsample by a factor of 2 in both dimensions
    x = None(x) # Replace None with a droput layer with a fraction of 0.25
    x = keras.layers.Flatten()(x)
    x = None(x) # Replace None with a final dense output layer with num_classes neurons
    cnn_model = keras.Model(input_layer, x)
    cnn_model.summary()
```

- I have already defined the input here

- You need to define the first convolutional layer using keras.layers.Conv2D(…)

Building the CNN

- Lets remember our network architecture…

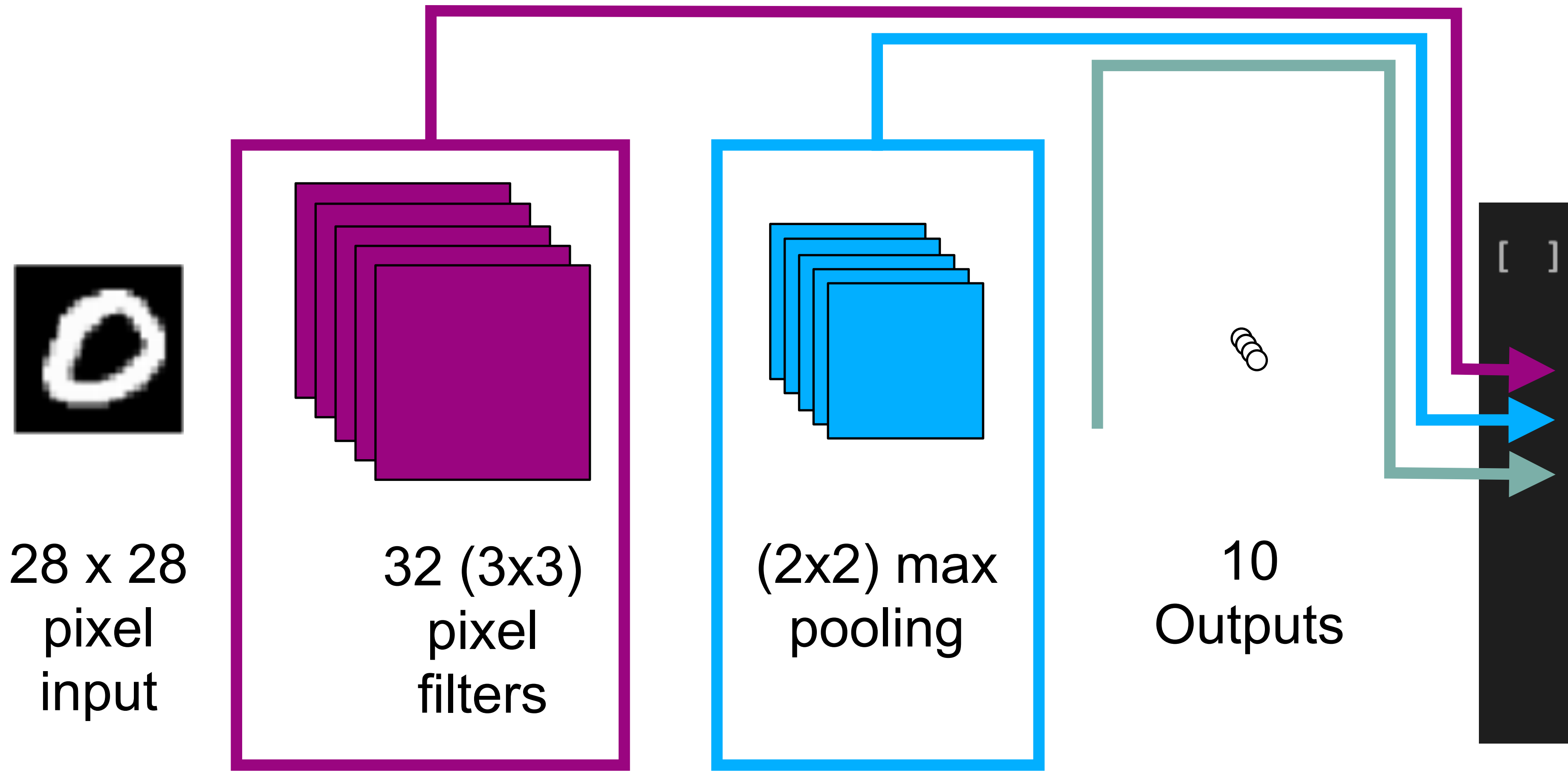



```
[ ] # Define our simplest model first
    input_layer = keras.layers.Input(x_train[0].shape)
    x = None(input_layer) # Replace None with a 2D convolution with 32 filters of size (3,3) and relu activation
    x = None(x) # Replace None with a MaxPooling2D layer to downsample by a factor of 2 in both dimensions
    x = None(x) # Replace None with a dropout layer with a fraction of 0.25
    x = keras.layers.Flatten()(x)
    x = None(x) # Replace None with a final dense output layer with num_classes neurons
    cnn_model = keras.Model(input_layer, x)
    cnn_model.summary()
```

- Next, define the pooling layer using keras.layers.MaxPooling2D(…)

Building the CNN

- Lets remember our network architecture…

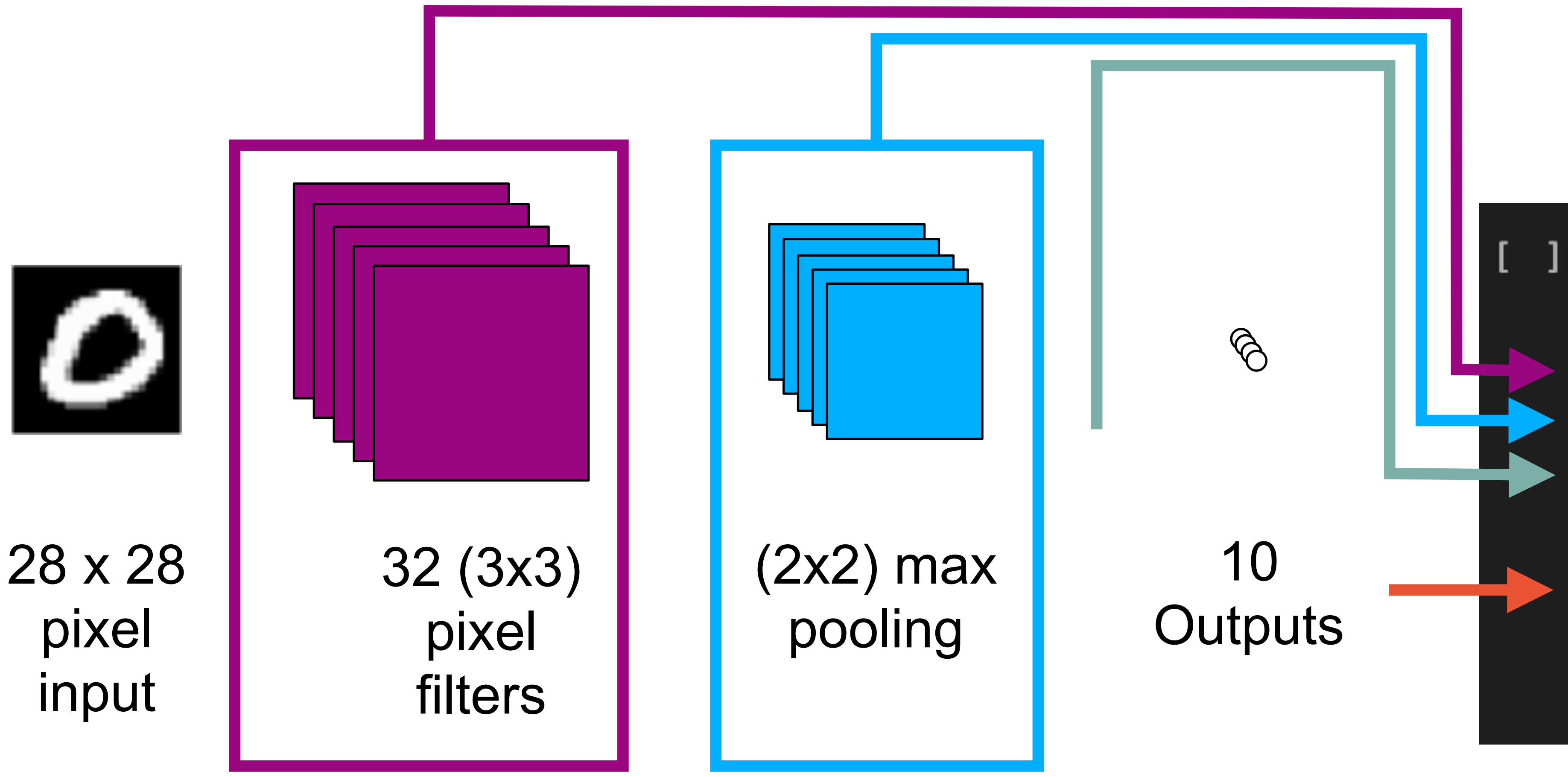



```
[ ] # Define our simplest model first
    input_layer = keras.layers.Input(x_train[0].shape)
    x = None(input_layer) # Replace None with a 2D convolution with 32 filters of size (3,3) and relu activation
    x = None(x) # Replace None with a MaxPooling2D layer to downsample by a factor of 2 in both dimensions
    x = None(x) # Replace None with a dropout layer with a fraction of 0.25
    x = keras.layers.Flatten()(x)
    x = None(x) # Replace None with a final dense output layer with num_classes neurons
    cnn_model = keras.Model(input_layer, x)
    cnn_model.summary()
```

- Next, define the dropout using keras.layers.Dropout(…)

Building the CNN

- Lets remember our network architecture…




```
[ ]  # Define our simplest model first
     input_layer = keras.layers.Input(x_train[0].shape)
     x = None(input_layer) # Replace None with a 2D convolution with 32 filters of size (3,3) and relu activation
     x = None(x) # Replace None with a MaxPooling2D layer to downsample by a factor of 2 in both dimensions
     x = None(x) # Replace None with a dropout layer with a fraction of 0.25
     x = keras.layers.Flatten()(x)
     x = None(x) # Replace None with a final dense output layer with num_classes neurons
     cnn_model = keras.Model(input_layer, x)
     cnn_model.summary()
```

- I've added the Flatten layer that takes the 2D tensor and makes it 1D
- Now you need to add the final output layer: keras.layers.Dense(…)
  - This layer needs to have a softmax activation

Building the CNN

- Once you've filled in the blanks and ran the code block you should see:
- Give a summary of the model:
  - Shows each layer:
    - Number of parameters
    - Shape of the data output
  - The total number of parameters

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 dropout (Dropout)           (None, 13, 13, 32)        0

 flatten (Flatten)           (None, 5408)              0

 dense (Dense)               (None, 10)                54090

=================================================================
Total params: 54410 (212.54 KB)
Trainable params: 54410 (212.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Defining some useful variables

- The next block of code defines some useful variables
  - See that some of these are hyper parameters like the learning rate

```
[ ] # The batch size controls the number of images that are processed simultaneously
    batch_size = 128
    # The number of epochs that we want to train the network for
    epochs = 5
    # The learning rate (step size in gradient descent)
    learning_rate = 0.001
```

  - As before, run it by pressing shift + enter
  - There isn't any output for this block of code

Training your CNN

- We need to tell the model how it should train
  - Which loss function? Which optimiser?

```
# Define the loss function - for a multi-class classification task we need to
# use categorical crossentropy loss
loss_function = keras.losses.categorical_crossentropy
# The optimiser performs the gradient descent for us. There are a few different
# algorithms, but Adam is one of the more popular ones
optimiser = keras.optimizers.Adam(learning_rate=learning_rate)
# Now we compile the model with the loss function and optimiser
cnn_model.compile(loss=loss_function, optimizer=optimiser, metrics=['accuracy'])
```

  - For n-category classification tasks we use categorical crossentropy loss
  - In this example, we will use the Adam optimiser

- Finally, we compile the model and it is ready to train

Training your CNN

- Now we train the CNN
  - Train on the training sample and use the testing sample for validation

```
# Train the model using the training data with the true target outputs.
# Fill in the required arguments using the clues given above
cnn_model.fit(x = None, y = None, batch_size = None, epochs = None,
              validation_data = (None, None), verbose = 1)
```

  - Fill in the blanks with the variables we defined in the exercise
  - When finished, hit shift + enter and you'll see it start to train
  - It should just take a few minutes to train for five epochs
  - You can watch the loss (hopefully) decrease as it trains

Running inference

- Now we are getting to the real way that your CNN will be used
- We want to classify images without knowing the truth information
  - We do this with the **model.predict(…)** function
- To make it a little more interesting, we will use model.predict as we search for incorrectly classified images

Running inference

- Now we are getting to the real way that your CNN will be used
- We want to classify images without knowing the truth information
  - We do this with the **model.predict(…)** function

```
# Make a list of incorrect classifications
incorrect_indices = []
# Let's look at the whole test dataset, but you can reduce this to 1000 or so
# if you want run more quickly
n_images_to_check = x_test.shape[0]
# Use the CNN to predict the classification of the images. It returns an array
# containing the 10 class scores for each image. It is best to write this code
# using the array notation x[:i] that means use all values of x up until
# the index i, such that if you changed the number of images above then it all
# still works efficiently
raw_predictions = cnn_model.predict(x = None, batch_size = None)
for i in range(0,n_images_to_check):
  # Remember the raw output from the CNN gives us an array of scores. We want
  # to select the highest one as our prediction. We need to do the same thing
  # for the truth too since we converted our numbers to a categorical
  # representation earlier. We use the np.argmax() function for this
  prediction = np.argmax(raw_predictions[i])
  truth = np.argmax(y_test[i])
  if prediction != truth:
    incorrect_indices.append([i,prediction,truth])
print('Number of images that were incorrectly classified =',len(incorrect_indices))
```

- You will need to just supply the correct images to the predict function
- See the hint on the a[:b] notation to get the first b elements of a

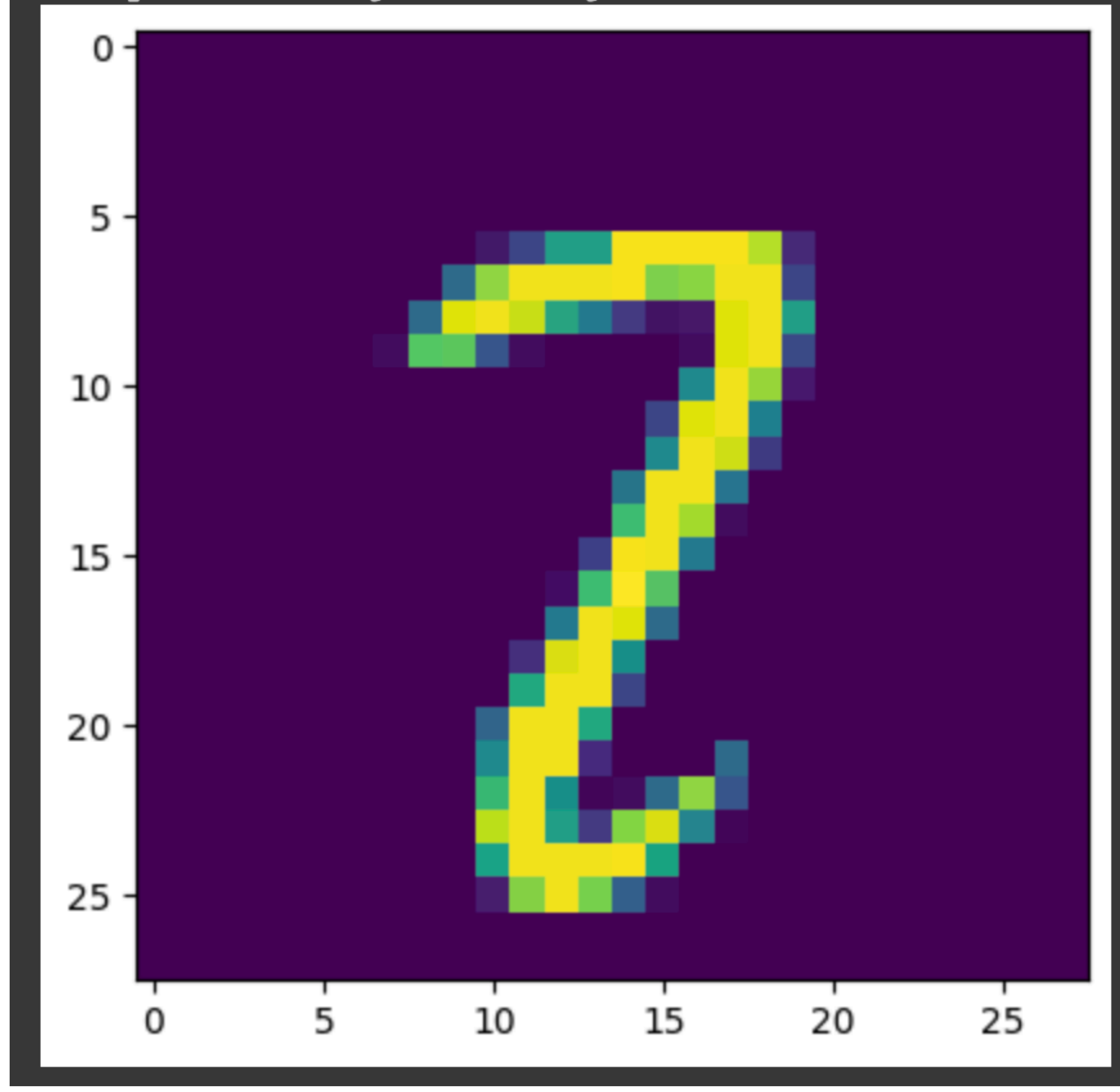
Checking the incorrect images

- The next block of code will visualise these failures

```
# Now you can modify this part to draw different images from the failures list
# You can change the value of im to look at different failures
im = 0
image_to_plot = x_test[incorrect_indices[im][0]]
fig, ax = plot.subplots(1, 1)
print('Incorrect classification for image',incorrect_indices[im][0],
      ': predicted =',incorrect_indices[im][1],
      'with true =',incorrect_indices[im][2])
ax.imshow(image_to_plot)
```

- You'll see an image alongside some information
- Change the value of im to see different images

```
Incorrect classification for image 321 : predicted = 7 with true = 2
<matplotlib.image.AxesImage at 0x7d4bbfb0d540>
```



A slightly tougher task

- Let's move on to the CIFAR10 dataset
  - This contains (very) low resolution colour images with 10 categories:

0	Aeroplane	5	Dog
1	Car	6	Frog
2	Bird	7	Horse
3	Cat	8	Ship
4	Deer	9	Lorry

- Do you notice any change in the architecture summary?
- How well does it perform compared to MNIST?

Have some fun and play around a bit

- There are lots of things you can do to add complexity to the model and see how well the classification works
  - Add more filters to the convolutional layer
  - Add a second (third, etc) convolutional layer
  - Add a dense layer with more neurons before the output?

Loading and saving models

- This isn't part of today's tutorial, but just for reference…
- To use our network in a realistic way we need to save it
  - You can use the model.save(<filename>) function for this
  - Similarly, model.load(<filename>) allows you to load a model
- For more information on all of the model functions:
  - https://www.tensorflow.org/api_docs/python/tf/keras/Model

Summary

- So, this brings me to the end of the tutorial
  - Use the File menu to save / download your finished exercise
- There are many things that I couldn't show you, but I hope this small introduction can help you get started with deep learning
  - There are lots of tutorials and resources online these days
- The other big framework is PyTorch
  - Some things are better supported in PyTorch as custom libraries
    - Graph neural networks (torch_geometric)
    - SparseCNNs (MinkowskiEngine by Nvidia, Facebook's SparseConvNet (less maintained))

Some thoughts (1)

- There aren't really any solid rules about what architecture is best for a certain job
- Hyperparameters are very important
  - The learning rate is probably the most important of all
- If the network learns but doesn't reach good accuracy it is possible that it is too simple and needs more layers or filters
- If your training accuracy is much higher than the validation accuracy then your network is likely overtrained… maybe add more dropout?
- Normalising your input parameters from (0,1) typically helps a lot to keep values "sensible" in the network (we didn't do this in the tutorial)

Some thoughts (2)

- Deep learning is not a replacement for brain power!
  - You need to think and try to understand why a certain approach will work for a given task
  - There isn't a golden architecture that will work for all use cases
- There are lots of resources online, so do some research when you have defined a problem that you want to solve
- Don't just start using CNNs for everything!