UNIVERSITY OF CAMBRIDGE

DEEP UNDERGROUND NEUTRINO EXPERIMENT

Convolutional Neural Network Tutorial

Leigh Whitehead

11th November 2022

7th 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

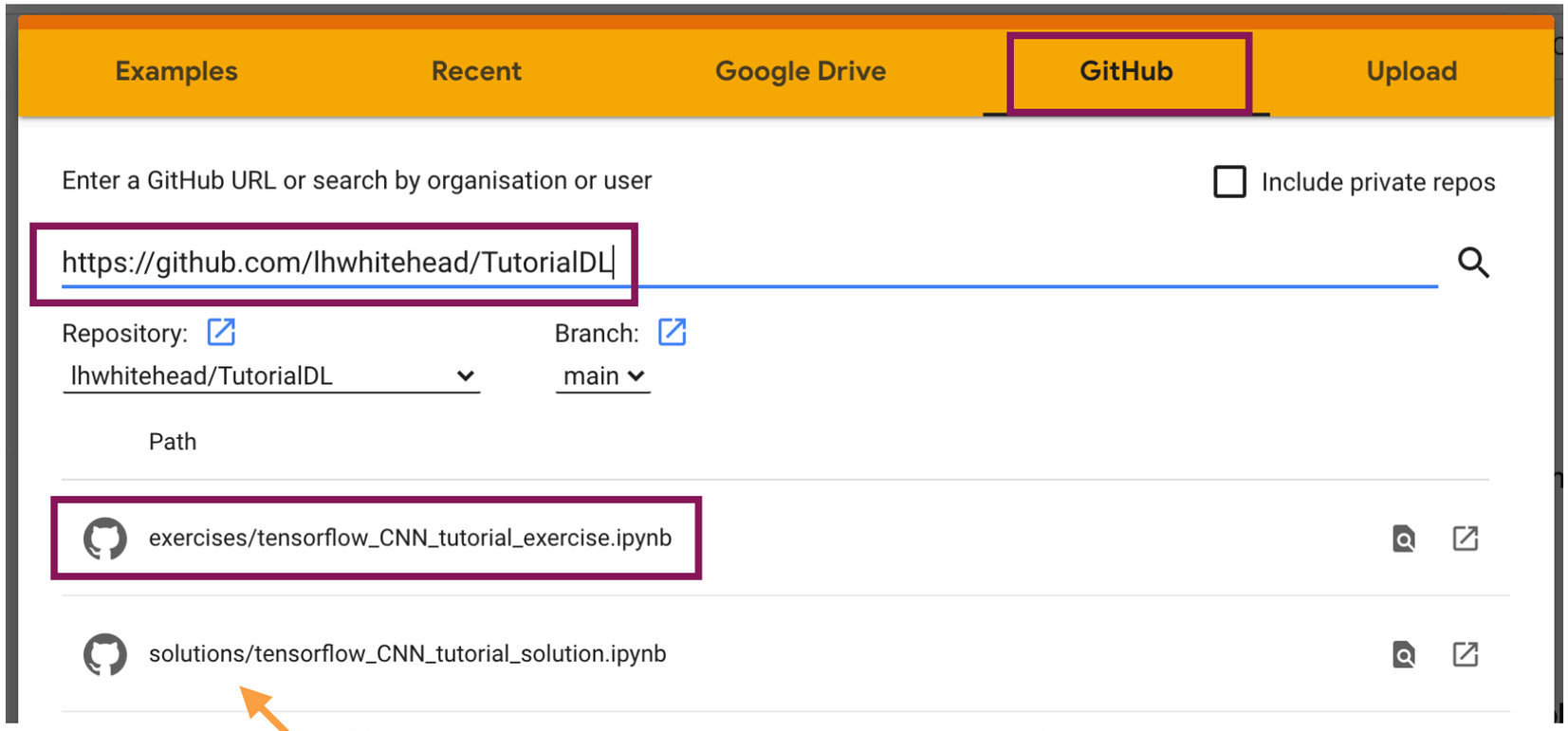
- Today we will work with python notebooks (also called Jupyter notebooks)

- 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 one of two methods:
  - Using Google Colab if you have a Google account
  - Binder (via a GitHub repository)
  - NB: the Google Colab machines seem to run ~2x faster

Google Colab

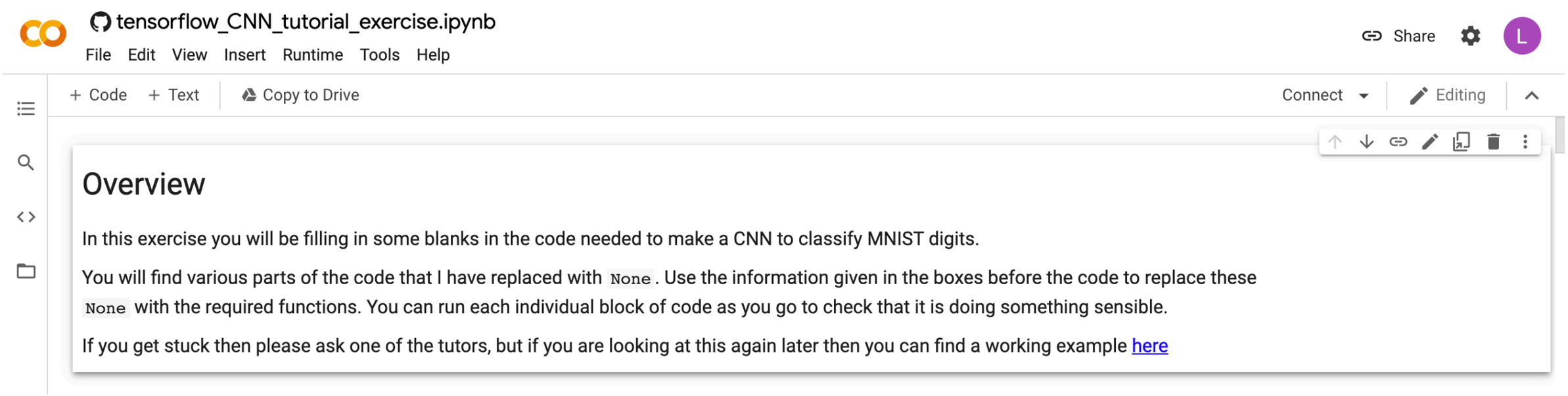
- Load Google Colab: https://colab.research.google.com
  - A popup to load a notebook will appear
  - Click on the GitHub tab
  - Enter this GitHub URL: https://github.com/lhwhitehead/TutorialDL
  - Select the exercise



NB: this is the solution… only look if you are completely stuck! Ask for help first.

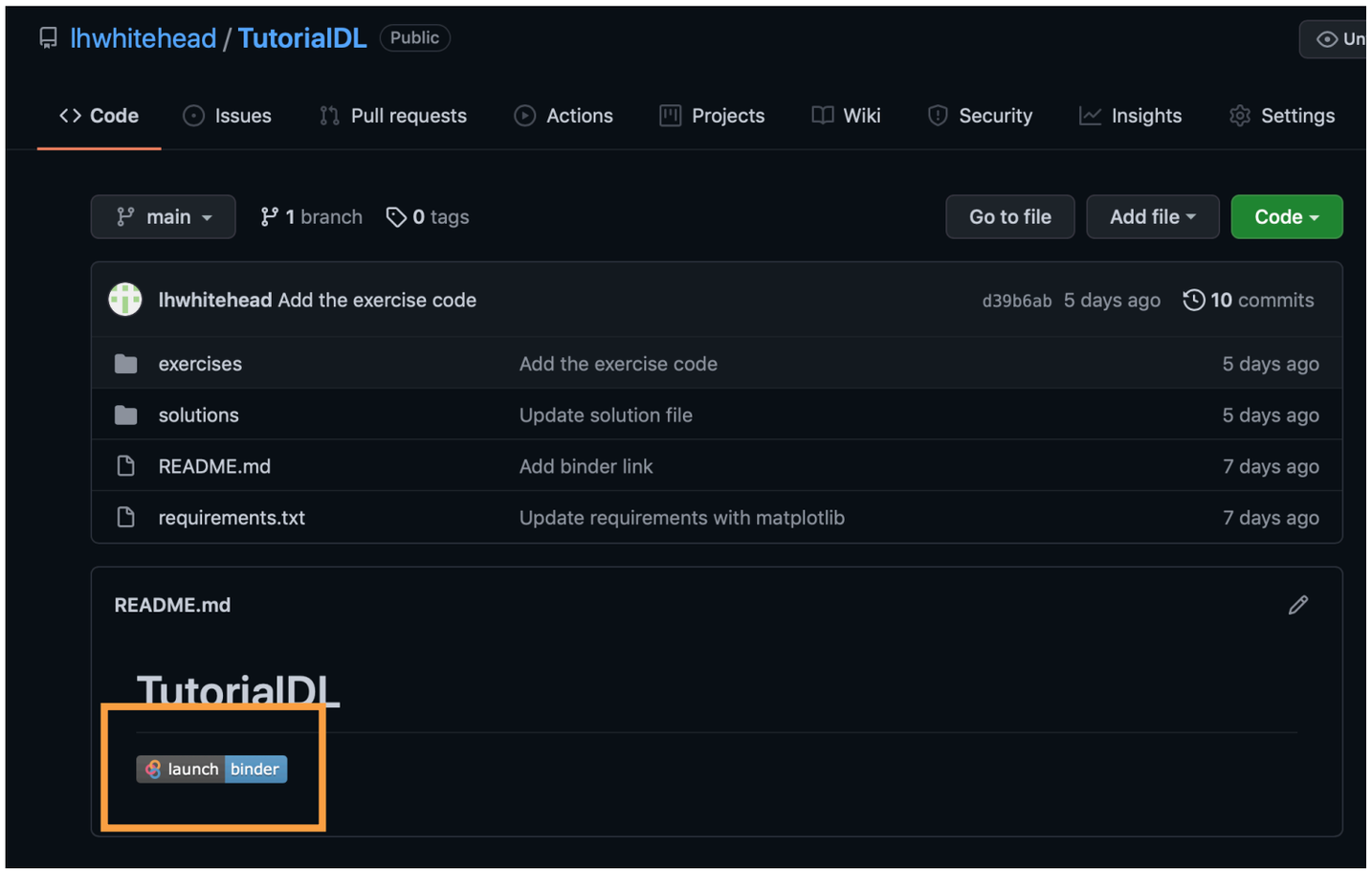
Google Colab

- 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
  - You should see something like this!



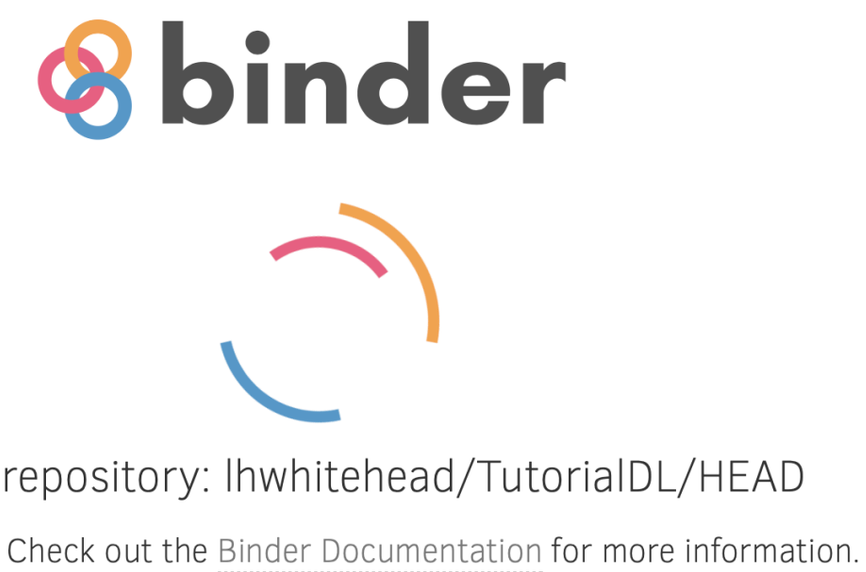
Binder

- You can run Binder directly from my GitHub
- Visit my GitHub page: https://github.com/lhwhitehead/TutorialDL
- Click on the launch binder button at the bottom
- Note that it seems about 2.5x slower to train the network here
- Also takes a couple of minutes to load up



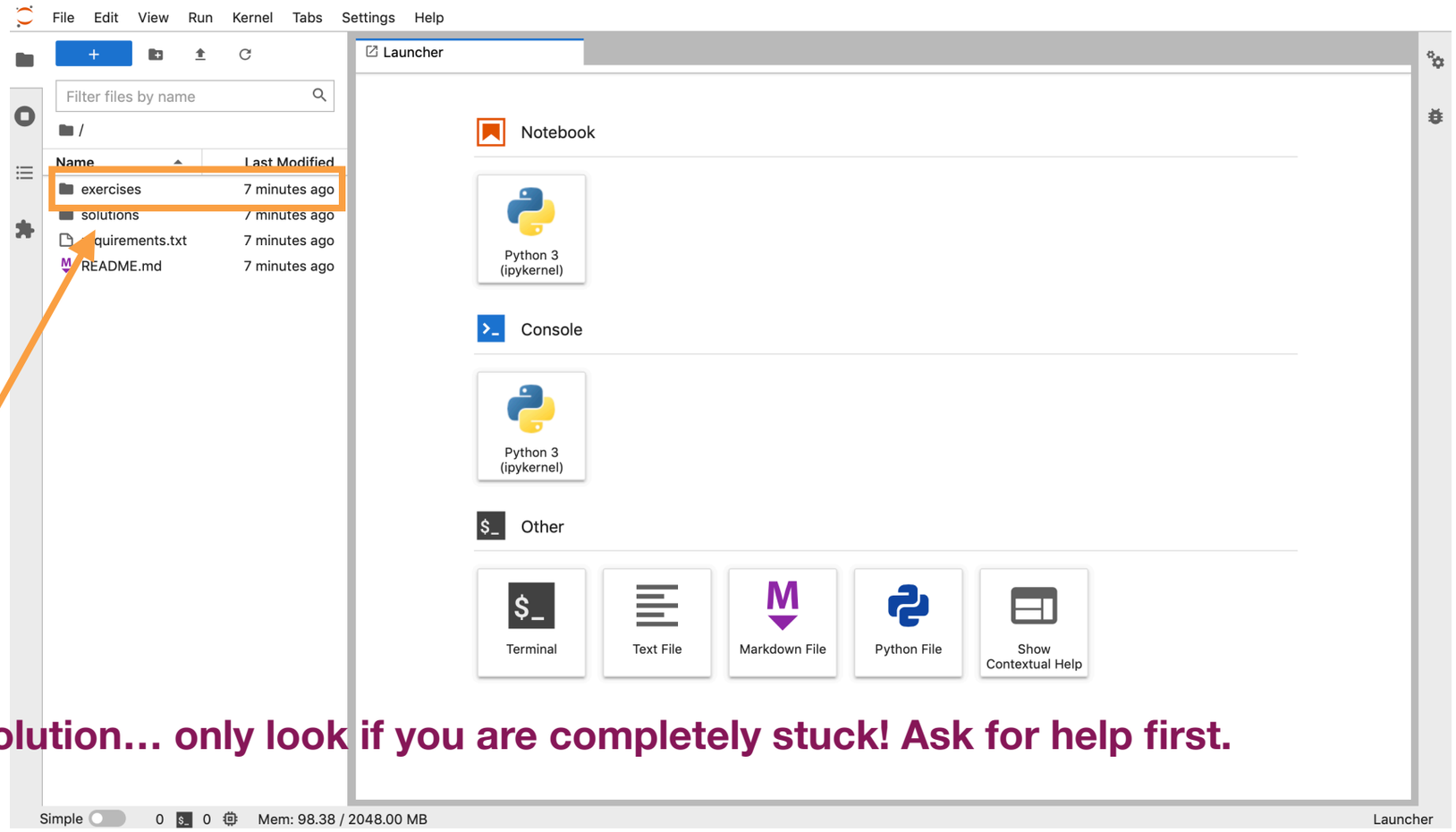
Binder

- You can run Binder directly from my GitHub
- Visit my GitHub page: https://github.com/lhwhitehead/TutorialDL
- Click on the launch binder button at the bottom
- You should see a screen like this, and a terminal below. Just wait…
- After a few minutes you'll have the server running



Starting Binder

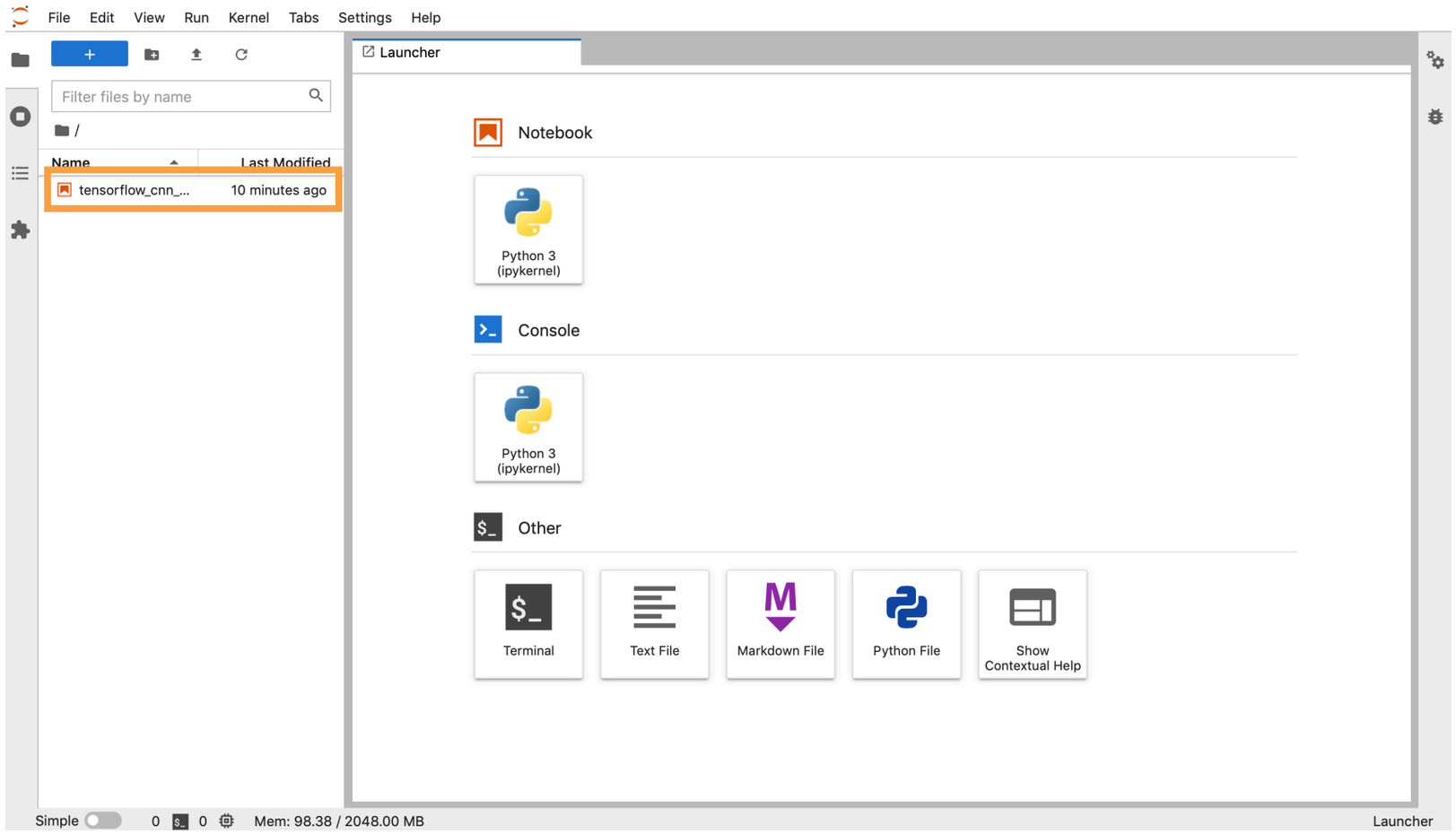
- You should see something like this
  - Navigate into exercises on the left



NB: this is the solution… only look if you are completely stuck! Ask for help first.

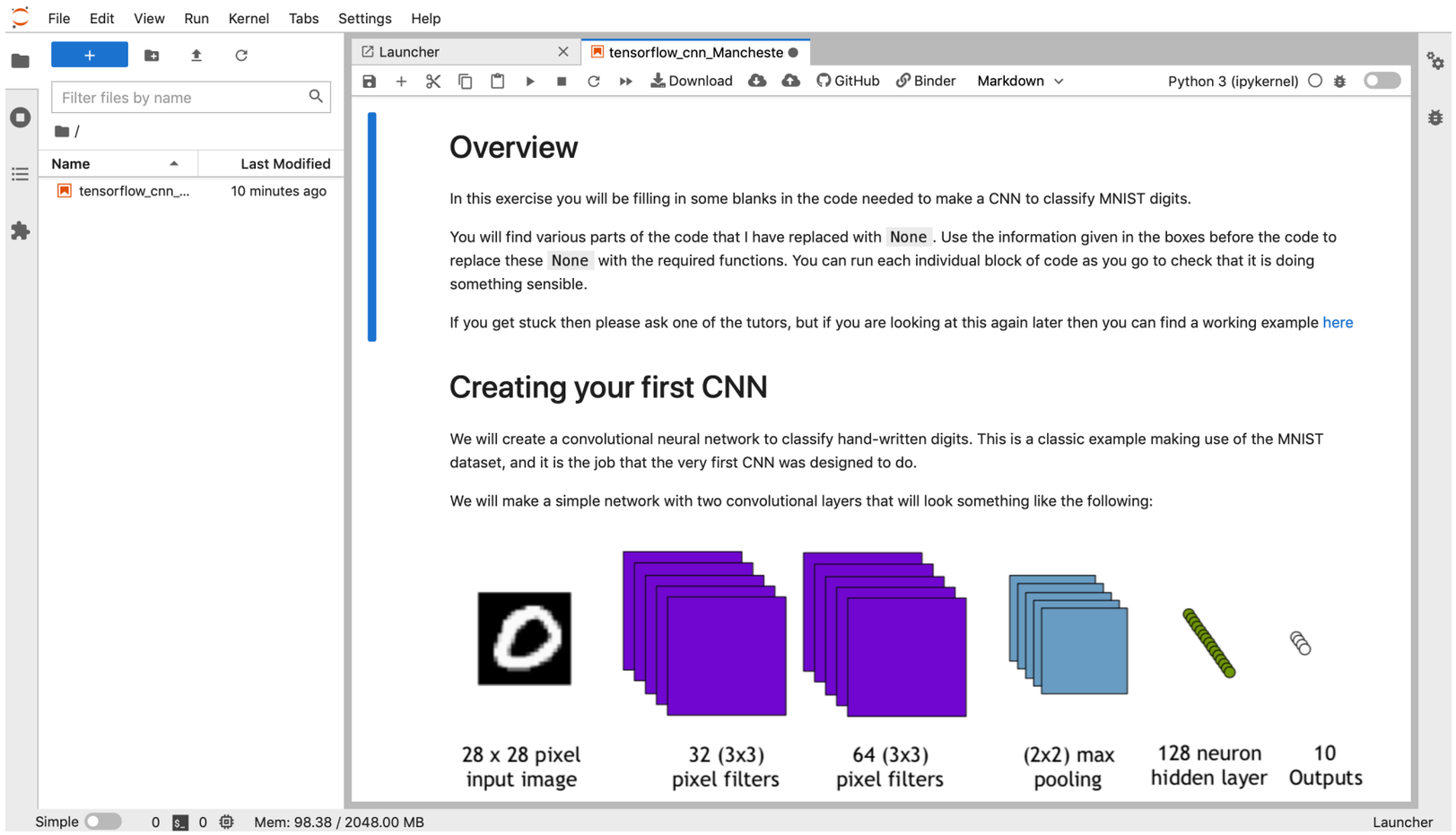
Starting Binder

- You should see something like this
  - Navigate into exercises on the left
  - Select the only file in there



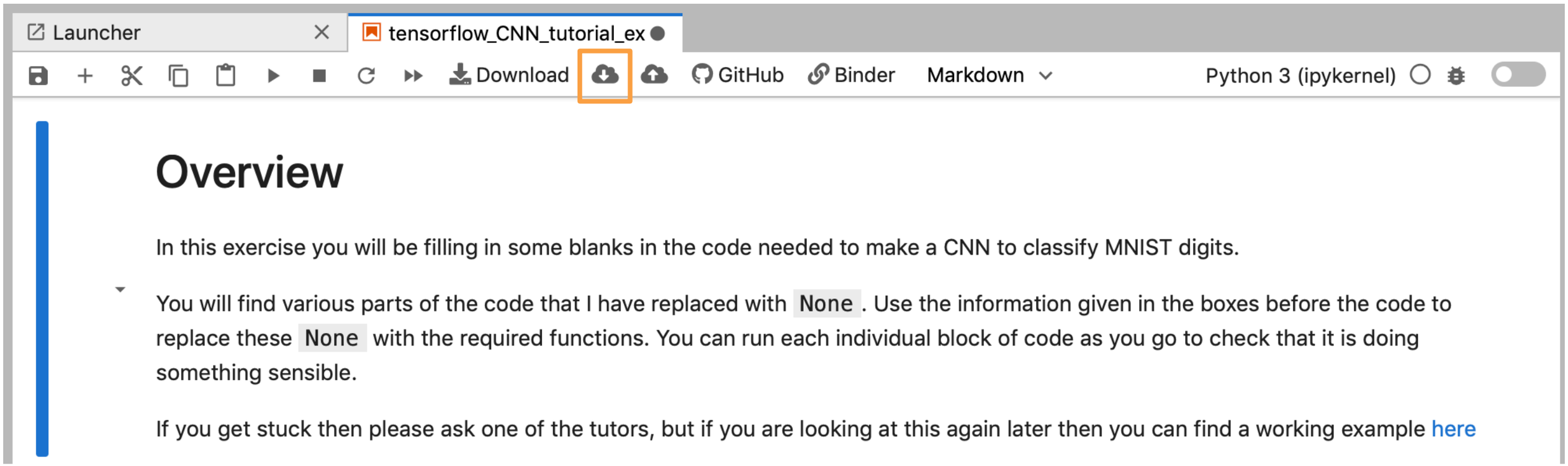
Starting Binder

- You should see something like this
  - Navigate into exercises on the left
  - Select the only file in there
  - This will load the actual notebook



Reconnecting Binder

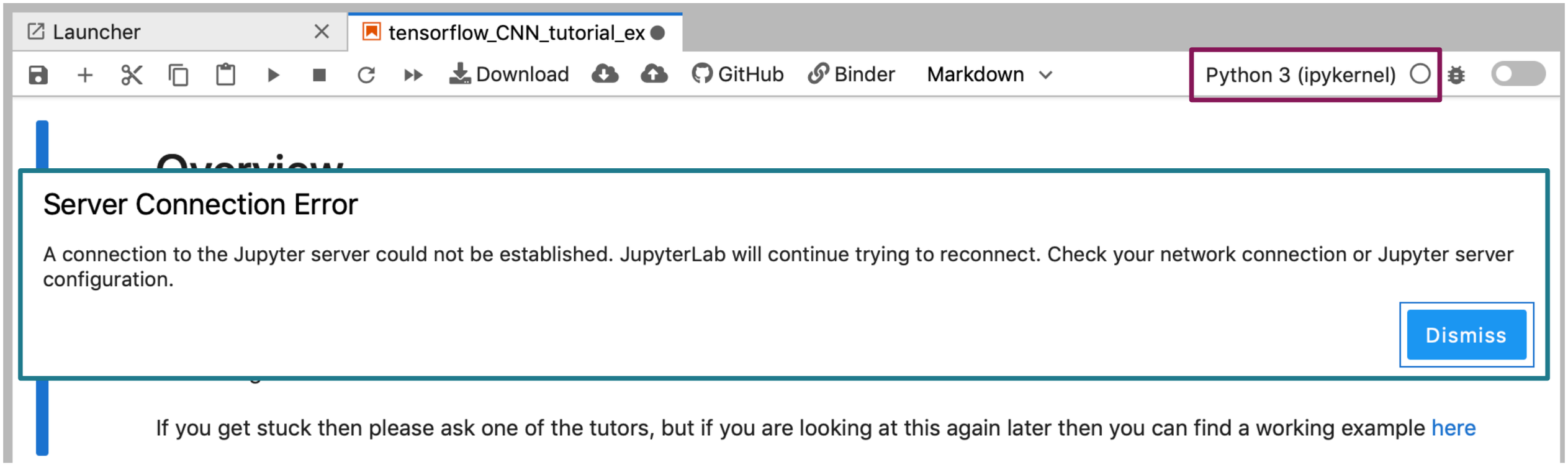
- Binder has the unfortunate “feature” that sessions disconnect quickly after periods with no activity
  - Please try to regularly save your work to the browser storage using this button



- If you see the disconnect message, first try reconnecting to the kernel

Reconnecting Binder

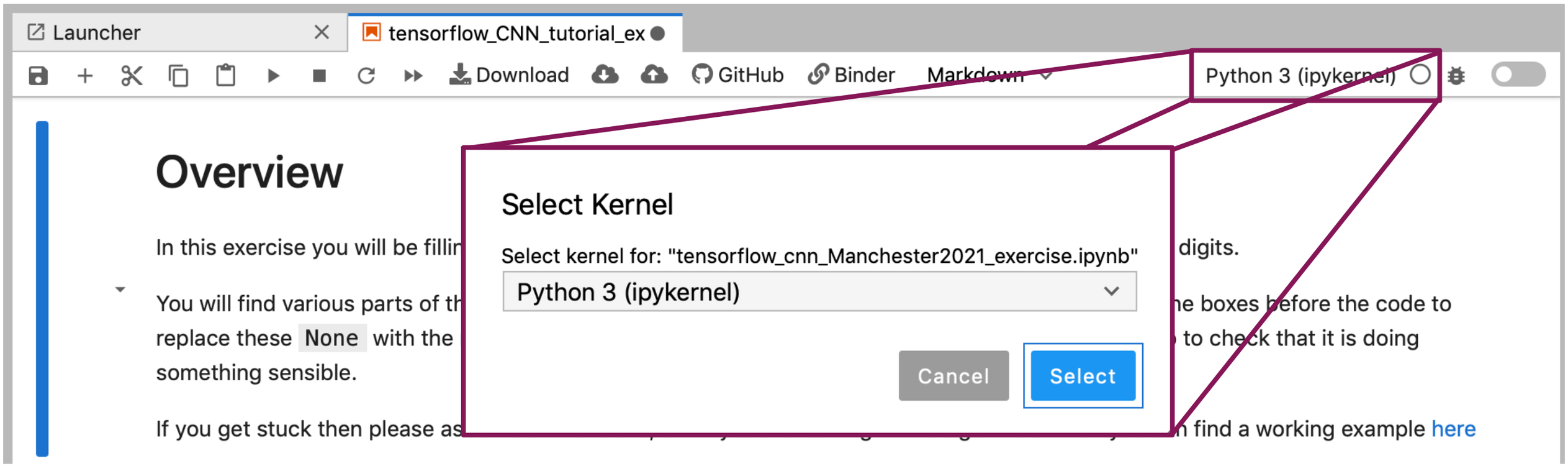
- Binder has the unfortunate “feature” that sessions disconnect quickly after periods with no activity
  - Please try to regularly save your work to the browser storage using this button



- If you see the disconnect message, first try reconnecting to the kernel

Reconnecting Binder

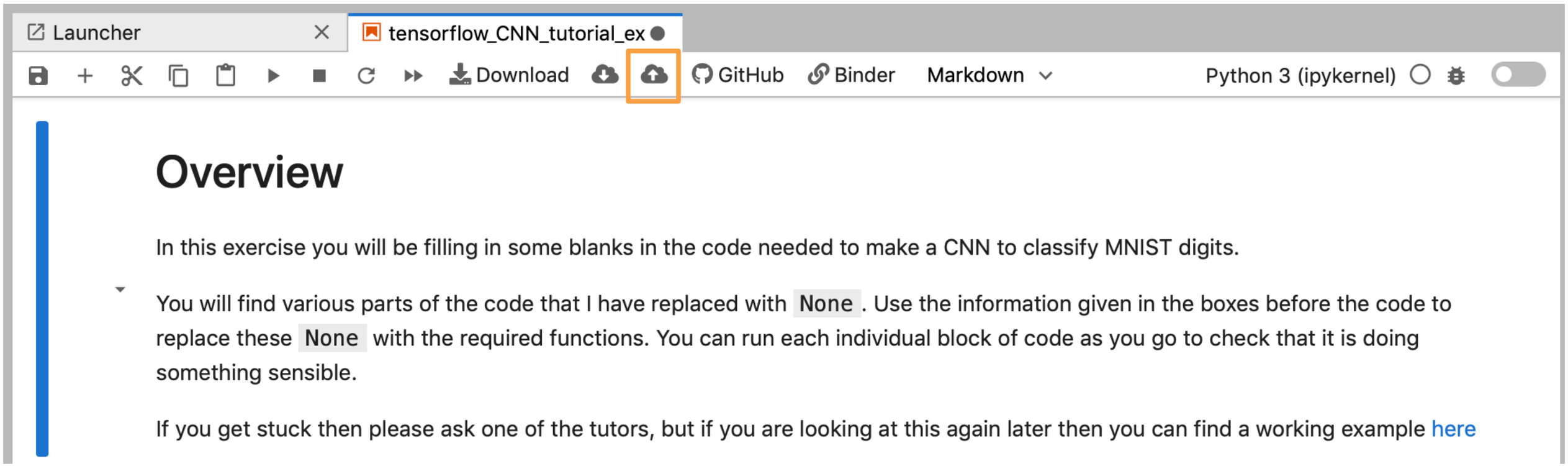
- Binder has the unfortunate “feature” that sessions disconnect quickly after periods with no activity
  - Please try to regularly save your work to the browser storage using this button



- If you see the disconnect message, first try reconnecting to the kernel

Reconnecting Binder

- Binder has the unfortunate “feature” that sessions disconnect quickly after periods with no activity
  - Please try to regularly save your work to the browser storage using this button



- If you see the disconnect message, first try reconnecting to the kernel
  - If this doesn't work, I think you need to start again from the GitHub page and use the restore from browser storage button

The Aim

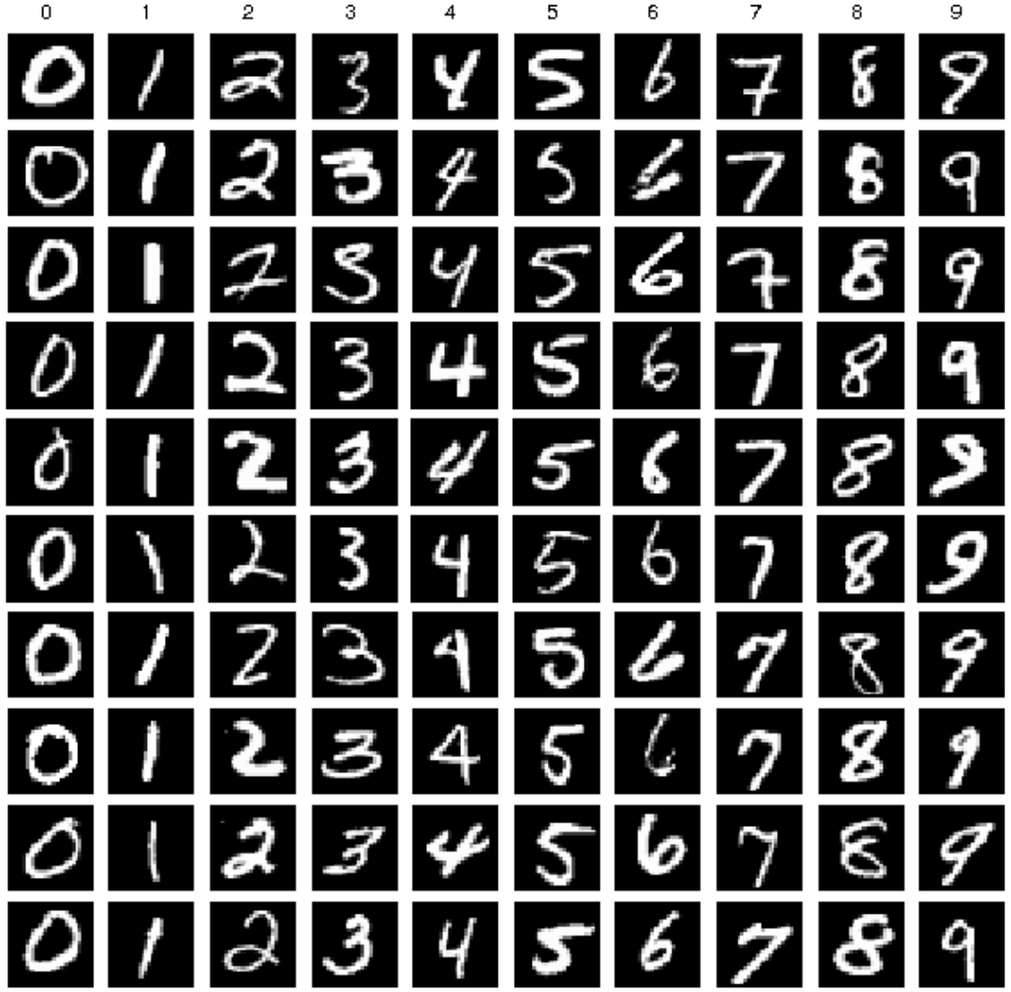
- We don’t have time to use a large neutrino dataset to classify neutrinos
  - We will use a simple convolutional neural network to classify the MNIST benchmark data set

- 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

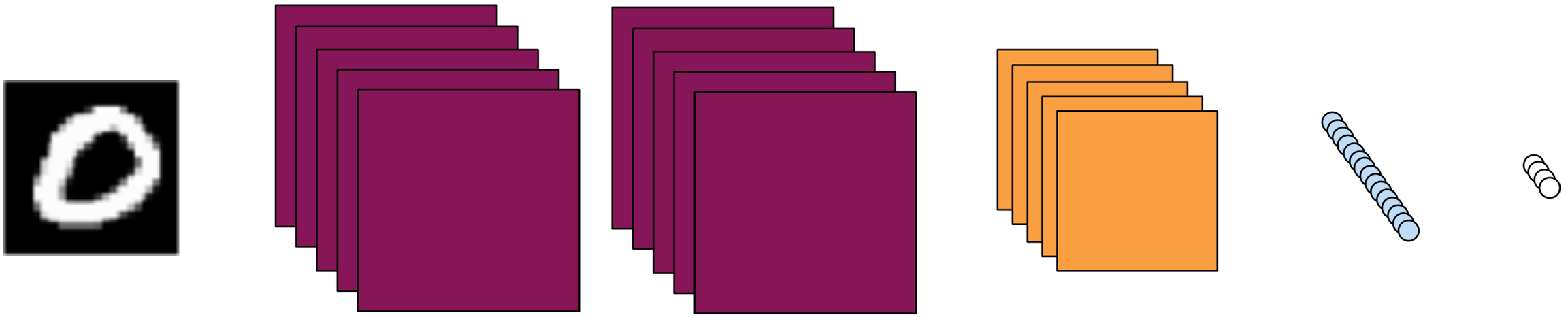
- Was a benchmark dataset for CNNs for a number of years



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

- The network we will build looks something like this:



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

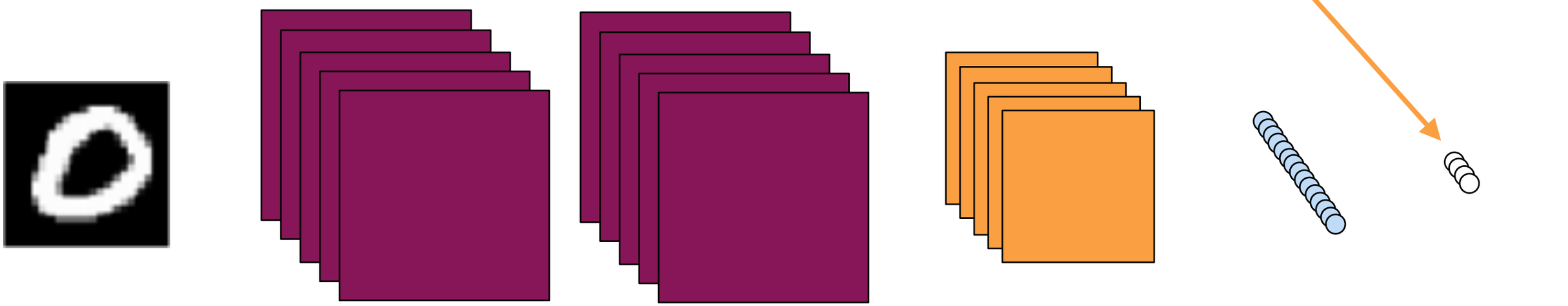
(2x2) max pooling

128 neuron hidden layer

10 Outputs

Our Network

- The network we will build looks something like this:



The exercise

- Ok, now we can play with something!
  - The following will be based on this nice tutorial

- 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
  - The first block of code takes care of this
  - Run it by selecting the box it is in and pressing shift + enter

[2]:
```
import tensorflow
from tensorflow import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
import numpy as np
import matplotlib.pyplot as plot

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

```
Tensorflow version: 2.8.2
```

  - 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

- You can think of these `import` statements as the python version of the C++ `#include` statement

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

[3]:
```
# The batch size controls the number of images that are processed simultaneously
# It helps with computational efficiency
batch_size = 128
# Since we are classifying hand-written digits, we want to classify each image
# as one of ten values: 0, 1, 2, ... , 8, 9
num_classes = 10
# The number of epochs (iterations over the entire training set) that we want
# to train the network for
epochs = 1
# The learning rate is a very important hyperparameter, as discussed in the
# lecture. This is a fairly common default value to try
learning_rate = 0.001
```

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

Loading the MNIST dataset

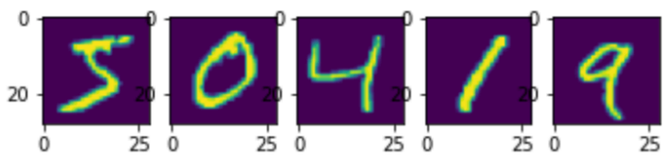
- We are now ready to load the MNIST dataset
  - Automatically downloads when requested from keras
  - We can print a few images just to see how they look
  - Note that the dataset has been split into train and test samples for us

```
[4]: # Since MNIST is a standard dataset, we can just get it straight from keras.
# It is also 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
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Let's store the shape of the images for convenience
print("Shape of input array =",x_train.shape)
training_size = x_train.shape[0]
testing_size = x_test.shape[0]
img_rows = x_train.shape[1]
img_cols = x_train.shape[2]
print('Input images have shape',img_rows,'x',img_cols)
print('There are',training_size,'images for training and',testing_size,'images for testing')

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
Shape of input array = (60000, 28, 28)
Input images have shape 28 x 28
There are 60000 images for training and 10000 images for testing
Example images with true classes [5 0 4 1 9]
```



Preparing the data

- We need to slightly rearrange the data shapes for the CNN
  - Here are the first two lines of code for you to fill in
  - Click on the block to start editing
  - Read the hints above the code block

`keras.utils.to_categorical(y,number_of_classes)`

```
# We need to make sure the numpy arrays are in the correct format for the CNN
# These are 4D tensors where the first number is the number of images, the
# following two arguments are the image size, and the final one is the image
# depth, which for greyscale is 1, and if these were rgb images, it would be 3
x_train = x_train.reshape(training_size,img_rows,img_cols,1)
x_test = x_test.reshape(testing_size,img_rows,img_cols,1)

# 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 = None
y_test = None

# Let's just check the shapes
print('x_train shape =',x_train.shape)
print('y_train shape =',y_train.shape)
```

```
x_train shape = (60000, 28, 28, 1)
y_train shape = (60000, 10)
```

← Expected output

Building the CNN

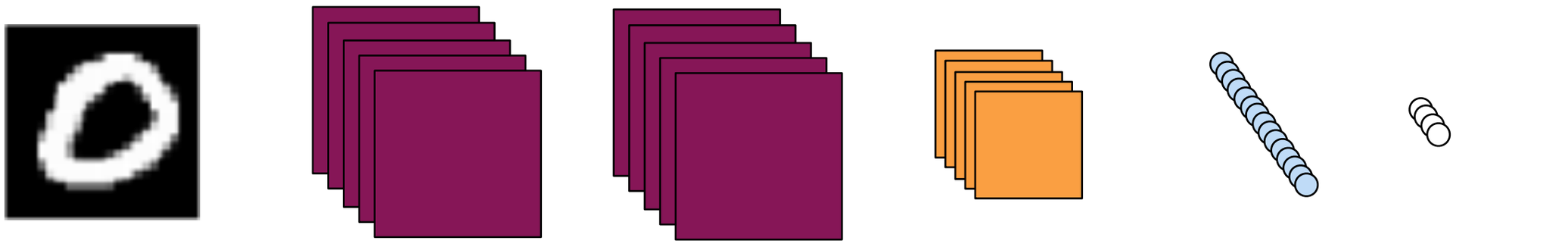
- This large block of code is used to build our CNN
  - There are lots of blanks to fill in here!
  - Don’t worry, we will discuss these missing lines

- More details on the functions are given in the python notebook

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

(2x2) max pooling

128 neuron hidden layer

10 Outputs

- Define the first convolutional layer using **Conv2D(…)**
- Define input size to match the input image

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

Building the CNN

- Let's see this function a little more clearly
  - We add a 2D convolutional layer
  - We want 32 filters
  - The kernel size (or filter size) is 3 x 3
  - We will use the ReLU activation function
  - This is the first layer: need to tell the Sequential model what to expect

    https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D

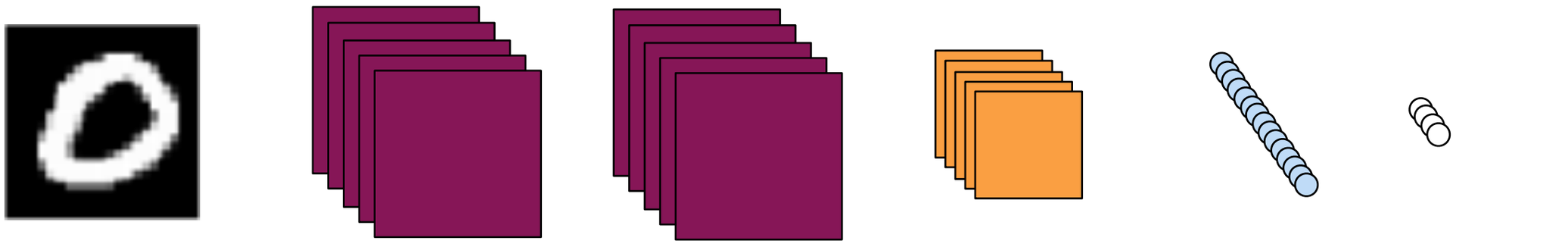
```
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
```

- Define the first convolutional layer using **Conv2D(…)**
- Define input size to match the input image

```
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image | 32 (3x3) pixel filters | 64 (3x3) pixel filters | (2x2) max pooling | 128 neuron hidden layer | 10 Outputs

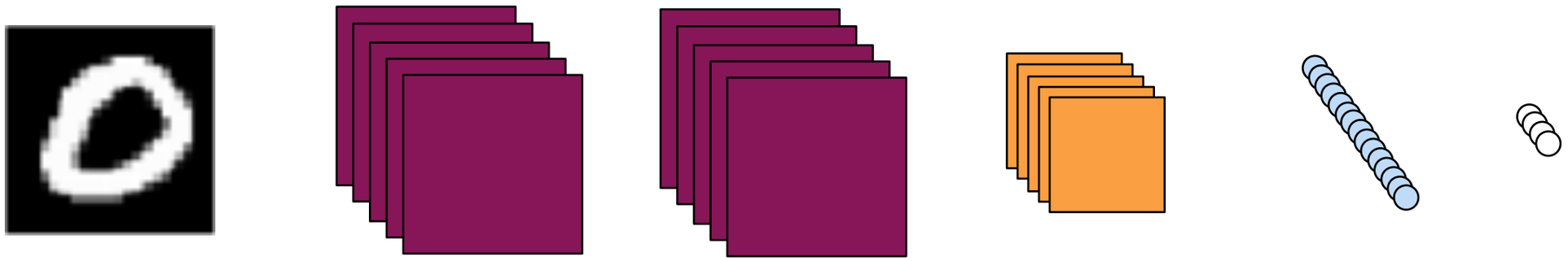
- Now for the second convolution

- The Sequential model keeps track of the data shape between layers
  - Omit the input_shape for all layers now

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image | 32 (3x3) pixel filters | 64 (3x3) pixel filters | (2x2) max pooling | 128 neuron hidden layer | 10 Outputs

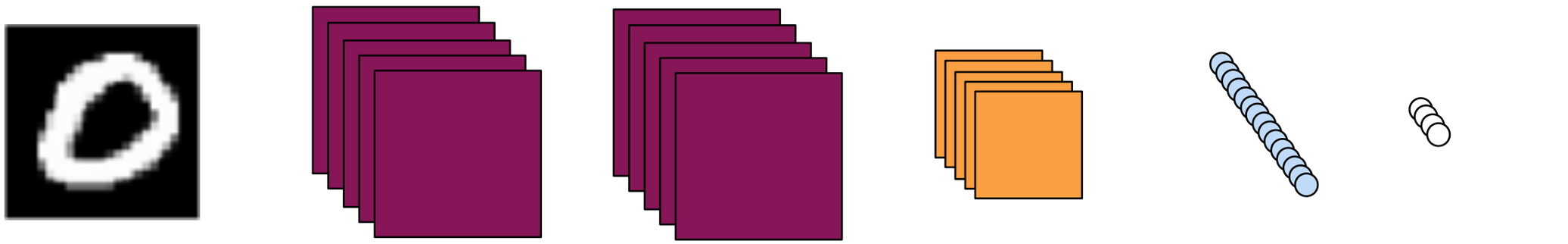
```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

- Now for the max pooling layer
  - Merges 2x2 pixels and assigns its value as the maximum of the four pixels

- Use **MaxPooling2D(…)**
  - See hints in the exercise

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

(2x2) max pooling

128 neuron hidden layer

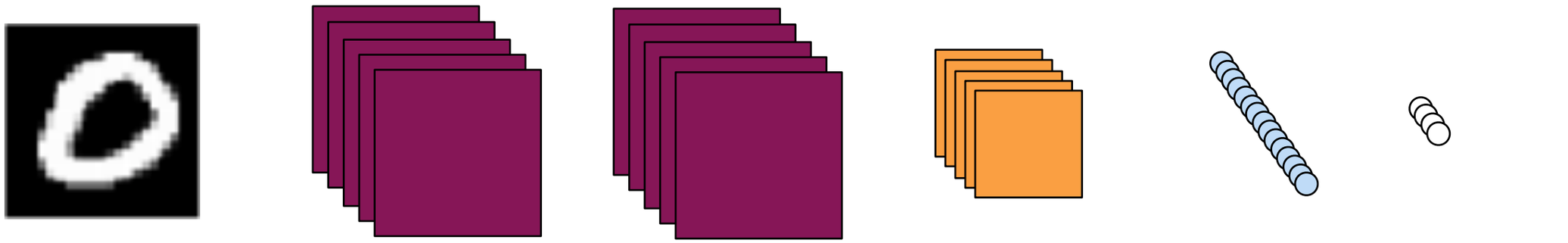
10 Outputs

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

- Now to add a Dropout layer (not shown on the diagram)
  - Use **Dropout(…)**
  - We want to switch off 25% of neurons

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

(2x2) max pooling

128 neuron hidden layer

10 Outputs

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

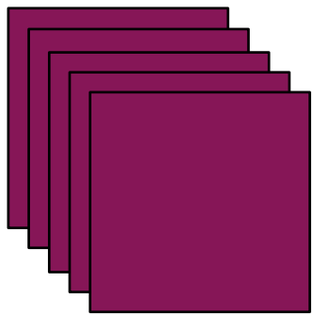
- We need to go from 2D data down to 1D for the dense layers (not shown in the diagram)

- Use **Flatten()**

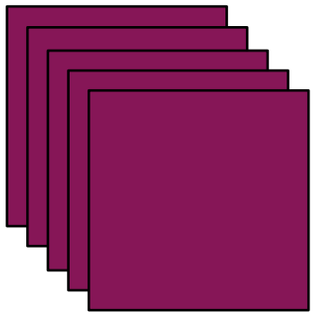
Building the CNN

- Lets remember our network architecture…

28 x 28 pixel input image

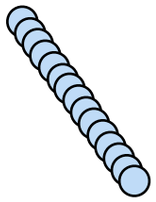


32 (3x3) pixel filters



64 (3x3) pixel filters

(2x2) max pooling



128 neuron hidden layer

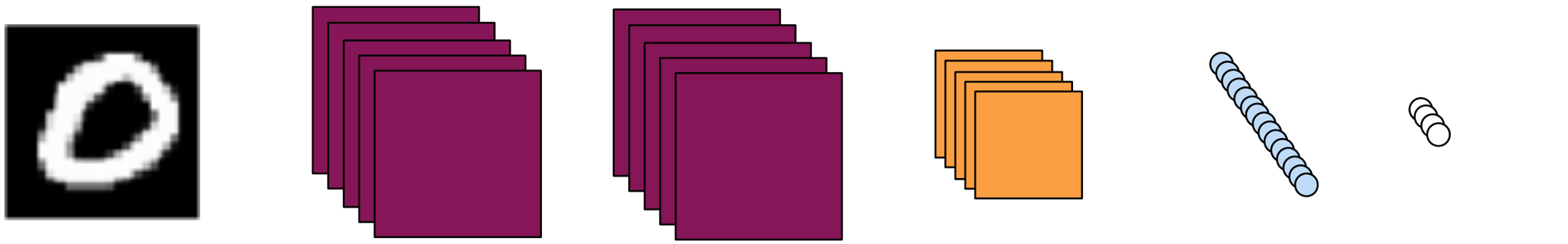
10 Outputs

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

- This hidden layer is a dense layer with 128 neurons
- Use **Dense(…)**
- Use the ReLU activation

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

(2x2) max pooling

128 neuron hidden layer

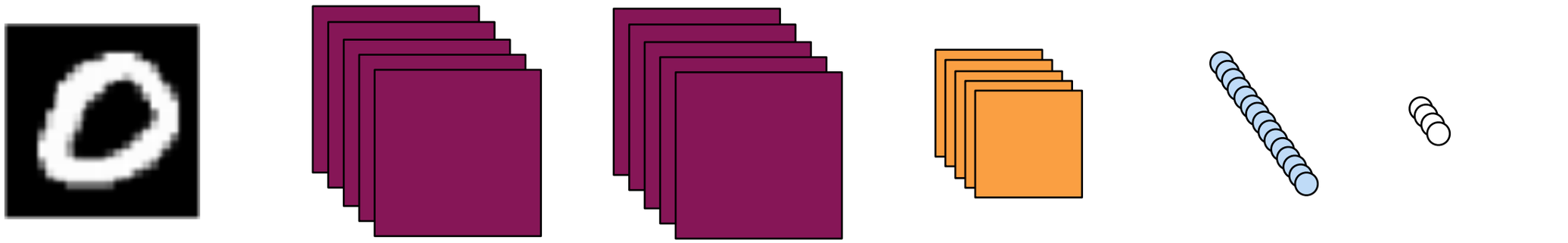
10 Outputs

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

- Add a second Dropout layer (not shown on the diagram)
  - Use **Dropout(…)**
  - This time we want to switch off 50% of neurons

Building the CNN

- Lets remember our network architecture…



28 x 28 pixel input image

32 (3x3) pixel filters

64 (3x3) pixel filters

(2x2) max pooling

128 neuron hidden layer

10 Outputs

```
# Now we get to define our neural network
model = Sequential()
# The first convolutional layer needs to know what size images it will operate
# on, as given by the input_shape variable. Here we apply a vector of 32
# filters of size (3,3) pixels. The 'relu' function provides some non-
# linearity, feel free to read up on other activation functions
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows,img_cols,1)))
# Now we add a second convolutional layer, this time with 64 (3,3) filters
# Use the relu activation function as above. We don't need to provide the
# input shape for any of the following layers as the sequential model
# knows to pass the output of the previous layer to the current one
model.add(None)
# Pooling layers downsample the images - in this case 2x2 pixels become
# one pixel. Specifically, we use a MaxPooling2D layer
model.add(None)
# Dropout disables some of the neurons to prevent overtraining.
# We will use a dropout fraction of 0.25.
model.add(None)
# We flatten the images into a single vector to pass into the dense
# layers
model.add(None)
# The dense layer is what you have seen from a standard neural network
# We will use a dense layer with 128 nodes and the relu activation
model.add(None)
# More dropout to avoid overtraining, this time with a fraction of 0.5
model.add(None)
# The final layer is a dense layer containing (num_classes) nodes.
# Using a softmax activation ensures that the sum of these 10 outputs is 1.
model.add(None)
# Let's have a look at our model to check it has come together as we expect
model.summary()
```

- Add the final output layer
  - This is another **Dense(…)** layer, but with 10 neurons

- Must use the softmax activation so that the 10 output scores sum to one

Building the CNN

- We now have our CNN!
  - The last line prints out a summary of the model
  - You should see this output if all is correct

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1179776
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

- I will pause here until most of you have successfully built your CNNs

Training your CNN

- Now that you've got your CNN, you want to train it!
  - Firstly we need to tell the model how it should train
  - Which loss function? Which optimiser?

```
# Now we build the model, defining the loss function,
# optimiser (I typically use Adam)
# Learning rate(learning_rate) is a parameter that we can optimise to help
# convergence - remember that we defined this near the start of the script.
# We use categorical cross entropy loss, which is the one
# to use for a classification task with more than two classes.
loss_function = None
optimiser = None
# We compile the model and tell it which loss function and optimiser to use
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
  - Follow the clues in the exercise for these

- 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

```
# Now it is time to actually train the model using the training data with the
# true target outputs from MNIST. Use model.fit(...) here. In this case we will
# the test sample for validation
None
```

  - You will need to replace None with **model.fit(…)**
  - There are quite a lot of arguments to include

```
fit(x=<train_images>, y=<train_labels>, batch_size=<batch_size>, epochs=<epochs> verbose=<verbose_level>,
validation_data=(<validation_images,validation_labels>))
```

In this case, we use our test sample for validation

  - Just 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
  - This will take a few minutes
    - ~ 2.5 minutes on GoogleColab
    - ~ 5 minutes on Binder

A quick test

- Now that you have your network, I want to demonstrate a couple of uses
  - The first uses truth information to validate the performance
  - This is exactly what the code does with the validation data

```
[ ]  # Run the network on the test sample and see how we do.
     # This should match the final validation print out
     # from the training as we are using the same test sample.
     # We use the model.evaluate function that makes use of
     # truth labels to gauge the performance to get the score variable
     score = None
     print('Test loss:', score[0])
     print('Test accuracy:', score[1])
```

- Run using model.evaluate

`evaluate(x=<test_images>, y=<test_labels>, verbose=<verbose_level>)`

  - I use `verbose=0` here, but feel free to set it to `1`

- Your loss and accuracy values should match the final validation loss and accuracy from the training print outs

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
- 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

```
# Make a list of incorrect classifications
incorrect_indices = []
# By default, let's check one thousand images from x_test.
# You can check more, up to the value of testing_size.
n_images_to_check = 1000
# Use the CNN to predict the classification of the images. It returns an array
# containing the 10 class scores for each image. Remember to use the x[:n]
# notation mentioned above in the following function call
raw_predictions = model.predict(x=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))
```

The number of incorrectly classified images depends on the training. As a guide, after one epoch, I had 21 / 1000 incorrect classifications.

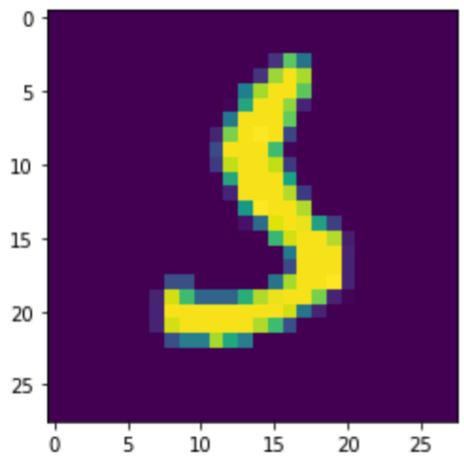
Checking the incorrect images

- Now, for fun, let's have a look at the incorrect images

```
# Now you can modify this part to draw different images from the failures list
# The reshape just removes the depth dimension for drawing
im = 0
image_to_plot = x_test[incorrect_indices[im][0]].reshape(28,28)
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.reshape((28,28)))
```

- You'll see an image alongside some information

```
Incorrect classification for image 340 : predicted = 3 with true = 5
<matplotlib.image.AxesImage at 0x7f9a31405f90>
```



This number 5 was classified as a 3. It isn't the best number 5 that I've ever seen!

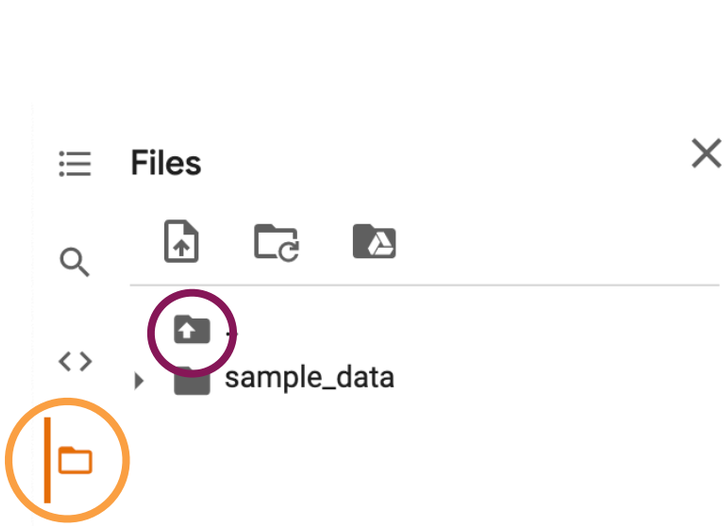
- Just change the value of `im` to see different images

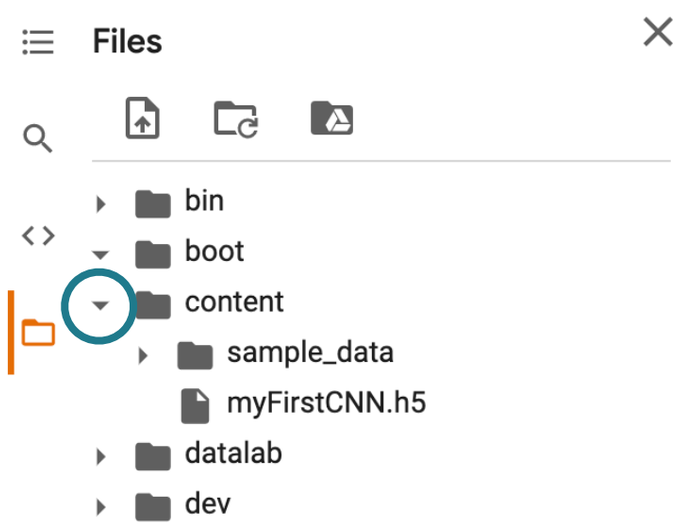
Saving your model

- To use our network in a realistic way we need to save it
  - To do so, we simply use the **model.save(…)** function
    - Just choose some name with a .h5 file extension `model.save(<file_name>)`

```
# Save the model as a .h5 file
None
```

  - Do a quick check to see that the file was created
    - (Google Colab) Click on the folder icon on the left side to open the file browser, navigate up one level, then click the little down arrow next to content





Loading your model

- It is just as simple to load a model

- We make use of a keras function here: `keras.models.load_model(<file_name>)`
  - Load the model and then print the summary to check it

```
# Load the model and then print the summary to check it
# looks how we expect
loaded_model = None
loaded_model.summary()
```

- You should find that you get the same network as before!

- Now we can do a little test to make sure it works too

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               1179776
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Testing your model

- Let's just run over a few images from the testing data
  - We'll do it one-by-one this time just to show a trick that you might need at some point
    - Processing the images together would be more efficient
    - I haven't left blanks here, but you can compare it to the earlier code where we looked for incorrect classifications

```
for i in range(0,10):
  # Note that here I have to use reshape to prevent losing the first
  # dimension of the array when using x_test[i]. Otherwise
  # test_image would only have three dimensions (img_rows,img_cols,1)
  # and would be incompatible with the CNN input layer
  test_image = x_test[i].reshape(1,img_rows,img_cols,1)
  print('Predicted class =',np.argmax(loaded_model.predict(test_image)),
        "True class =",np.argmax(y_test[i]))
```

```
Predicted class = 7 True class = 7
Predicted class = 2 True class = 2
Predicted class = 1 True class = 1
Predicted class = 0 True class = 0
Predicted class = 4 True class = 4
Predicted class = 1 True class = 1
Predicted class = 4 True class = 4
Predicted class = 9 True class = 9
Predicted class = 5 True class = 5
Predicted class = 9 True class = 9
```

Play around a bit!

- At this stage you can just explore changing things and see what difference it makes

- I advise starting with the number of epochs
  - Try increasing the epochs to 2 and then 5
  - Should see the loss decreasing and accuracy increasing with each epoch

- Change the learning rate
  - Raise and lower by a factor of 10

- On the next slide I'll show what happens when I do this but will wait for you to have a go before discussing it

Play around a bit!

- Running for 5 epochs:

```
Epoch 1/5
469/469 [==============================] - 144s 306ms/step - loss: 0.8255 - accuracy: 0.8635 - val_loss: 0.0675 - val_accuracy: 0.9787
Epoch 2/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1556 - accuracy: 0.9549 - val_loss: 0.0537 - val_accuracy: 0.9825
Epoch 3/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1118 - accuracy: 0.9674 - val_loss: 0.0513 - val_accuracy: 0.9848
Epoch 4/5
469/469 [==============================] - 142s 304ms/step - loss: 0.0950 - accuracy: 0.9726 - val_loss: 0.0414 - val_accuracy: 0.9859
Epoch 5/5
469/469 [==============================] - 143s 305ms/step - loss: 0.0820 - accuracy: 0.9752 - val_loss: 0.0432 - val_accuracy: 0.9857
```

  - Results improve as we train for longer (of course!)

- Increase the learning rate to 0.01

```
Epoch 1/5
469/469 [==============================] - 144s 307ms/step - loss: 2.4211 - accuracy: 0.7741 - val_loss: 0.2411 - val_accuracy: 0.9268
Epoch 2/5
469/469 [==============================] - 143s 306ms/step - loss: 0.4542 - accuracy: 0.8644 - val_loss: 0.2787 - val_accuracy: 0.9268
Epoch 3/5
469/469 [==============================] - 143s 306ms/step - loss: 0.4001 - accuracy: 0.8793 - val_loss: 0.1894 - val_accuracy: 0.9407
Epoch 4/5
469/469 [==============================] - 143s 305ms/step - loss: 0.3959 - accuracy: 0.8814 - val_loss: 0.1922 - val_accuracy: 0.9433
Epoch 5/5
469/469 [==============================] - 142s 303ms/step - loss: 0.3898 - accuracy: 0.8813 - val_loss: 0.1923 - val_accuracy: 0.9464
```

  - Accuracy reaches a maximum of 88%
  - The optimiser hasn’t been able to find the correct minimum

Play around a bit!

- Running for 5 epochs:

```
Epoch 1/5
469/469 [==============================] - 144s 306ms/step - loss: 0.8255 - accuracy: 0.8635 - val_loss: 0.0675 - val_accuracy: 0.9787
Epoch 2/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1556 - accuracy: 0.9549 - val_loss: 0.0537 - val_accuracy: 0.9825
Epoch 3/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1118 - accuracy: 0.9674 - val_loss: 0.0513 - val_accuracy: 0.9848
Epoch 4/5
469/469 [==============================] - 142s 304ms/step - loss: 0.0950 - accuracy: 0.9726 - val_loss: 0.0414 - val_accuracy: 0.9859
Epoch 5/5
469/469 [==============================] - 143s 305ms/step - loss: 0.0820 - accuracy: 0.9752 - val_loss: 0.0432 - val_accuracy: 0.9857
```

- Decrease the learning rate to 0.0001

```
Epoch 1/5
469/469 [==============================] - 149s 315ms/step - loss: 1.2330 - accuracy: 0.7115 - val_loss: 0.1695 - val_accuracy: 0.9517
Epoch 2/5
469/469 [==============================] - 147s 314ms/step - loss: 0.3261 - accuracy: 0.9060 - val_loss: 0.0937 - val_accuracy: 0.9727
Epoch 3/5
469/469 [==============================] - 147s 314ms/step - loss: 0.2022 - accuracy: 0.9397 - val_loss: 0.0694 - val_accuracy: 0.9790
Epoch 4/5
469/469 [==============================] - 147s 313ms/step - loss: 0.1509 - accuracy: 0.9550 - val_loss: 0.0542 - val_accuracy: 0.9832
Epoch 5/5
469/469 [==============================] - 147s 313ms/step - loss: 0.1177 - accuracy: 0.9646 - val_loss: 0.0515 - val_accuracy: 0.9841
```

  - Accuracy improves but more slowly that with the default value
  - Would need to run for more epochs to see if was in the global minimum

Play around a bit!

- Running for 5 epochs:

```
Epoch 1/5
469/469 [==============================] - 144s 306ms/step - loss: 0.8255 - accuracy: 0.8635 - val_loss: 0.0675 - val_accuracy: 0.9787
Epoch 2/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1556 - accuracy: 0.9549 - val_loss: 0.0537 - val_accuracy: 0.9825
Epoch 3/5
469/469 [==============================] - 143s 305ms/step - loss: 0.1118 - accuracy: 0.9674 - val_loss: 0.0513 - val_accuracy: 0.9848
Epoch 4/5
469/469 [==============================] - 142s 304ms/step - loss: 0.0950 - accuracy: 0.9726 - val_loss: 0.0414 - val_accuracy: 0.9859
Epoch 5/5
469/469 [==============================] - 143s 305ms/step - loss: 0.0820 - accuracy: 0.9752 - val_loss: 0.0432 - val_accuracy: 0.9857
```

- Add another convolutional layer (in this case, just before Flatten())

```
Epoch 1/5
469/469 [==============================] - 164s 349ms/step - loss: 0.4977 - accuracy: 0.8720 - val_loss: 0.0628 - val_accuracy: 0.9818
Epoch 2/5
469/469 [==============================] - 163s 348ms/step - loss: 0.1161 - accuracy: 0.9670 - val_loss: 0.0466 - val_accuracy: 0.9857
Epoch 3/5
469/469 [==============================] - 163s 348ms/step - loss: 0.0863 - accuracy: 0.9758 - val_loss: 0.0431 - val_accuracy: 0.9864
Epoch 4/5
469/469 [==============================] - 165s 351ms/step - loss: 0.0683 - accuracy: 0.9800 - val_loss: 0.0376 - val_accuracy: 0.9886
Epoch 5/5
469/469 [==============================] - 165s 351ms/step - loss: 0.0598 - accuracy: 0.9823 - val_loss: 0.0358 - val_accuracy: 0.9882
```

  - The model is now more complex and has more parameters
  - As expected, the accuracy improves!

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 (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

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!