Question (i)

Part (a)

To implement convolution, while moving the kernel through the *nxn* input array from left to right and top to bottom, you have to calculate the weighted sum of said kernel with a subset of the input array. Put simply, we multiply each point in the kernel with it's corresponding point in the input array (getting this with indexes [$iterationOneFromKernel+iterationOneFromMatrixSize$] and [$iterationTwoFromKernel+iterationTwoFromMatrixSize$]). This is done for all kernel points. The function below from section 1.1 in the appendix shows how the implementation of convolving the kernel to the input array looks like:

```
def part_a(NxN_Array, KxK_Kernel):
    returningMatrixSize = (NxN_Array.shape[0] - KxK_Kernel.shape[0]) + 1
    returningMatrix = np.empty([returningMatrixSize, returningMatrixSize])
    for iterationOneFromMatrixSize in range(returningMatrixSize):
        for iterationTwoFromMatrixSize in range(returningMatrixSize):
            summation = 0
            for iterationOneFromKernel in range(KxK_Kernel.shape[0]):
                for iterationTwoFromKernel in range(KxK_Kernel.shape[0]):
                    summation = summation+NxN_Array[iterationOneFromKernel
                                + iterationOneFromMatrixSize]
                                [iterationTwoFromKernel +
                                iterationTwoFromMatrixSize] *
                                KxK_Kernel[iterationOneFromKernel]
                                          [iterationTwoFromKernel]
            returningMatrix[iterationOneFromMatrixSize]
                           [iterationTwoFromMatrixSize] = summation
    return returningMatrix
```

Part (b)

The following kernels:

$$kernel1 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad kernel2 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

can be used as input to the convolution function made in part (a) (i). We can then choose an image (200 pixels x 200 pixels), extract 3 RGB arrays from it and choose one to work with (such as the color red), and use the convolution function with this to get an output using only kernel1, only kernel2 and both kernels to see what they look like. The code below is from section 1.2 in the appendix and it demonstrates this:

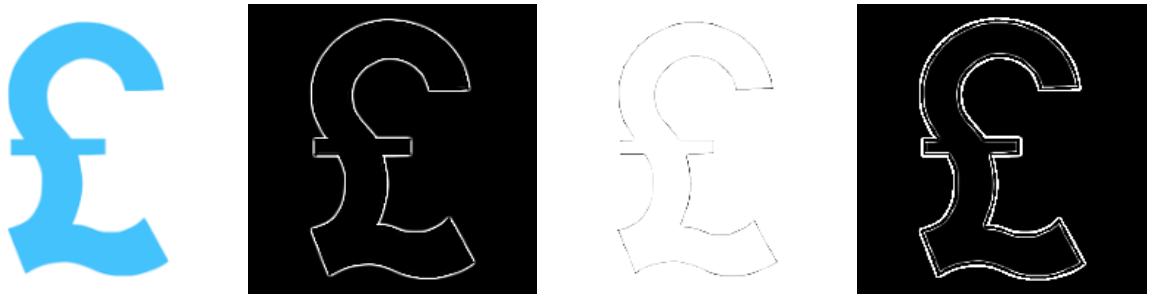
```
def part_b():
    # Code given in question:
    im = Image.open('pentagon.jpg')
    rgb = np.array(im.convert('RGB'))
    r = rgb[:, :, 0]  # array of R pixels
    img_array = np.uint8(r)

    # Using only kernel #1
    firstKernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    imageOneUsingFirstKernel = part_a(img_array, firstKernel)
    Image.fromarray(imageOneUsingFirstKernel).show()
```

```
# Using only kernel #2
secondKernel = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])
imageTwoUsingSecondKernel = part_a(img_array, secondKernel)
Image.fromarray(imageTwoUsingSecondKernel).show()

# Using both kernel #1 and kernel #2
firstAndSecondKernel = part_a(img_array, firstKernel)
firstAndSecondKernel = part_a(firstAndSecondKernel, secondKernel)
Image.fromarray(firstAndSecondKernel).show()
```

The inputted image as well as the output images are as follows:



(a) Image input. (b) *kernel*1 output. (c) *kernel*2 output. (d) *kernel*1&2 output.

Figure 1: Input and output images using *kernel*1 and *kernel*2.

The output images show that *kernel*1 and *kernel*2 are edge detectors as they outline the edges of the original shape. We can see that the output colours of both are inverted (black/white & white/black) however most importantly, we can see from the images that *kernel*1's output is slightly better than *kernel*2's output in terms of detecting the edge. This makes sense since *kernel*1 takes diagonals into account (−1's on the corners of the matrix) while *kernel*2 does not, so it is slightly worse.

Question (ii)

As I went through question (ii), I modified the provided code for this assignment by putting it into a function and adding parameters such as the number of training data points (**n**), if we want to compare to a baseline (**boolean compareWithBaseline**), the number of epochs (**integer epochs**), if we should record the time to train the network (**boolean recordTime**), the L1 values to use (**list L1_values**), if we should only show the accuracy plot (**boolean showOnlyAccuracyPlot**), if we should use max pooling (**boolean useMaxPooling**) and if we should use a thinner and deeper network (**boolean thinnerAndDeeper**). This is located in section 1.3 and called in 1.5 of the appendix.

Part (a)

```
model.add(Conv2D(16, (3, 3), padding='same',
input_shape=x_train.shape[1:], activation='relu')) #1
model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same',
activation='relu')) #2
model.add(Conv2D(32, (3, 3), padding='same', activation='relu')) #3
model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same',
activation='relu')) #4
model.add(Dropout(0.5)) #5
model.add(Flatten()) #6
model.add(Dense(num_classes,
activation='softmax',kernel_regularizer=regularizers.l1(0.0001))) #7
```

The architecture of the convolutional network from the code is as follows (with *#number* indicating the line I'm dicussing):

- (#1): We notice a 2D convolutional layer with 16 output filters, a kernel size of 3x3, padding to ensure the input and output are the same size, and using relu validation (i.e. a rectified linear activation function) to map negative outputs to 0. A default stride of 1x1 is implied (how much the window moves by).
- (#2): We notice the same 2D convolutional layer as above, but this time it uses a stride of 2x2.
- (#3 & #4): We notice the same two 2D convolutional layers as the 2 above, but this time they both use 32 output filters.
- (#5): There is a dropout step to minimize over-fitting by using a rate of 0.5 to randomly set inputs to 0.
- (#6 & #7): The data is flattened, and the "dense" function uses the activation function of softmax for the output, using 10 classes with L1 regularization.

To visualize the output channels:

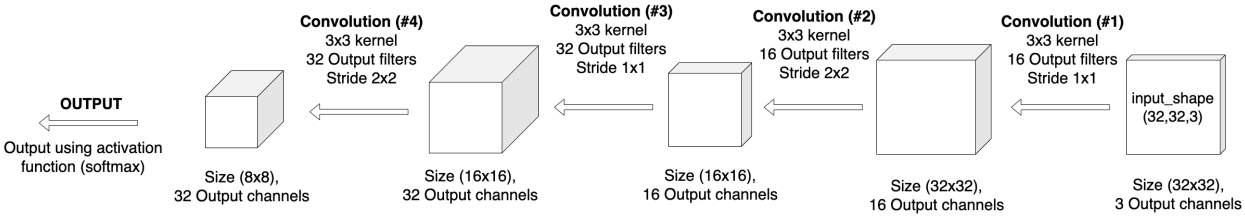


Figure 2: Convolutional Network architecture

Part (b) (i)

Using keras, it tells us that the model has a total of **37,146** parameters. The **convolutional layer** with the most parameters is conv2d_3 (Conv2D) which is the **last convolutional layer**. It has **9,348 parameters**, and we can check this with the equation for the number of parameters in a layer:

$$(\#InputChannels * kernelHeight * kernelWidth + 1) * (\#OutputChannels)$$
$$= (32 * 3 * 3 + 1) * (32) = (289) * (32) = 9,248$$

This large number makes sense because the last layer has one of the biggest outputs (32 output channels) and the biggest inputs (32 input channels). However, if you wanted to take **all layers** into account, keras states that the layer with most parameters would be the **dense layer** with **20,490 parameters**. This makes sense because the fully-connected (FC) layer takes in input as a vector x, and it's outputs are a function of a weighted sum of all the inputs such that $y = f(w^T x)$ where w are the weights/parameters and f is a non-linear function. Since it takes parameters from every element in the input vector x, it makes sense why this has the most total parameters. The training data gives an accuracy of around 62%, while the test data gives a lower accuracy of around 51%. It is sensible that the accuracy can be lower for test data because the model is using data that it has not been seen before (unlike the training data, which it has seen already). I did the following to compare this performance with a baseline that predicts the most common label (I had to flatten the matrices beforehand using code from section 1.4 in the appendix):

```
if compareWithBaseline:
    flattened_x_train = flatten_matrix(x_train)
    flattened_x_test = flatten_matrix(x_test)
    dummy_clf = DummyClassifier(strategy="most_frequent")
    dummy_clf.fit(flattened_x_train, y_train_before_categorical)
    DummyClassifierMeanAccuracy = dummy_clf.score(flattened_x_test,
    y_test_before_categorical)
    print("Dummy classifier's mean accuracy: " +
    str(DummyClassifierMeanAccuracy))
```

The results from the baseline comparison gives us a score of 10% (0.1) which is expected because $num\_classes = 10$. This comparison leads us to believe that the convolutional network is worthwhile since it's accuracy is around 62% (using training data) or 51% (using test data), and is much better than 10%.

Part (b) (ii)

Using the history variable, we can plot accuracy and loss shown below:

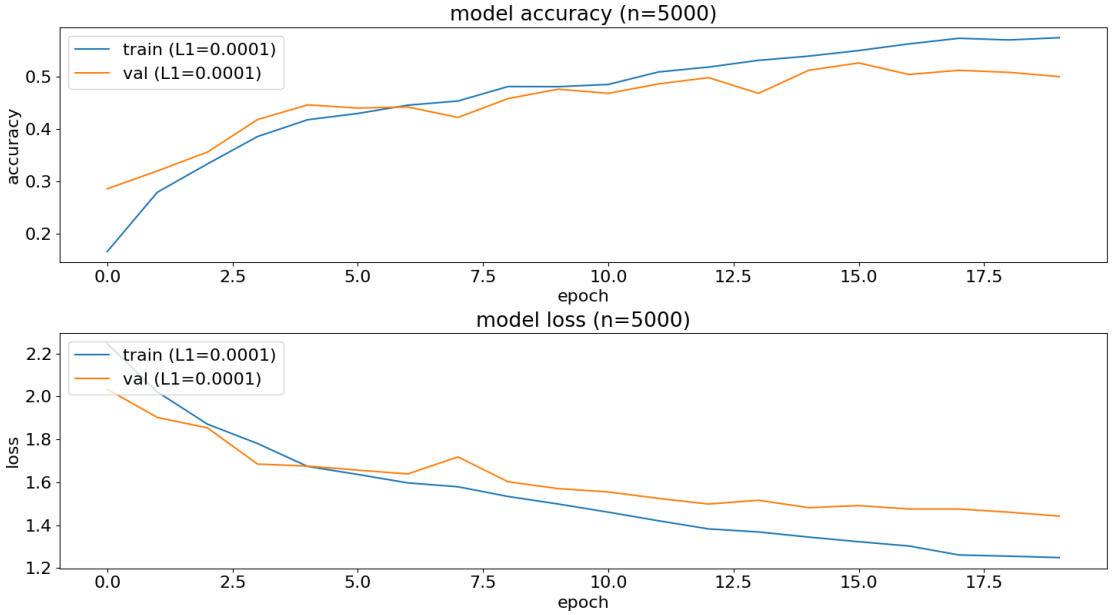


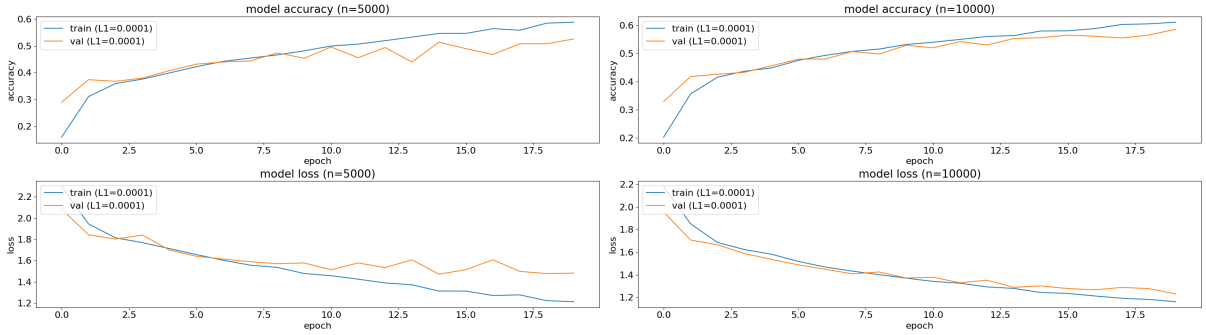
Figure 3: Plots showing epoch vs accuracy (top) and epoch vs loss (bottom).

Diagnostics which we can deduce from these plots include the fact that the accuracy for the training and test data (validation) diverges when the epoch $\gtrapprox 15$ which is a clear indicator of over-fitting. This is because the model depends too much on the training data, and gives a high accuracy for it. Meanwhile, it struggles with new (test/validation) data which is why the orange line dips downward. This hypothesis is also validated by the model loss plot where the orange line for the test/validation data struggles to decrease after epoch $\gtrapprox 15$ while the blue line for the training data continues to soar downwards.

Part (b) (iii)

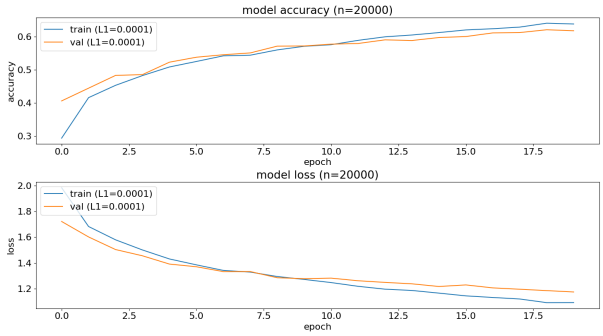
I trained the ConvNet using 5K, 10K, 20K and 40K training data points and recorded the time taken to explore the effect of increasing the training data size (n). The code and plots for each training data size can be found below:

```
convolutional_network(n=5000, recordTime=True)
convolutional_network(n=10000, recordTime=True)
convolutional_network(n=20000, recordTime=True)
convolutional_network(n=40000, recordTime=True)
```

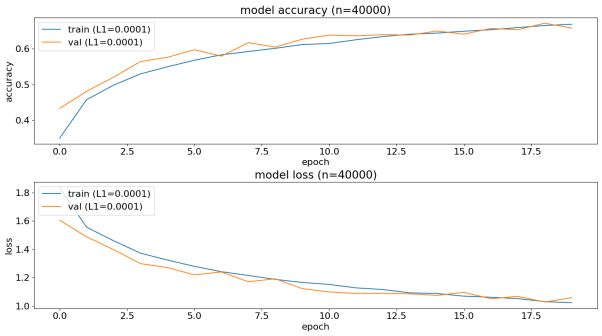


(a) n=5,000

(b) n=10,000



(c) n=20,000.



(d) n=40,000

Figure 4: Plots for epoch vs accuracy & epoch vs loss for various training data sizes (n).

We can see from the plots of the **history variable** above that generally the **prediction accuracy** of the training data $\approx 60\%$ when $epoch = 20$, no matter the training size. There is only a slight improvement in the accuracy with increased training size. On the other hand, there is a more noticeable difference for test data, as the prediction accuracy gets very close to the prediction accuracy of the training data as n tends to 40,000. This leads us to believe that the more training data we use, the better the network will be (indicating that there will be a **smaller chance of over-fitting** with more training data). The code below counts the time taken to train the network (with results):

```
if recordTime:
    time_before_training = time.time()
    history = model.fit(x_train, y_train, batch_size=batch_size,
    epochs=epochs, validation_split=0.1)
    print("Time taken to train network: " + str(time.time() -
    time_before_training))
```

Results			
n	**Training data accuracy**	**Testing data accuracy**	**Time**
5,000	62%	51%	115 seconds
10,000	69%	56%	233 seconds
20,000	70%	63%	632 seconds
40,000	71%	66%	917 seconds

From the plots and the results, we can determine that using $n = 40,000$ (as much training data as possible) seems to be the best option. The training data gives an accuracy of 71% without any over-fitting which is ideal. However, we can see the increase in time from using more data points above (it took 917 seconds to use 40,000 training data points), therefore it is necessary to find a 'Goldilocks Zone' where the trade-off between both are covered. We can see that there is only a 1% difference between using n=20,000 and n=40,000 training data points however there is a proportional increase in time to train both networks (917-632=285 second difference). Therefore, I would recommend using 20,000 for the training data size if you didn't change the L1 penalty (40,000 might potentially suit a faster & quicker machine where there isn't as much of a time difference).

Part (b) (iv)

Using 5K data points, I used a wide range of values for L1 (0, 0.0001, 0.001, 0.01, 1, 50) to analyze its effect on the prediction accuracy on the training and test data using the code below:

```
convolutional_network(n=5000, L1_values=[0, 0.0001, 0.001, 0.01, 1,
50], showOnlyAccuracyPlot = True)
```

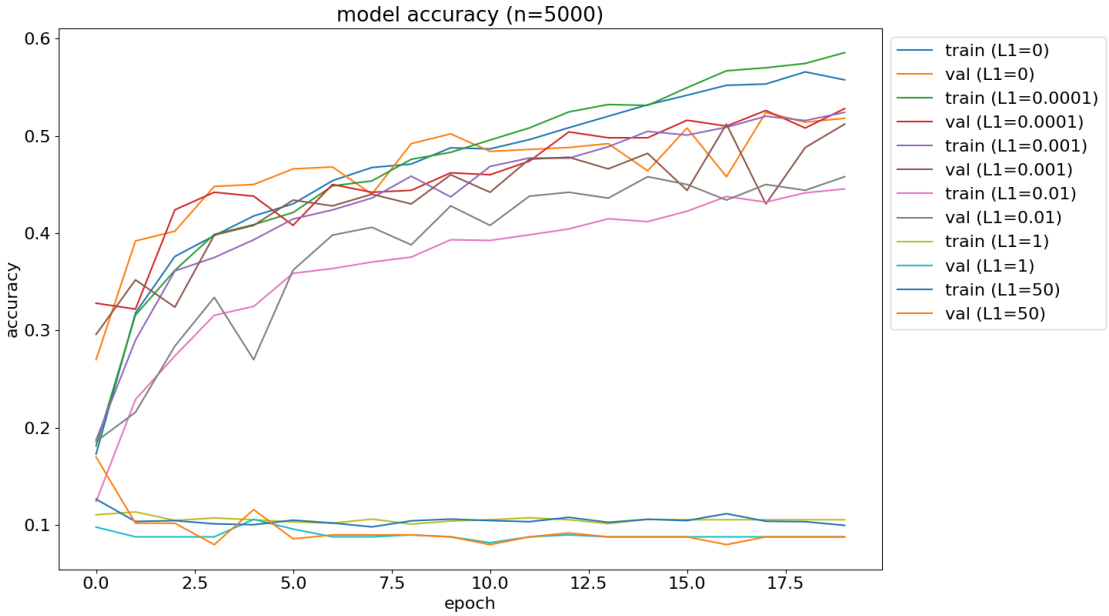


Figure 5: Plot of epoch vs accuracy for various L1 penalty values & n=5000.

It's clear to see that the prediction accuracy goes down when the L1 penalty is big (≥ 1). However, it's also clear that the divergence (i.e. going in different directions) of the prediction accuracy for training and test data points increases when the L1 penalty is small (≈ 0), which we mentioned in part (ii) as being over-fitting. The best value for L1 therefore will manage the trade-off between being too small and too big. To help us find this L1 value, we can compare/contrast with increasing the amount of training data to n=40,000, while using values for L1 that we already know do not over-fit but also yield good accuracy from figure 5 ($0.0001 \leq$ L1 ≤ 0.01). We can do this with the code below:

```
convolutional_network(n=40000, L1_values=[0.0001, 0.001, 0.01],
showOnlyAccuracyPlot=True)
```

The plot below shows the results of using n=40,000. We can see that L=0.0001 has the best accuracy, while also not showing signs of over-fitting (since the prediction accuracy using the training data increases steadily along with the prediction accuracy using the test data, without any sudden flattening for one on it's own). Hence, I would recommend using L1=0.0001 and n=40000 to prevent over-fitting while also getting the highest accuracy. A better method to select L1 would be using cross-validation but since convolutional networks take long periods of time to train, it is better to use the hold-out technique to try a few values for L1.

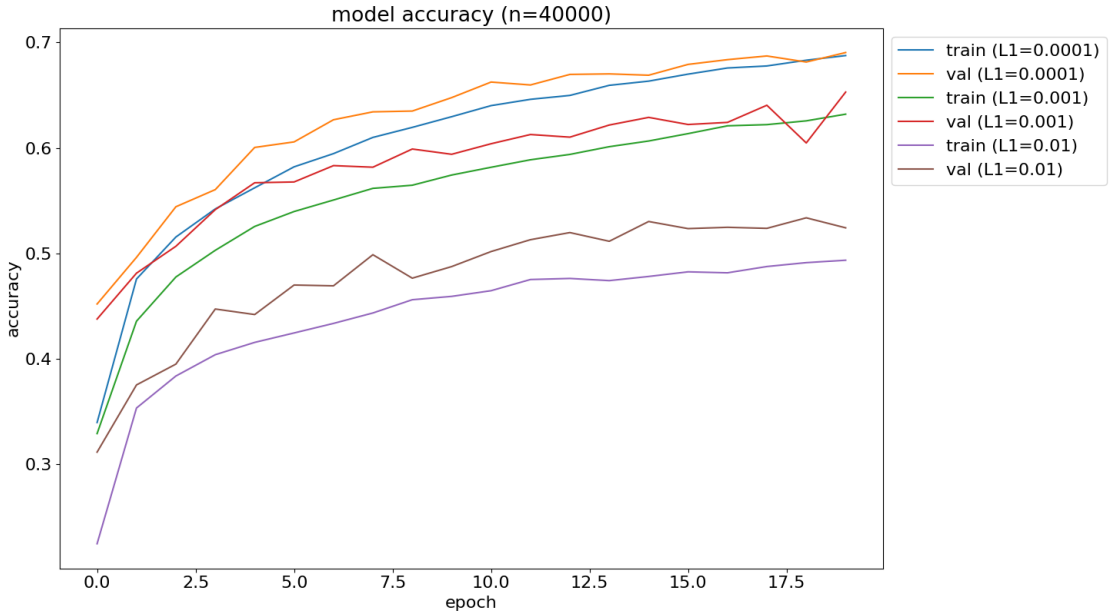


Figure 6: Plot of epoch vs accuracy for various L1 penalty values & n=40000.

Part (c) (i)

I modified the code to use max-pooling by removing strides and adding MaxPooling2D as shown using the code below:

```
if useMaxPooling:
    model.add(Conv2D(16, (3, 3), padding='same',
    input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), padding='same',
    input_shape=x_train.shape[1:], activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

Part (c) (ii)

Keras states that this ConvNet has 37,146 parameters, which is the same as before. This is logical as the (2,2) stride from before outputs the same size as the new (2,2) max-pool layer. Using 5K training data points, the addition of the max-pool layer resulted in an increased time to train by 96 seconds (211-115 = 96 second difference) compared to the original network. This also resulted in an accuracy of 65% for training data and

an accuracy of 54% for test data, which is more than the previous accuracy of 62% for training data and 51% for test data for the original network. The training time has changed because using strides means that it can leave out some values in the matrix while max-pooling on the other hand has to do a lot more computation using algorithms to find an array's biggest value.

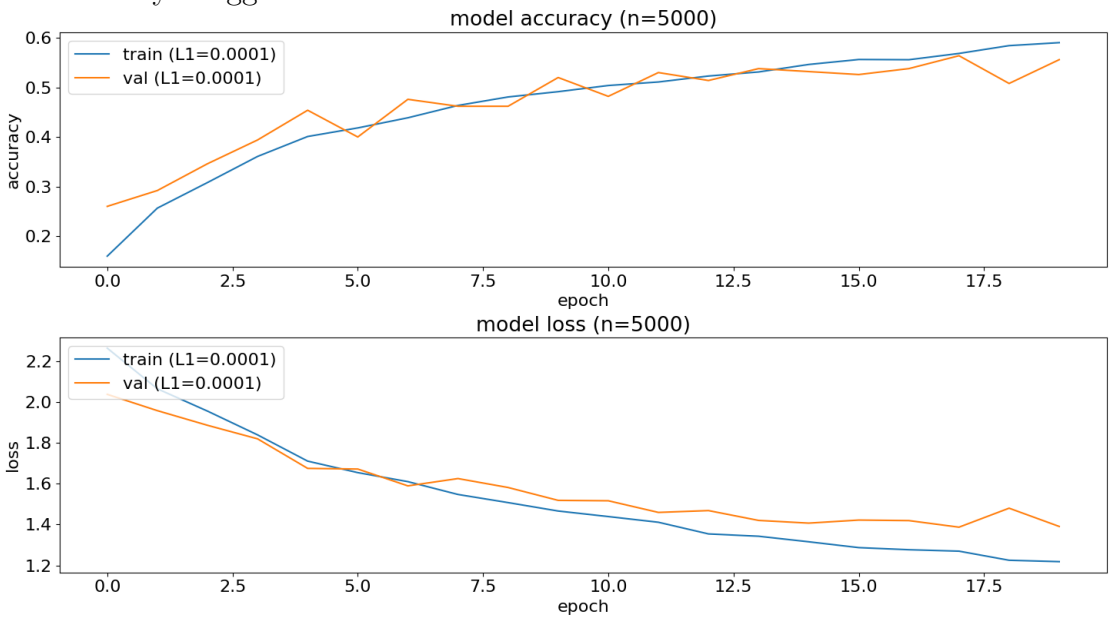


Figure 7: Max-pooling used with ConvNet with n=5000.

Since it's clear now that using max-pooling is more accurate but requires more algorithmic computation, I then tried to use n=40,000 to see if the increased prediction accuracy would carry over.

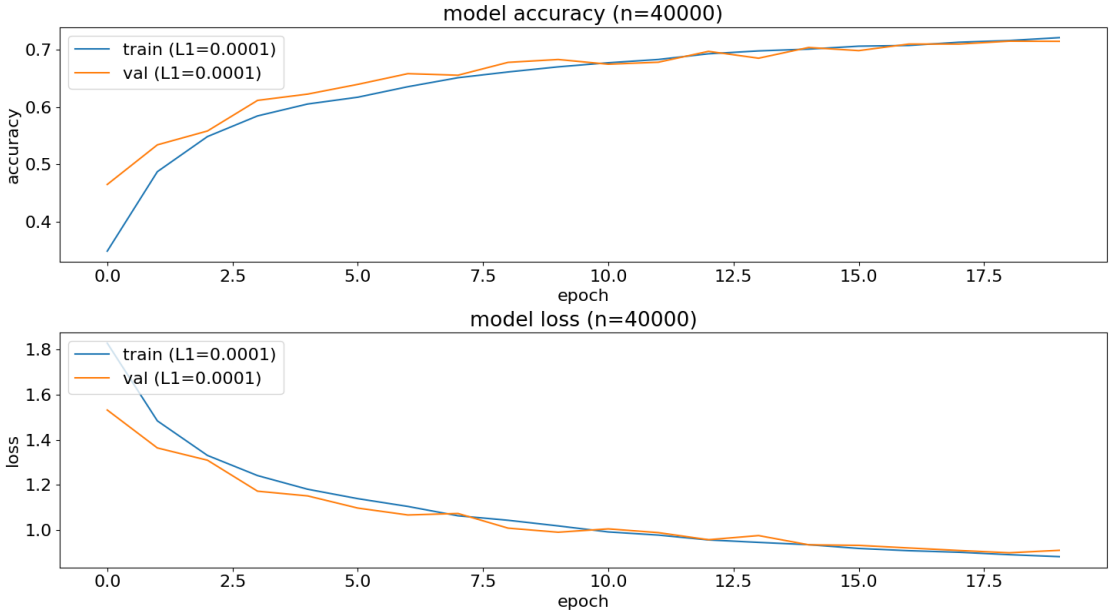


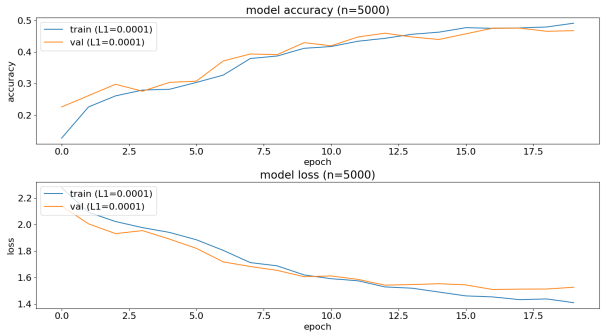
Figure 8: Max-pooling used with ConvNet with n=40000.

The time it took to train the new network was 1730 seconds, compared to the 917 seconds it took for the original network. Also, the prediction accuracy using n=40000 was 77% for the training data, and 73% for the test data, compared to 71% for the training data and 66% for the test data using the original network. Hence, it is clear that using max-pooling increases the accuracy, but takes around 1.88x more time to train the network.

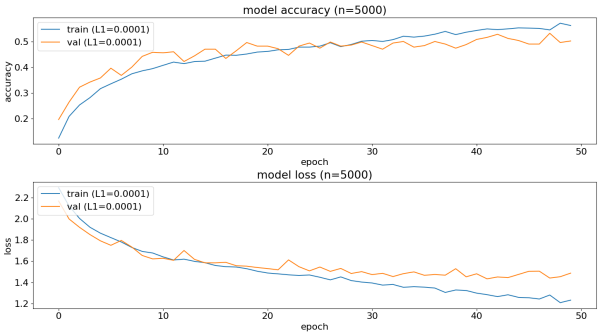
Part (d) [OPTIONAL]

I played around with the ConvNet architectures by making it thinner and deeper. I did this with the code below:

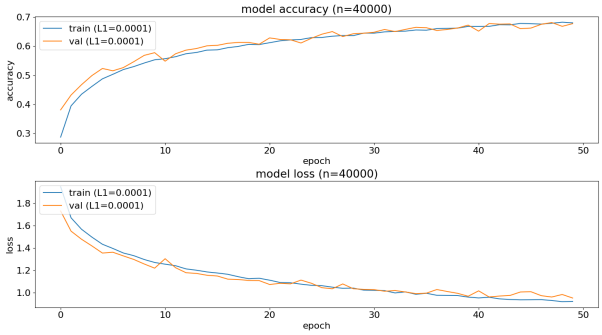
```
elif thinnerAndDeeper:
    model.add(Conv2D(8, (3, 3), padding='same',
    input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(8, (3, 3), strides=(2, 2), padding='same',
    activation='relu'))
    model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same',
    activation='relu'))
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same',
    activation='relu'))
```



(a) n=5,000 & epochs=20



(b) n=5000 & epochs=50



(c) n=40,000 & epochs=50

Figure 9: Plots for epoch vs accuracy & epoch vs loss for various training data sizes (n) using a thinner and deeper ConvNet.

For n=5000, we get a predicted accuracy of 53% for training data and 45% for test data, which is comparable but slightly worse than the predicted accuracy of 62% for training data and 51% for test data which we got in the original network. It took 92 seconds to train, which is 23 seconds faster than the original network (115-93=23 seconds). However, there are less signs of over-fitting with this network than the initial network as seen in figure 9 (a) since there is less of a divergence between the prediction accuracy using the training and test data. Because of this, we can increase the number of epochs to see how this network reacts.

For n=5000 and epochs=50 as seen in figure 9 (b), we get a predicted accuracy of 64% for training data and 51% for test data, which is comparable and slightly better than

the predicted accuracy of 62% for training data and 51% for test data which we got in the original network. It took 222 seconds to train, which is 107 seconds slower than the original network (222-115=107 seconds). The main problem is that we can see that the network is over-fitted as there is a noticeable divergence between the orange and blue lines. From my experience in this assignment, I thought that a good idea would be to increase the training data size to see if this would eliminate that issue, so I decided to use n=40000.

For n=40000 and epochs=50 as seen in figure 9 (c), we get a predicted accuracy of 75% for training data and 68% for test data, which is slightly better than the predicted accuracy of 71% for training data and 66% for test data which we got in the original network. It took 1784 seconds to train, which is 867 seconds slower than the original network (1784-917=867 seconds). An accuracy this high, with minimal over-fitting (both orange and blue lines closely follow each other, meaning the model doesn't depend on the training data too much and the test data too little) was the best I could come up with, while also not exceeding the execution time over 2000 seconds.

From my experiments, I can see that adding more layers allows more features to be used from the training data since there will be more parameters from the fact that there are more weights. But adding too many layers causes over-fitting, while adding few layers will cause under-fitting. We can see from the experiments above that that the prediction performance increases when n (the number of training data points) increases. A large number for n also has the benefit of allowing the network to more easily use more features from the training data (using n=40,000 had less over-fitting than n=5,000). At the same time, it takes longer to train the network when we provide more training data points. I've learnt that to find the best possible trade-off, you can consider as many layers as possible until there are signs of over-fitting, and that is when you can manage the over-fitting by adding more training data for example.

1 Appendix

1.1 Question (i) (a)

```
import numpy as np
from PIL import Image

def part_a(NxN_Array, KxK_Kernel):
    returningMatrixSize = (NxN_Array.shape[0] - KxK_Kernel.shape[0]) + 1
    returningMatrix = np.empty([returningMatrixSize, returningMatrixSize])
    for iterationOneFromMatrixSize in range(returningMatrixSize):
        for iterationTwoFromMatrixSize in range(returningMatrixSize):
            summation = 0
            for iterationOneFromKernel in range(KxK_Kernel.shape[0]):
                for iterationTwoFromKernel in range(KxK_Kernel.shape[0]):
                    summation = summation+NxN_Array[iterationOneFromKernel
                                + iterationOneFromMatrixSize]
                                [iterationTwoFromKernel +
                                iterationTwoFromMatrixSize] *
                                KxK_Kernel[iterationOneFromKernel]
                                          [iterationTwoFromKernel]
```

```
            returningMatrix[iterationOneFromMatrixSize]
                           [iterationTwoFromMatrixSize] = summation
    return returningMatrix
```

1.2 Question (i) (b)

```
def part_b():
    # Code given in question:
    im = Image.open('pound.png')
    rgb = np.array(im.convert('RGB'))
    r = rgb[:, :, 0]  # array of R pixels
    img_array = np.uint8(r)

    # Using only kernel #1
    firstKernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    imageOneUsingFirstKernel = part_a(img_array, firstKernel)
    Image.fromarray(imageOneUsingFirstKernel).show()

    # Using only kernel #2
    secondKernel = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])
    imageTwoUsingSecondKernel = part_a(img_array, secondKernel)
    Image.fromarray(imageTwoUsingSecondKernel).show()

    # Using both kernel #1 and kernel #2
    firstAndSecondKernel = part_a(img_array, firstKernel)
    firstAndSecondKernel = part_a(firstAndSecondKernel, secondKernel)
    Image.fromarray(firstAndSecondKernel).show()
```

1.3 Question (ii) modified code

```
import warnings
import matplotlib
from sklearn.dummy import DummyClassifier
from itertools import cycle
import numpy as np
from tensorflow.python.keras import layers, regularizers
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
from tensorflow import keras
from keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
import time

matplotlib.use('TkAgg')


def convolutional_network(n=None, compareWithBaseline=None, epochs=None,
recordTime=None, L1_values=None,
                          showOnlyAccuracyPlot=None, useMaxPooling=None,
                          thinnerAndDeeper=None):
```

```
# Set all default values for variables if they are not passed in.
if n is None:
    n = 5000
if compareWithBaseline is None:
    compareWithBaseline = False
if epochs is None:
    epochs = 20
if recordTime is None:
    recordTime = False
if L1_values is None:
    L1_values = [0.0001]
if showOnlyAccuracyPlot is None:
    showOnlyAccuracyPlot = False
if useMaxPooling is None:
    useMaxPooling = False
if thinnerAndDeeper is None:
    thinnerAndDeeper = False

# Using this command to ensure the graphs fit properly in the plot
without overlapping
plt.rcParams['figure.constrained_layout.use'] = True
plt.rc('font', size=16)
for L1 in L1_values:
    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)

    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) =
    keras.datasets.cifar10.load_data()

    x_train = x_train[1:n];
    y_train = y_train[1:n]
    # x_test=x_test[1:500]; y_test=y_test[1:500]

    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    print("orig x_train shape:", x_train.shape)

    y_train_before_categorical = y_train
    y_test_before_categorical = y_test

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    use_saved_model = False
    if use_saved_model:
        model = keras.models.load_model("cifar.model")
    else:
        model = keras.Sequential()

        if useMaxPooling:
```

```
        model.add(Conv2D(16, (3, 3), padding='same',
        input_shape=x_train.shape[1:], activation='relu'))
        model.add(Conv2D(16, (3, 3), padding='same',
        input_shape=x_train.shape[1:], activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Conv2D(32, (3, 3), padding='same',
        activation='relu'))
        model.add(Conv2D(32, (3, 3), padding='same',
        activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
    elif thinnerAndDeeper:
        model.add(Conv2D(8, (3, 3), padding='same',
        input_shape=x_train.shape[1:], activation='relu'))
        model.add(Conv2D(8, (3, 3), strides=(2, 2),
        padding='same', activation='relu'))
        model.add(Conv2D(16, (3, 3), padding='same',
        activation='relu'))
        model.add(Conv2D(16, (3, 3), strides=(2, 2),
        padding='same', activation='relu'))
        model.add(Conv2D(32, (3, 3), padding='same',
        activation='relu'))
        model.add(Conv2D(32, (3, 3), strides=(2, 2),
        padding='same', activation='relu'))
    elif (not useMaxPooling) and (not thinnerAndDeeper):
        model.add(Conv2D(16, (3, 3), padding='same',
        input_shape=x_train.shape[1:], activation='relu'))
        model.add(Conv2D(16, (3, 3), strides=(2, 2),
        padding='same', activation='relu'))
        model.add(Conv2D(32, (3, 3), padding='same',
        activation='relu'))
        model.add(Conv2D(32, (3, 3), strides=(2, 2),
        padding='same', activation='relu'))

    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax',
    kernel_regularizer=regularizers.l1(L1)))
    model.compile(loss="categorical_crossentropy",
    optimizer='adam', metrics=["accuracy"])
    model.summary()
    batch_size = 128

    if recordTime:
        time_before_training = time.time()
        history = model.fit(x_train, y_train,
        batch_size=batch_size, epochs=epochs, validation_split=0.1)
        print("Time taken to train network: " + str(time.time() -
        time_before_training))
    else:
        history = model.fit(x_train, y_train,
        batch_size=batch_size, epochs=epochs, validation_split=0.1)

    model.save("cifar.model")

    if showOnlyAccuracyPlot:
```

```
                plt.plot(history.history['accuracy'], label='train (L1=' +
                str(L1) + ')')
                plt.plot(history.history['val_accuracy'], label='val (L1='
                + str(L1) + ')')
                plt.title('model accuracy (n=' + str(n) + ')')
                plt.ylabel('accuracy')
                plt.xlabel('epoch')
                plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
            else:
                plt.subplot(211)
                plt.plot(history.history['accuracy'], label='train (L1=' +
                str(L1) + ')')
                plt.plot(history.history['val_accuracy'], label='val (L1='
                + str(L1) + ')')
                plt.title('model accuracy (n=' + str(n) + ')')
                plt.ylabel('accuracy')
                plt.xlabel('epoch')
                plt.legend(loc='upper left')
                plt.subplot(212)
                plt.plot(history.history['loss'], label='train (L1=' +
                str(L1) + ')')
                plt.plot(history.history['val_loss'], label='val (L1=' +
                str(L1) + ')')
                plt.title('model loss (n=' + str(n) + ')')
                plt.ylabel('loss')
                plt.xlabel('epoch')
                plt.legend(loc='upper left')

    plt.show()

    preds = model.predict(x_train)
    y_pred = np.argmax(preds, axis=1)
    y_train1 = np.argmax(y_train, axis=1)
    # checkPerformanceWithTrainingData = Dictionary containing
    {classification report, confusion matrix}
    checkPerformanceWithTrainingData = {"classification-report":
    classification_report(y_train1, y_pred),
                                        "confusion-matrix":
                                        confusion_matrix(y_train1, y_pred)}
    print(checkPerformanceWithTrainingData["classification-report"])
    print(checkPerformanceWithTrainingData["confusion-matrix"])

    preds = model.predict(x_test)
    y_pred = np.argmax(preds, axis=1)
    y_test1 = np.argmax(y_test, axis=1)
    # checkPerformanceWithTestData = Dictionary containing {classification
    report, confusion matrix}
    checkPerformanceWithTestData = {"classification-report":
    classification_report(y_test1, y_pred),
                                    "confusion-matrix":
                                    confusion_matrix(y_test1, y_pred)}
    print(checkPerformanceWithTestData["classification-report"])
    print(checkPerformanceWithTestData["confusion-matrix"])

    if compareWithBaseline:
```

```
        # Check how this performance compares with a simple baseline that
        always predicts the most common label
        flattened_x_train = flatten_matrix(x_train)
        flattened_x_test = flatten_matrix(x_test)
        dummy_clf = DummyClassifier(strategy="most_frequent")
        dummy_clf.fit(flattened_x_train, y_train_before_categorical)
        DummyClassifierMeanAccuracy = dummy_clf.score(flattened_x_test,
        y_test_before_categorical)
        print("Dummy classifier's mean accuracy: " +
        str(DummyClassifierMeanAccuracy))
        return checkPerformanceWithTrainingData,
        checkPerformanceWithTestData, DummyClassifierMeanAccuracy
    else:
        return checkPerformanceWithTrainingData,
        checkPerformanceWithTestData
```

1.4 Question (ii) function to flatten matrix

```
def flatten_matrix(input_matrix):
    output_array = []
    for i in range(input_matrix.shape[0]):
        output_array.append(input_matrix[i].flatten(order='C'))
    return np.array(output_array)
```

1.5 Question (ii) calling modified code

```
if __name__ == '__main__':
    # Uncomment the part you wish to run below

    # PART (B) (I)
    # Keras says this model has 37,146 parameters.
    # The layer with the most parameters is the last layer (conv2d_3) with
    9248 parameters.
    convolutional_network(n=5000, compareWithBaseline=True)

    # PART (B) (II)
    # From the plots in function convolutional_network() above, we can see
    over-fitting since the training and
    # validation accuracy diverges when epoch is around 15 and greater.

    # PART (B) (III)
    # convolutional_network(n=5000, recordTime=True)
    # convolutional_network(n=10000, recordTime=True)
    # convolutional_network(n=20000, recordTime=True)
    # convolutional_network(n=40000, recordTime=True)

    # PART (B) (IV)
    # convolutional_network(n=5000, L1_values=[0, 0.0001, 0.001, 0.01, 1,
    50], showOnlyAccuracyPlot = True)
    # convolutional_network(n=40000, L1_values=[0.0001, 0.001, 0.01],
    showOnlyAccuracyPlot=True)
```

```
# PART (C) (I) AND (II)
# convolutional_network(n=5000, recordTime=True, useMaxPooling=True)
# convolutional_network(n=40000, recordTime=True, useMaxPooling=True)

# PART (D)
# convolutional_network(n=5000, recordTime=True, thinnerAndDeeper=True)
# convolutional_network(n=5000, recordTime=True, epochs=50,
thinnerAndDeeper=True)
# convolutional_network(n=40000, recordTime=True, epochs=50,
thinnerAndDeeper=True)
```