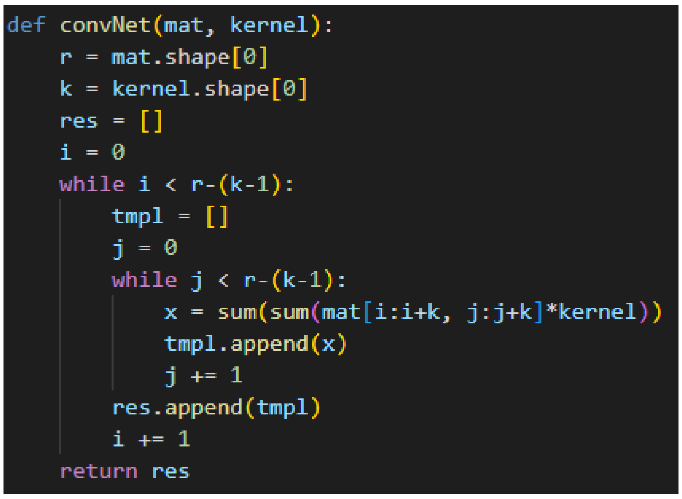
Machine Learning - Week 8 Assignment
Mark Lysaght - 19334391

i) a)
Using vanilla python, I created a function to convolve a $k\ x\ k$ kernel to a given $n\ x\ n$ array and returns the result.



```
def convNet(mat, kernel):
    r = mat.shape[0]
    k = kernel.shape[0]
    res = []
    i = 0
    while i < r-(k-1):
        tmpl = []
        j = 0
        while j < r-(k-1):
            x = sum(sum(mat[i:i+k, j:j+k]*kernel))
            tmpl.append(x)
            j += 1
        res.append(tmpl)
        i += 1
    return res
```

Figure 1. Convolution function

The method works by summing the product of a $k\ x\ k$ sized sliding window of the original matrix and a given kernel. In the innermost loop we append this sum to a list and slide across 1 pixel to get the next entry to our resultant matrix. When we reach the end of the first row, the window slides down 1 pixel and the process repeats across the next row and so on. At the end of each row, a list containing all the convolved values is added to the result matrix and when all rows have been computed, the result matrix is returned.
The stride is 1 in my version of this function but it can be changed by increasing the iteration of the loops.

b)
I selected an image of a chess piece to run through my function.

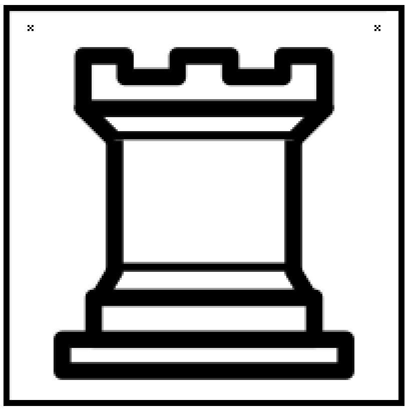
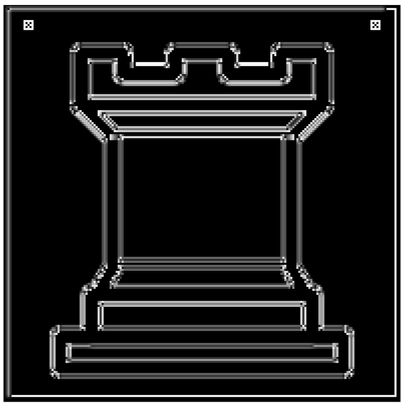


Figure 2. Raw image used in part b)

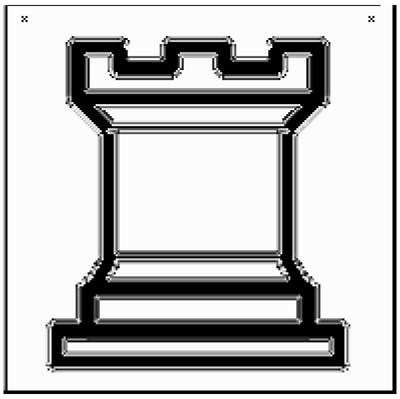
When kernel1 was used in my convolution function with the image in figure 2, I got the following result.



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

Figure 3. Image after convolution with kernel1

When kernel2 was used in my convolution function with the image in figure 2, I got the following result.



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

Figure 4. Image after convolution with kernel2

ii) a)

The architecture of the ConvNet used in the week8.py file has four convolutional layers, a dropout layer and a softmax pooling layer. The model starts with input_shape = (32, 32, 3).

The first convolution layer has a 3x3 kernel with 16 output channels. After the convolution to the input_shape the new model becomes (32, 32, 16).

```
model.add(Conv2D(16,(3,3),padding='same',input_shape = x_train.shape[1:],
activation='relu'))
```

The second convolution layer also has a 3x3 kernel with 16 output channels. However, the stride is now 2 so this halves the size of our model's shape from (32, 32, 16) to (16, 16 ,16).

```
model.add(Conv2D(16,(3,3),strides=(2,2),padding='same', activation='relu'))
```

The third convolution layer has a 3x3 kernel with 32 output channels. This changes our model shape from (16,16,16) to (16,16,32).

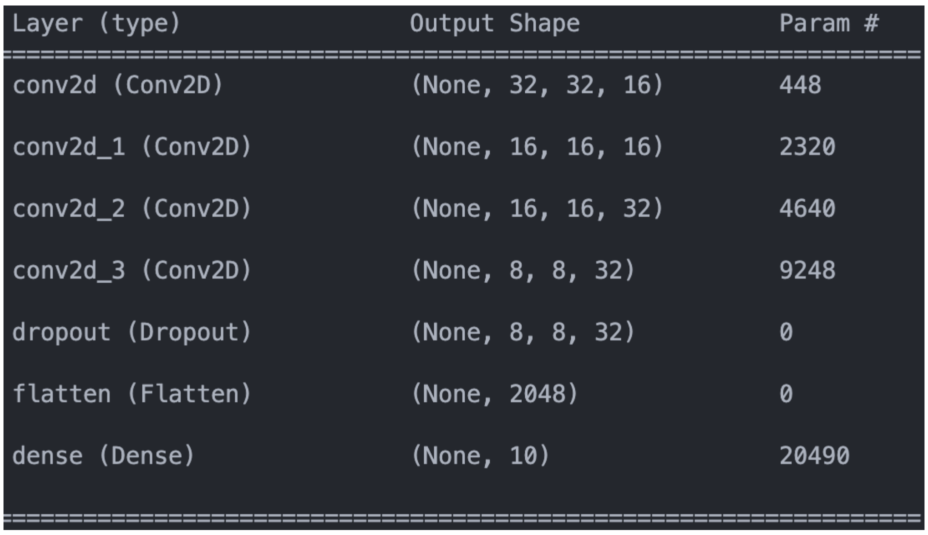
```
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
```

The fourth convolution layer has a 3x3 kernel with 32 output channels. The stride for this layer is 2 so the shape of our model is halved again from (16,16,32) to (8,8,32).

```
model.add(Conv2D(32,(3,3),strides=(2,2),padding='same',activation='relu'))
```

A dropout layer is applied to the model. Dropout regularisation is applied to the model to reduce overfitting by randomly dropping some nodes.
The last layer is a fully connected layer. Softmax is applied to the data to normalise it over a probability distribution.

b) i)
After running the code, the output shows that the model has 37,416 total parameters. These parameters are split between the multiple layers.



```
Layer (type)                Output Shape              Param #
=================================================================
conv2d (Conv2D)             (None, 32, 32, 16)        448

conv2d_1 (Conv2D)           (None, 16, 16, 16)        2320

conv2d_2 (Conv2D)           (None, 16, 16, 32)        4640

conv2d_3 (Conv2D)           (None, 8, 8, 32)          9248

dropout (Dropout)           (None, 8, 8, 32)          0

flatten (Flatten)           (None, 2048)              0

dense (Dense)               (None, 10)                20490

=================================================================
```

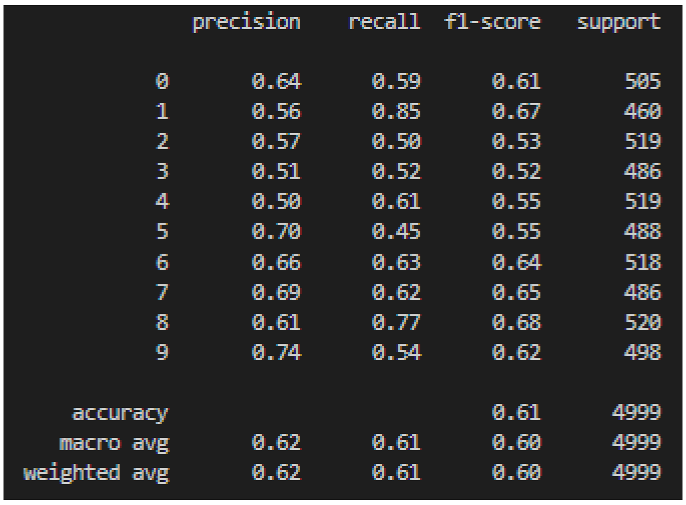
Figure 5. Output showing distribution of parameters to each layer

On the right hand side of figure 5, we can see how many parameters are used in each layer. The fourth layer has the most parameters, 9248. This is equal to:

$$output\ channels \times (input\ channels \times kernelwidth \times kernelheight\ + 1)$$

$$32 \times (32 \times 3 \times 3 + 1) = 9248$$

The model performs better on the training data than on the test data as we can see from the tables below. The f1 score on average for the training data gave an accuracy of 0.61 whereas for the test data it gave an accuracy of 0.51. The closer an f1 score is to 1, the more accurate the model is. We can see also in the tables that there is more consistency in the training data’s f1 score than in the test data’s.

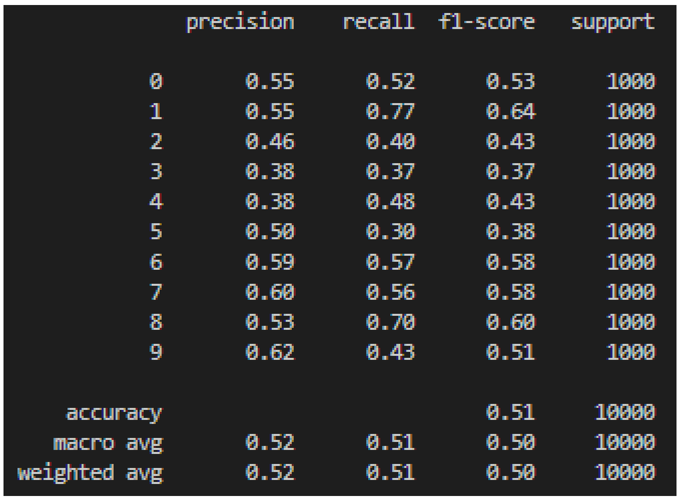


```
              precision    recall  f1-score   support

           0       0.64      0.59      0.61       505
           1       0.56      0.85      0.67       460
           2       0.57      0.50      0.53       519
           3       0.51      0.52      0.52       486
           4       0.50      0.61      0.55       519
           5       0.70      0.45      0.55       488
           6       0.66      0.63      0.64       518
           7       0.69      0.62      0.65       486
           8       0.61      0.77      0.68       520
           9       0.74      0.54      0.62       498

    accuracy                           0.61      4999
   macro avg       0.62      0.61      0.60      4999
weighted avg       0.62      0.61      0.60      4999
```

Figure 6. Performance on training data



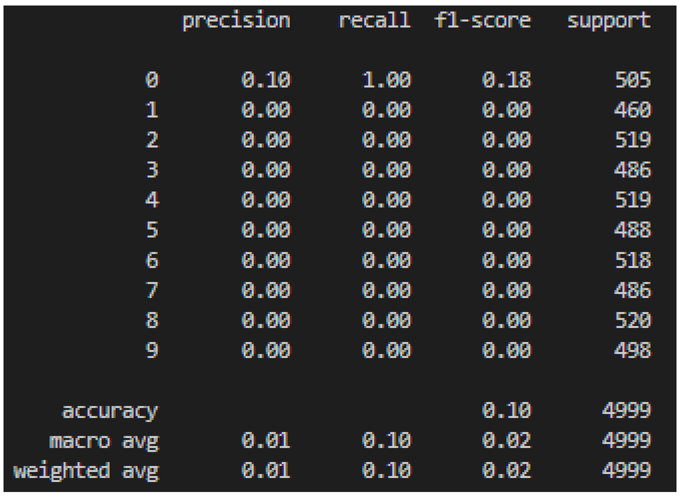
```
              precision    recall  f1-score   support

           0       0.55      0.52      0.53      1000
           1       0.55      0.77      0.64      1000
           2       0.46      0.40      0.43      1000
           3       0.38      0.37      0.37      1000
           4       0.38      0.48      0.43      1000
           5       0.50      0.30      0.38      1000
           6       0.59      0.57      0.58      1000
           7       0.60      0.56      0.58      1000
           8       0.53      0.70      0.60      1000
           9       0.62      0.43      0.51      1000

    accuracy                           0.51     10000
   macro avg       0.52      0.51      0.50     10000
weighted avg       0.52      0.51      0.50     10000
```

Figure 7. Performance on test data

Comparing against a baseline model that predicts the most common label. In this case we can see that the model only predicts class 0. The accuracy of the model is very poor as it can only distinguish 1 of the 10 classes.



```
              precision    recall  f1-score   support

           0       0.10      1.00      0.18       505
           1       0.00      0.00      0.00       460
           2       0.00      0.00      0.00       519
           3       0.00      0.00      0.00       486
           4       0.00      0.00      0.00       519
           5       0.00      0.00      0.00       488
           6       0.00      0.00      0.00       518
           7       0.00      0.00      0.00       486
           8       0.00      0.00      0.00       520
           9       0.00      0.00      0.00       498

    accuracy                           0.10      4999
   macro avg       0.01      0.10      0.02      4999
weighted avg       0.01      0.10      0.02      4999
```

Figure 8. Baseline model predicting the most common label

ii)

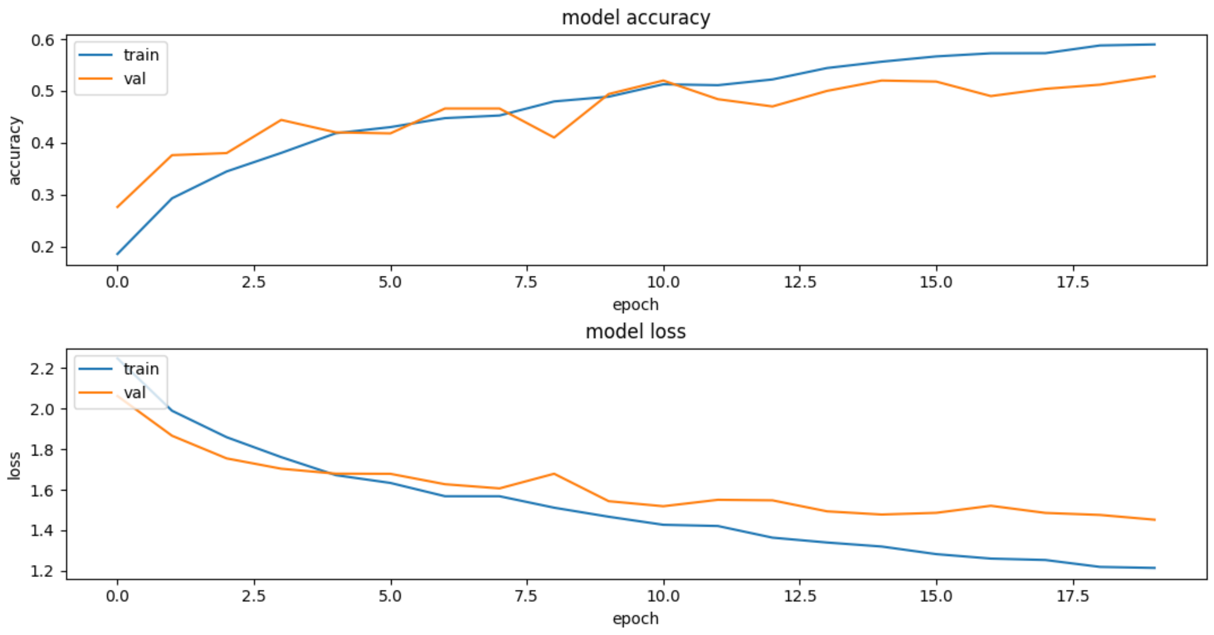


Figure 9. Model accuracy and model loss over 20 epochs

In the accuracy plot above, we can see that after 11 or so epochs the training accuracy succeeds the accuracy of the validation data and looks to become increasingly accurate while the validation data accuracy levels off. This indicates the model starts to overfit after this point as the model is fitting to the noise in the training data.
The loss plot further reinforces the diagnosis that the model is overfitting after around 11-12 epochs as the validation data looks to level out while the training data tends closer to zero.

iii)
To vary the amount of data points we can change the variable n in the code. It took around 45 seconds to run with 5000 data points, around a minute and 20 seconds for 10000 data points, 2 minutes 40 seconds for 20000 data points, and about 5 minutes for 40000 data points. Below are the model accuracy and loss plots for each run.
From the plots, we can see an increase in accuracy with larger sizes of the training data. The validation data is a lot clearer to see change in as n gets larger. When n is 40000 we can see a drastic difference from when n was 5000 in that it fits to the accuracy and loss of the training data a lot better.

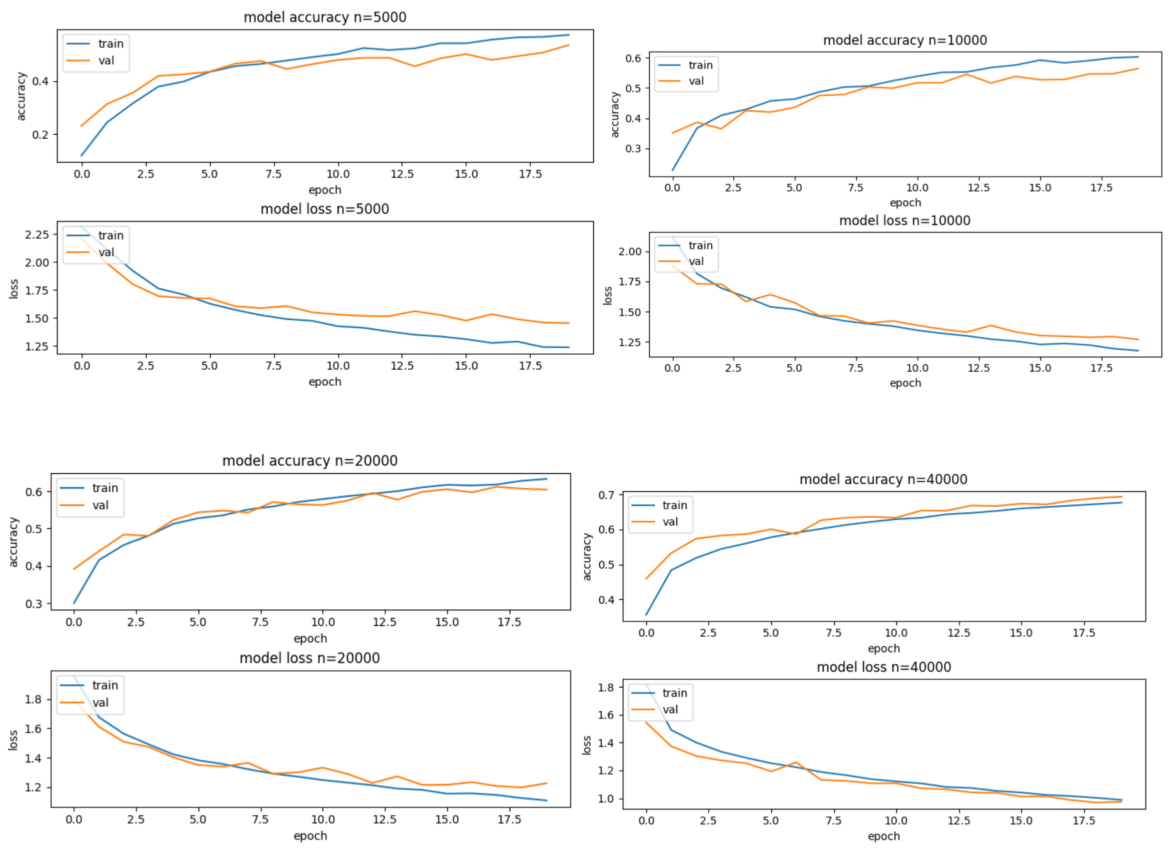


Figure 10. Model accuracy and loss plots for varying sizes of training data.

iv)
Varying the L1 penalty on the softmax output layer has a significant effect on the model accuracy. Below are the accuracy plots for larger values of L1. We can see from figure 11 that for larger L1 penalties, the accuracy gets worse as a larger weight being put on the penalty means that more model parameters are set to 0 and thus, they don't affect the model.

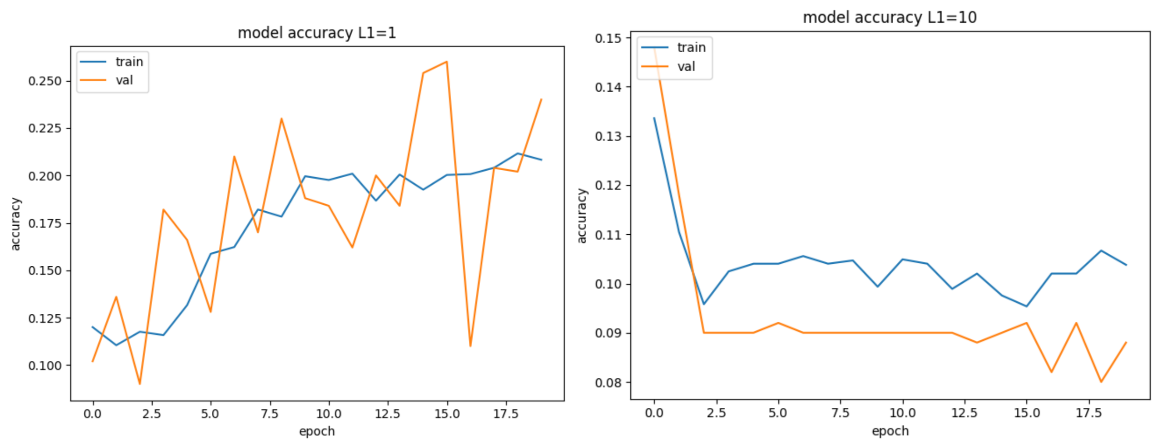


Figure 11. Model accuracy with larger L1 weight penalties.

In regards to increasing the amount of training data or changing the penalty, it would seem that a larger size of training data is more effective at managing overfitting.

c) i)
I modified the model to employ max-pooling by replacing the 16 channel strided layer with a 16 channel same layer followed by a max-pool layer and similar for the 32 channel layers. Below is a plot of the model accuracy and loss from this network utilising max-pooling along with the code of the new network architecture.

```
model = keras.Sequential()
model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:],
          activation='relu'))
model.add(Conv2D(16, (3, 3), padding='same', 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)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax',
          kernel_regularizer=regularizers.l1(L1)))
```

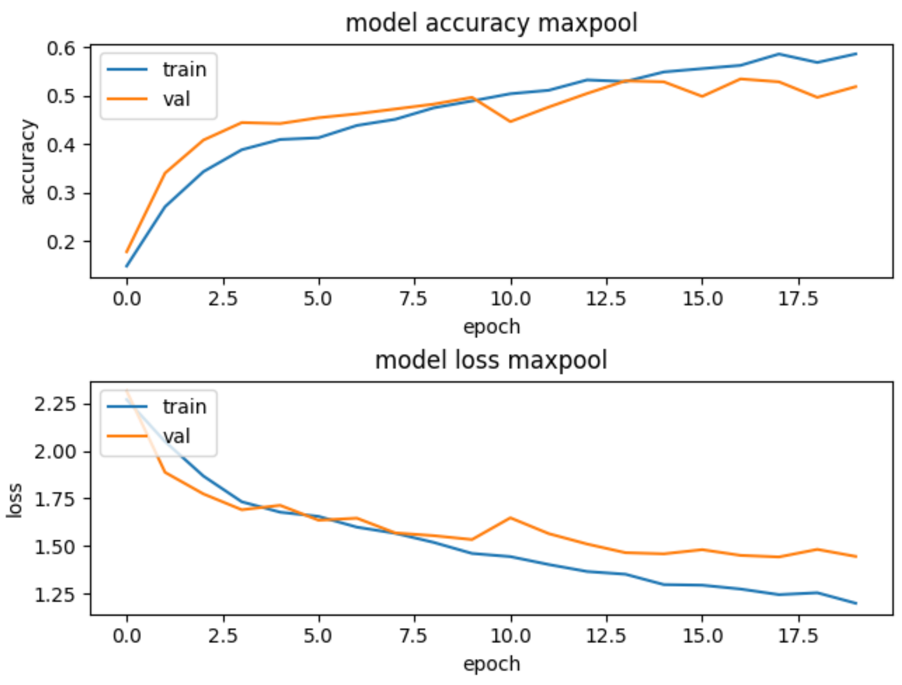
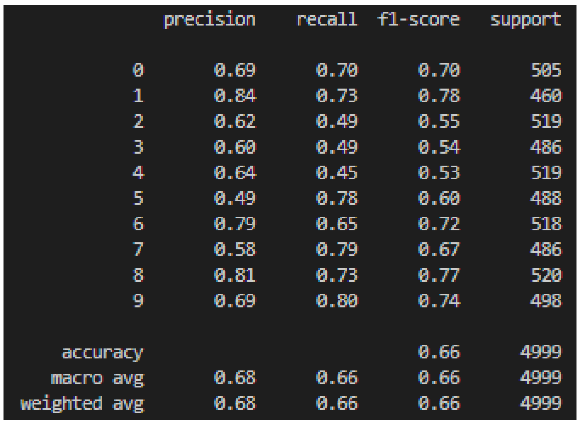


Figure 12. Model accuracy and model loss with max pooling.

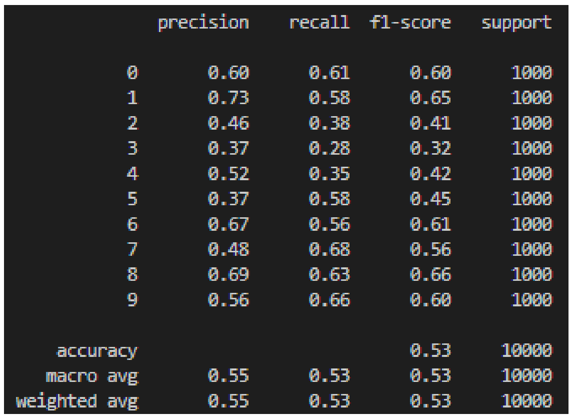
ii)
This ConvNet has the same number of parameters as before, 37146.



```
              precision    recall  f1-score   support

           0       0.69      0.70      0.70       505
           1       0.84      0.73      0.78       460
           2       0.62      0.49      0.55       519
           3       0.60      0.49      0.54       486
           4       0.64      0.45      0.53       519
           5       0.49      0.78      0.60       488
           6       0.79      0.65      0.72       518
           7       0.58      0.79      0.67       486
           8       0.81      0.73      0.77       520
           9       0.69      0.80      0.74       498

    accuracy                           0.66      4999
   macro avg       0.68      0.66      0.66      4999
weighted avg       0.68      0.66      0.66      4999
```



```
              precision    recall  f1-score   support

           0       0.60      0.61      0.60      1000
           1       0.73      0.58      0.65      1000
           2       0.46      0.38      0.41      1000
           3       0.37      0.28      0.32      1000
           4       0.52      0.35      0.42      1000
           5       0.37      0.58      0.45      1000
           6       0.67      0.56      0.61      1000
           7       0.48      0.68      0.56      1000
           8       0.69      0.63      0.66      1000
           9       0.56      0.66      0.60      1000

    accuracy                           0.53     10000
   macro avg       0.55      0.53      0.53     10000
weighted avg       0.55      0.53      0.53     10000
```

Figure 13. Accuracy of training(left) and test(right) data with max pooling

We can see from figure 13 that using max-pooling instead of the original model yields slightly more accurate results. Comparison with figures 6 and 7. We see an increase in the accuracy of the training data from 0.61 to 0.66, a 5 percent increase, and an increase in the accuracy of the test data from 0.51 to 0.53, a 2 percent increase. The new network took about double the amount of time to train as the original network for 5000 training data points. This is because the max-pooling requires some computation whereas the same computation is not required when using strides alone. The increase in accuracy promotes the use of this method but a trade-off between accuracy and training time must be considered depending on the size of the training data. If I were to train 40000 data points with this network I would expect to have to wait around 10 minutes for training to complete.

APPENDIX FOR CODE
ass4.py

```python
from PIL import Image
import numpy as np
def convNet(mat, kernel):
    r = mat.shape[0]
    k = kernel.shape[0]
    res = []
    i = 0
    while i < r-(k-1):
        tmpl = []
        j = 0
        while j < r-(k-1):
            x = sum(sum(mat[i:i+k, j:j+k]*kernel))
            tmpl.append(x)
            j += 1
        res.append(tmpl)
        i += 1
    return res
```

```
m = np.array([[1, 2, 3, 4, 5], [1, 3, 2, 3, 10], [
    3, 2, 1, 4, 5], [6, 1, 1, 2, 2], [3, 2, 1, 5, 4]])
k = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]])
print(convNet(m, k))


im = Image.open('ex1.jpg')
rgb = np.array(im.convert('RGB'))
r = rgb[:, :, 0]  # array of R pixels

kernel1 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
kernel2 = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])

# # Image.fromarray(np.uint8(r)).show()
# Image.fromarray(np.uint8(convNet(r, kernel1))).show()
Image.fromarray(np.uint8(convNet(r, kernel2))).show()
```

week8.py

```
import sys
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
from sklearn.dummy import DummyClassifier
import matplotlib.pyplot as plt
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

plt.rc('font', size=10)
plt.rcParams['figure.constrained_layout.use'] = True

# 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()
n = 5000
x_train = x_train[1:n]
```

```python
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)

# 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()
    model.add(Conv2D(16, (3, 3), padding='same',
              input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), padding='same', 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)))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax',
              kernel_regularizer=regularizers.l1(0.0001)))
    model.compile(loss="categorical_crossentropy",
                  optimizer='adam', metrics=["accuracy"])
    model.summary()

    batch_size = 128
    epochs = 20
    history = model.fit(x_train, y_train, batch_size=batch_size,
                        epochs=epochs, validation_split=0.1)
    model.save("cifar.model")
    plt.subplot(211)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('model accuracy maxpool')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.subplot(212)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss maxpool')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], 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)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1, y_pred))

preds = model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1, y_pred))

dummy = DummyClassifier(strategy="most_frequent")
dummy.fit(x_train, y_train)
dummypred = dummy.predict(x_train)
y_pred = np.argmax(dummypred, axis=1)

print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1, y_pred))
```