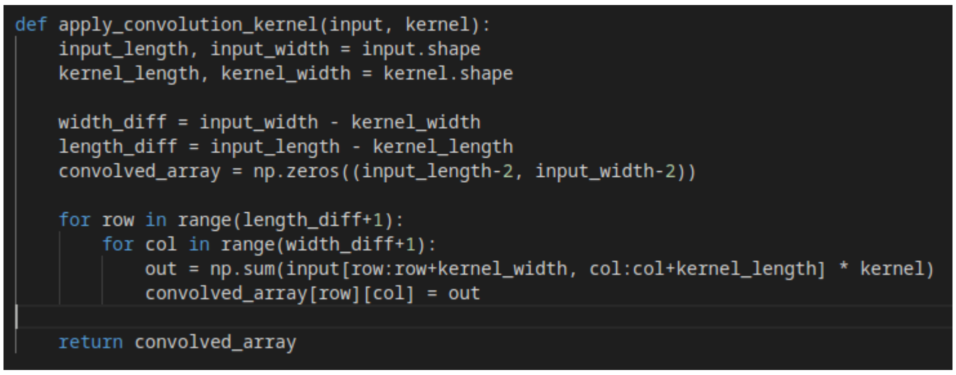
Vsevolod Syrtsov
18323202

QUESTION ONE.

(a).

Create function for applying any convolution kernel to input array.



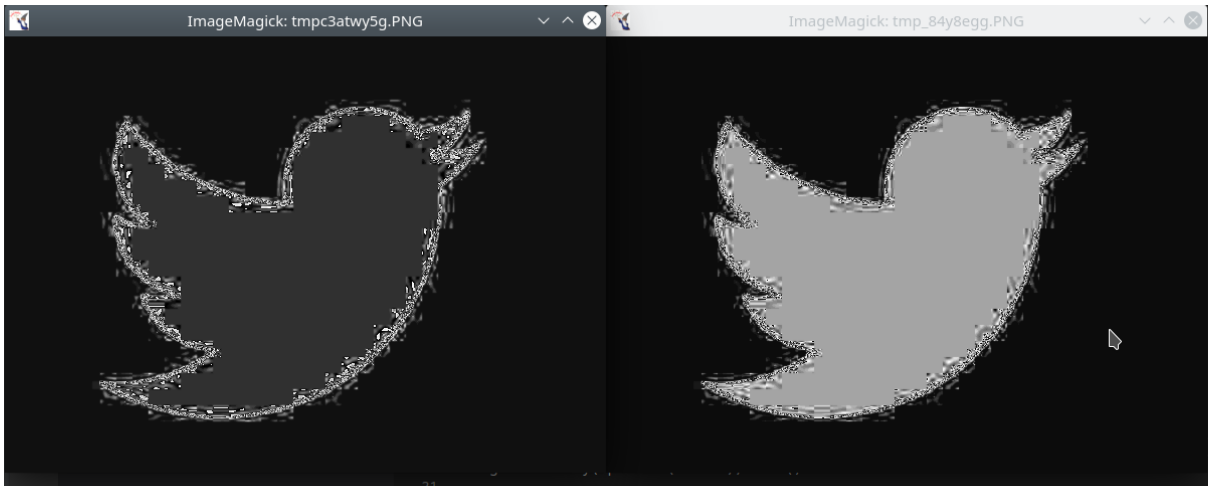
```python
def apply_convolution_kernel(input, kernel):
    input_length, input_width = input.shape
    kernel_length, kernel_width = kernel.shape

    width_diff = input_width - kernel_width
    length_diff = input_length - kernel_length
    convolved_array = np.zeros((input_length-2, input_width-2))

    for row in range(length_diff+1):
        for col in range(width_diff+1):
            out = np.sum(input[row:row+kernel_width, col:col+kernel_length] * kernel)
            convolved_array[row][col] = out

    return convolved_array
```

(b).



Above are shown the output images after applying the two provided convolution matrices on an image of the twitter bird.

QUESTION TWO.

(a).

The model used is a sequential one provided by the keras library, which can be modified by adding layers which pipe output from one layer to another. There are 4 convolution layers that are added to the model, with each respective layer having the following properties:

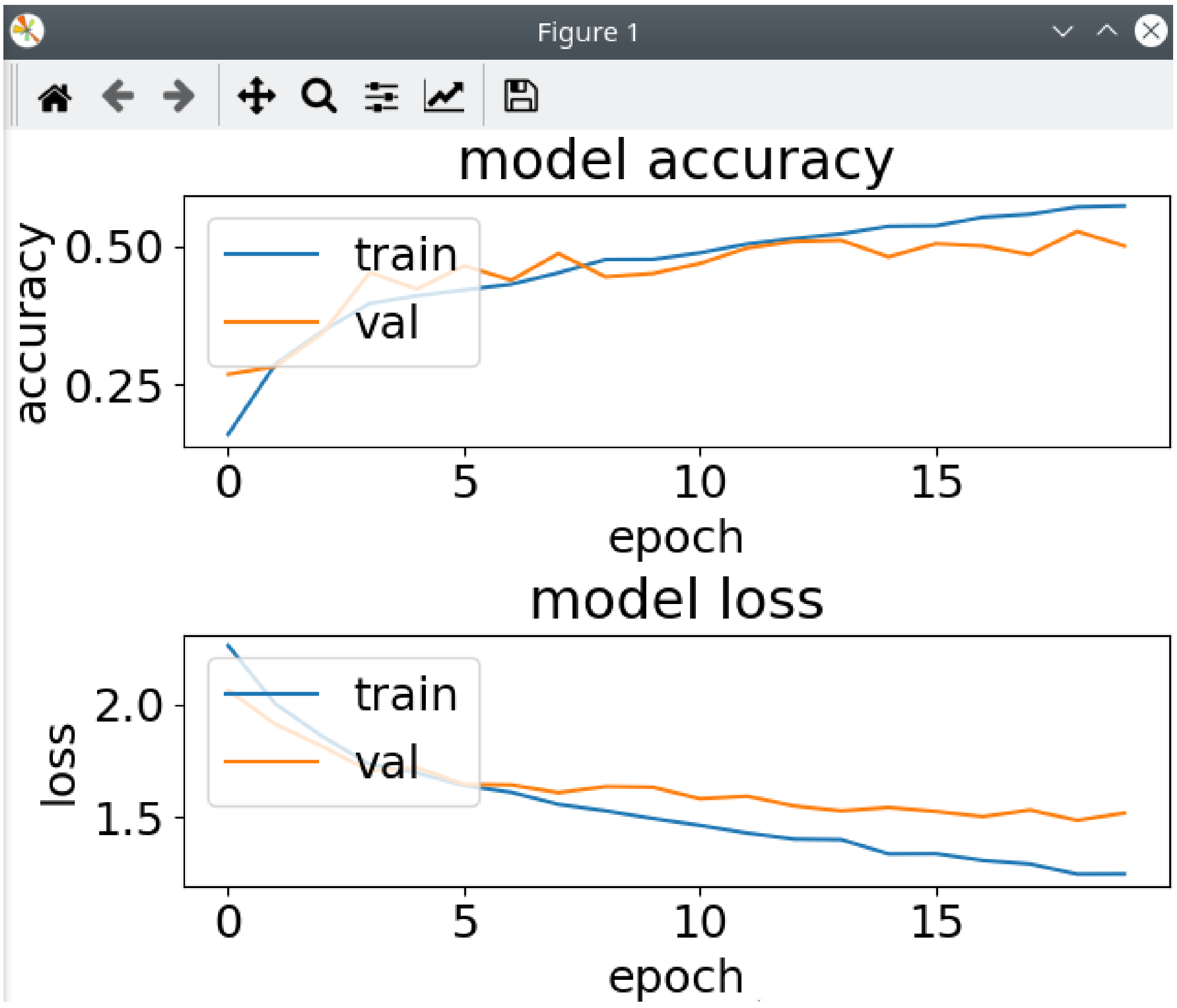
1. 16 convolution kernels of size 3x3 to apply to the input array. Input array padded so that the convolved matrix retains the input matrix's shape. First layer is parameterised so that it accepts the shape of the input array, which is a 32x32 image with 3 colour channels. The layer is also parameterised with a relu activation

function, which, explained in the lectures, is a highly efficient and popular method for convergence in neural networks.

2. The second layer applies another 16 convolution kernels of size 3x3 to the input matrix. In this layer the stride is altered from the default to jump by two spaces within the input matrix. Padding is applied to the input array so that the output once again retains the input array's shape.
3. The third layer performs the same function as the second, except with 32 kernels and using the default stride of 1 space.
4. The fourth layer performs the same function as the third, except with a stride of 2 spaces.

The model also contains a dropout layer which sets some input units to 0 and those not set to 0 scaled up by a rate defined by the function 1(1 - desired dropout rate); a flattening layer designed to reshape the dimensions of a tensor to be equal to the number of elements contained within the tensor; a final layer to map the outputs to their corresponding neuron.

(b).
Below is the plot for the accuracy and loss overtime in the model.



(i).

Keras says that this model has 37,146 total parameters, with the dense layer containing the most parameters. This layer contains the most parameters because it is the most used layer in the model. Comparing the results of the model's performance on the train and test data, the train data(0.60) performs better than the test data(0.48), though this can be attributed to the data being fitted better to the training data. Compared against a model that predicts the most common label, it definitely performs better, with the most frequent classifier only being successful 15.51% of the time.

(ii).

From observing the plot of accuracy and loss over time from the model, the validation accuracy begins to lag behind the accuracy of the model to predict the trained data. Over time this gulf, I can only assume, will increase as the model will be very well trained at predicting the class output for the training data, but not for generalisations.

(iii).

5K:
20s time to run.
Train Accuracy: 0.60
Test Accuracy: 0.48
10K:
The validation accuracy begins to close in on the training data accuracy.
40s time to run.
Train Accuracy: 0.65
Test Accuracy: 0.55
20K:
The validation accuracy and loss mimics the plot for the training data accuracy and loss.
65s time to run
Train Accuracy: 0.70
Test Accuracy: 0.63
40K:
Similar to 20K, the validation accuracy loss and accuracy mimics the plot for training accuracy and loss, this time with a little better prediction in earlier epochs.
140s time to run.
Train Accuracy: 0.74
Test Accuracy: 0.69

From observing the above data, I can conclude that the accuracy of the model does increase over time when using more data points, but does take more time to run.

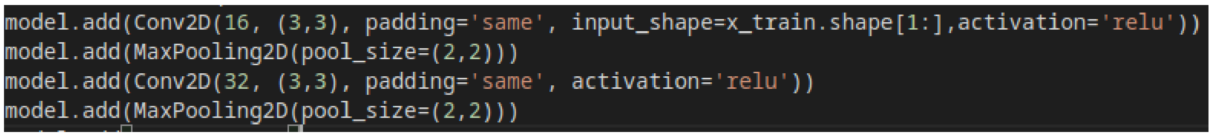
(iv).
For each L1 weight parameter in 0, 0.0001, 0.001, 0.01, 0.1, 1

L1	**Train Accuracy**	**Test Accuracy**
0	0.65	0.51
0.0001	0.60	0.48
0.001	0.53	0.46
0.01	0.44	0.43
0.1	0.31	0.32
1	0.24	0.23

It seems that changing the amount of datapoints is a more effective way of managing the issue of over-fitting, as changing the L1 weight parameter seems to negatively affect the accuracy of the model overall.

(c).

(i).



```
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(MaxPooling2D(pool_size=(2,2)))
```

Such is the modification to the model, as understood by me from the instructions given in the assignment.

(ii).

Keras reports that this model has 25,578 parameters associated with it.

Time taken with this new configuration to train the model on 5k datapoints comes in faster than with the strided layers, at **13.6s**. From my understanding the max-pool layer has less computation associated with it than a convolution layer, so chaining them together will result in less time to converge.

It performs slightly better on the accuracy values recorded in this document for the previous model, though those values lay in a **range** rather than being fixed values.

APPENDIX

This assignment was unconventional relative to the past assignments, and similarly the code that comes with it. The code changed a lot throughout, for ease of changing the weight parameters, layers, etc. without attempting to write code for subplots and risking waiting what I can assume to be at least over 20 minutes for running all the variations of the model in one file.

```python
import numpy as np
from PIL import Image

kernel_one = np.array([[-1,-1,-1],[-1,-8,-1],[-1,-1,-1]])
kernel_two = np.array([[0,-1,0],[-1,-8,-1],[0,-1,0]])

def apply_convolution_kernel(input, kernel):
   input_length, input_width = input.shape
   kernel_length, kernel_width = kernel.shape

   width_diff = input_width - kernel_width
   length_diff = input_length - kernel_length
   convolved_array = np.zeros((input_length-2, input_width-2))

   for row in range(length_diff+1):
       for col in range(width_diff+1):
           out = np.sum(input[row:row+kernel_width,
col:col+kernel_length] * kernel)
           convolved_array[row][col] = out

   return convolved_array


im = Image.open ('twitter.jpg')
rgb = np.array(im.convert('RGB'))
r=rgb[:,:,0]
out = apply_convolution_kernel(r, kernel_one)
Image.fromarray(np.uint8(out)).show()

result = apply_convolution_kernel(r, kernel_two)
Image.fromarray(np.uint8(out)).show()
```

```
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
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# 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]; 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(MaxPooling2D(pool_size=(2,2)))
   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')
   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')
   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))

# CALCULATE MOST FREQUENT CLASS PERFORMANCE
```

```
classes = {}
most_freq_class = None

for e in y_pred:
   if e not in classes:
       classes[e] = 0
   else:
       classes[e] = classes[e] + 1
   if most_freq_class != None:
       if classes[e] > classes[most_freq_class]:
           most_freq_class = e
   else:
       most_freq_class = e

print("Most frequent class classifier performance: [CLASS: {},
APPEARANCE:
{}]".format(most_freq_class,classes[most_freq_class]/len(y_pred)))
```