Assignment 8 – Deep Learning

Student number: 20324412

Module: CS7CS4/CSU44061

Part 1

Part a. Convolution Function

The function *convolve* as below was written which takes in an *n x n* array and a k x k kernel and returns the result of their convolution.

Function - convolve

```
def convolve(matrix, kernel, padding=0, strides=1):
    ''' Convolve a nxn matrix with a kxk kernel      '''

    #Get shapes
    xKernShape = kernel.shape[0]
    yKernShape = kernel.shape[1]
    xImgShape = matrix.shape[0]
    yImgShape = matrix.shape[0]

    # Shape of Output Convolution
    xOutput = int((xImgShape - xKernShape + 2) + 1)
    yOutput = int((yImgShape - yKernShape + 2) + 1)
    output = np.zeros((xOutput, yOutput))

    # Iterate through image
    for y in range(matrix.shape[1]):
        # Exit Convolution
        if y > matrix.shape[1] - yKernShape:
            break
        for x in range(matrix.shape[0]):
            # Go to next row once kernel is out of bounds
            if x > matrix.shape[0] - xKernShape:
                break
            output[x, y] = (kernel * matrix[x: x + xKernShape, y: y
+ yKernShape]).sum()

    return output
```

Apply Function

```
matrix = np.random.randint(1,8, size = (6,6))
kernel = np.random.randint(1,8, size = (3,3))
conv_output = convolve(matrix, kernel)
```

Part b. Application to image

The following image (Figure 1) was selected.

Figure 1. Sunflower Image (200 x 200)

When the image is convolved with kernel 1 as below, the output is as in Figure 2.

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

The following code was used to achieve this;

```
kernel1 = np.array([[-1,-1,1],[-1,8,-1],[-1,-1,-1]])
conv_output1 = convolve(r, kernel1)
conv_output1
Image.fromarray(np.uint(conv_output1)).show()
```

Figure 2. Convolution of the original image with kernel 1

When the image is convolved with kernel 2 as below, the output is as in Figure 3. The following code was used to achieve this;

```
#Apply function w/ kernel2

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

conv_output2 = convolve(r, kernel2)

#Display image

Image.fromarray(np.uint(conv_output2)).show()
```

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

Figure 3. Convolution of the original image with kernel 2

Part 2

Part a – Model Architecture

The model consists of four convolutional layers, one dropout layer and one final fully connected layer. The input is of size (32, 32, 3), i.e the image width is 32, the image height is 32 and there are three channels corresponding to the red (r), green (g) and blue (b) pixel intensities.

Layer 1

The first layer is as follows;

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

The output of the first layer is;

$a^{[1]} = g(w^{[1]} * a^{[0]} + b^{[1]})$

With kernel weights $w^{[1]}$, input $a^{[0]}$, bias/offset $b^{[1]}$ and activation function g which is a rectified linear unit in this case (i.e a 'relu'). The number of output filters in the convolution is 16. The kernel size, i.e the height and width of the 2D convolution window is 3x3. Padding is applied, i.e the original input is padded with additional zero entries before the kernel is applied. This results in the output being the same size as the input. The output is of size 32 x 32 x 16.

Layer 2

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

The number of output filters in the convolution is again 16 and the kernel size is again a 3 x 3. A stride of 2 is used in the convolution, i.e the kernel is moved along by two at each step. Increasing the stride reduces the size of the output matrix. A rectified linear unit is used as the activation function. Padding is again applied.

Layer 3

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

The number of output filters in the convolution is 32 and the kernel size is a 3 x 3. A rectified linear unit is used as the activation function. Padding is again applied.

Layer 4

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

The number of output filters in the convolution is 32 and the kernel size is again a 3x3. A rectified linear unit is used as the activation function. . A stride of 2 is used in the convolution, i.e the kernel is moved along by two at each step. Padding is again applied.

Dropout Layer

A dropout layer is included after the fourth convolutional layers. The Dropout layer randomly sets input units to 0 at each step during training time at a frequency specified by rate, 0.5 in this case. It is implemented to help prevent overfitting.

Final layer – Fully connected layers

```
model.add(Dense(num_classes,
activation='softmax',kernel_regularizer=regularizers.l1(0.0001)))
```

The final layer is a fully connected layer, i.e a dense layer in keras. It gives the final probabilities for each class label, which is 10 classes in this case. The non-linear function applied is the softmax function, which is equivalent to a multi-class logistic regression model.

Part b

Part i. Model parameters & performance

- The model has 37,146 parameters.
- The fourth layer has the most parameters, with 9248 parameters. This is because the number of channels increases as one moves deeper into the network, while the size of the image decreases. The number of parameters can be determined as follows;
  $Number\ of\ parameters$
  $= Num\_output\_channels$ x ($Num\ of\ output\ channels$ x kernel width x kenrel height $+ 1$)
- This corresponds to 32*(32 x 3 x 3 + 1) = 9248 parameters.

Performance on training & test data

The performance of the model on the training and test set are shown below in table 1 and table 2 respectfully. As expected the model performs significantly better on the training set then on the test set. This is because the model was fit to the training set whereas the test set is unseen data. The f1-score is a good indication of model performance as it is the harmonic mean of the precision and recall. On the training set, the f1 score ranges from 50% for class 3 up to 76% for class 1. While on the test set, the f1 score ranges from 29% for class 3 up to 65% for class 1. The overall average f1 score on the training set is 64% while it is 51% in the case of the training set, a reduction of 13%.

	precision	recall	f1-score	support
0	0.70	0.65	0.67	505
1	0.75	0.76	0.76	460
2	0.55	0.52	0.54	519
3	0.65	0.41	0.50	486
4	0.48	0.65	0.56	519
5	0.65	0.54	0.59	488
6	0.59	0.77	0.67	518
7	0.68	0.67	0.68	486
8	0.75	0.69	0.72	520
9	0.70	0.77	0.73	498
accuracy			0.64	4999
macro avg	0.65	0.64	0.64	4999
weighted avg	0.65	0.64	0.64	4999

Table 1. Performance on training data

	precision	recall	f1-score	support
0	0.59	0.55	0.57	1000
1	0.65	0.65	0.65	1000
2	0.40	0.38	0.39	1000
3	0.39	0.23	0.29	1000
4	0.37	0.49	0.42	1000
5	0.45	0.35	0.39	1000
6	0.48	0.67	0.56	1000
7	0.57	0.56	0.56	1000
8	0.62	0.59	0.61	1000
9	0.57	0.62	0.59	1000
accuracy			0.51	10000
macro avg	0.51	0.51	0.50	10000
weighted avg	0.51	0.51	0.50	10000

Table 2. Performance on test data

Baseline

The baseline model employed is one that always predicts the most common class, which is class 8 in this case. The performance of the model on the training and test set are shown below in table 3 and table 4 respectfully. As expected the model behaves poorly as it is only able to discriminate one of the ten classes. The resultant accuracy is thus 10%.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	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	0.10	1.00	0.19	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

Table 3. Baseline model results on training set

	precision	recall	f1-score	support
0	0.00	0.00	0.00	1000
1	0.00	0.00	0.00	1000
2	0.00	0.00	0.00	1000
3	0.00	0.00	0.00	1000
4	0.00	0.00	0.00	1000
5	0.00	0.00	0.00	1000
6	0.00	0.00	0.00	1000
7	0.00	0.00	0.00	1000
8	0.10	1.00	0.18	1000
9	0.00	0.00	0.00	1000
accuracy			0.10	10000
macro avg	0.01	0.10	0.02	10000
weighted avg	0.01	0.10	0.02	10000

Table 4. Baseline model results on test set

Part ii. History plot – Diagnostics

The plot of the model loss and model accuracy on the training and validation set is shown in Figure 4. From the plot, it can be deduced the point at which the model starts to overfit to the training data. To begin with, the model accuracy is very low while the loss is high and the model is underfitting the training data. After about 7 or 8 epochs of training the results on the training data begin to steadily outperform that of the validation data. From this point on the training accuracy never falls below that of the validation accuracy, while the training remains significantly less than that of the validation loss. This is a sign that the model is overfitting to the training data. It is fitting to the noise or the inherent idiosyncranices of the training data and so is unable to generalise well to new unseen data such as the validation set.

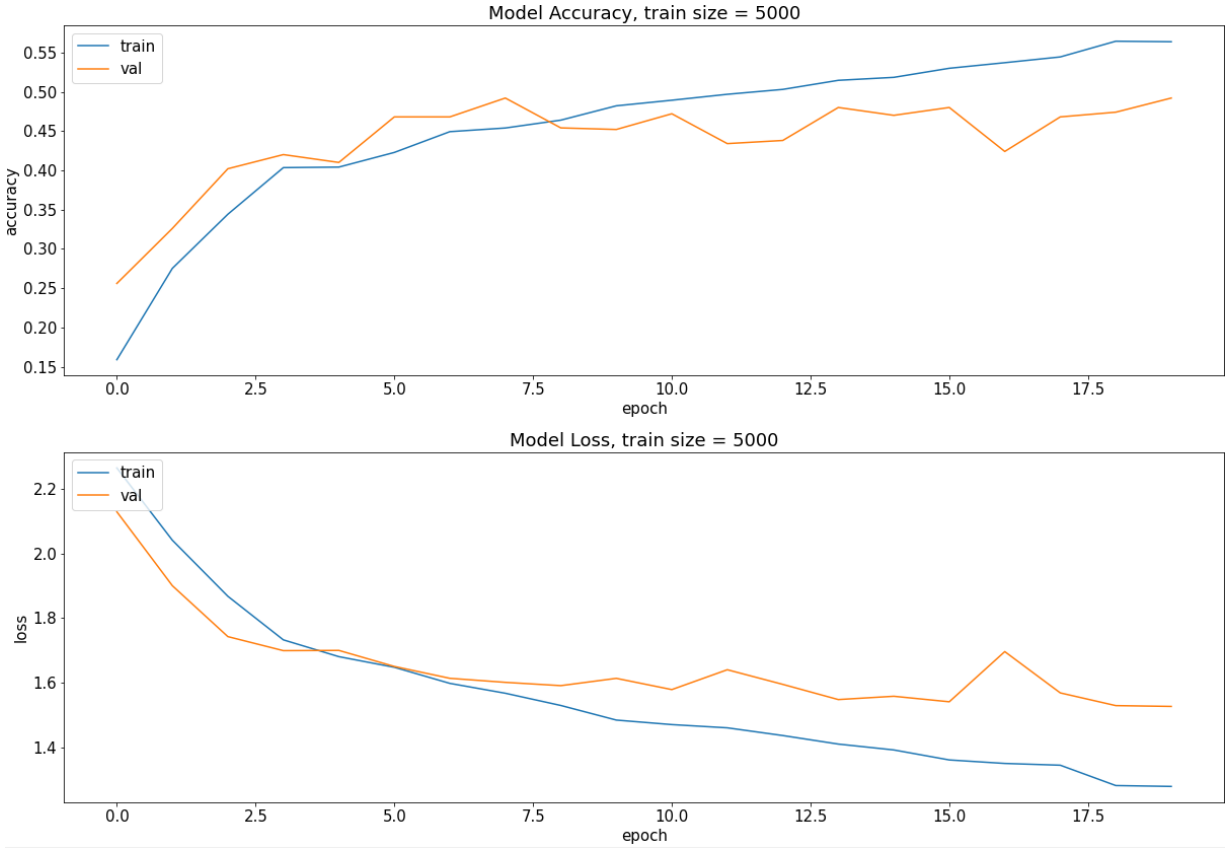

Figure 4. Model accuracy and loss across training epochs

Part iii. Effect of Training Size

The effect of the training size on model accuracy, model loss and training time was investigated. This was achieved through the function *model_train_sizes* as below. The function loops through the various train sizes (5k, 10, 20k, 40k), fits the model to the data and determines the resultant train accuracy, test accuracy and training time. The function returns a dataframe of results as shown in table 5. Both the training and test accuracy increase with increased training size. For small training size (e.g. n = 5k), the model has only been exposed to a small portion of the data and so it fails to generalise well to unseen data. The model may have been overfit to the training set and so performs poorly on unseen data, achieving a test accuracy of only 49.09%, worse than chance. As the training size is increased up to 40k, the model achieves a test accuracy of 68.16%, an increase of nearly 20%. It is unsurprising that an increased training size improves model performance. More samples gives the algorithm more opportunity to understand the underlying mapping of inputs to outputs, thus producing a better performing model.

Considering the history plot as shown in figure 5, similar results are seen. Across all iterations, the training accuracy is higher than the validation accuracy and the training loss is less than the validation loss, as expected. As the training size is increased the accuracy increases and the loss decreases, with the best model performance found for a training size of 40k as previously described.

The training time increases with the training size, increasing from 1 minute for a training size of 5k to almost 10 minutes for a training size of 40k as shown in table 5. This is expected given that, for increased training size, the model has to iterate through the entire training set for all 20 epochs and so a greater training size would result in a longer computation time.

Training size	Training time	Training accuracy	Test accuracy
5k	1 min, 15.5 secs	62.27%	49.09%
10k	2 min, 26.3 sec	71.12%	56.49%
20k	4 min, 57.7 secs	73.35%	62.51%
40k	9 min, 50.9 secs	74.00%	68.16%

Table 5. Results of varying the training size.

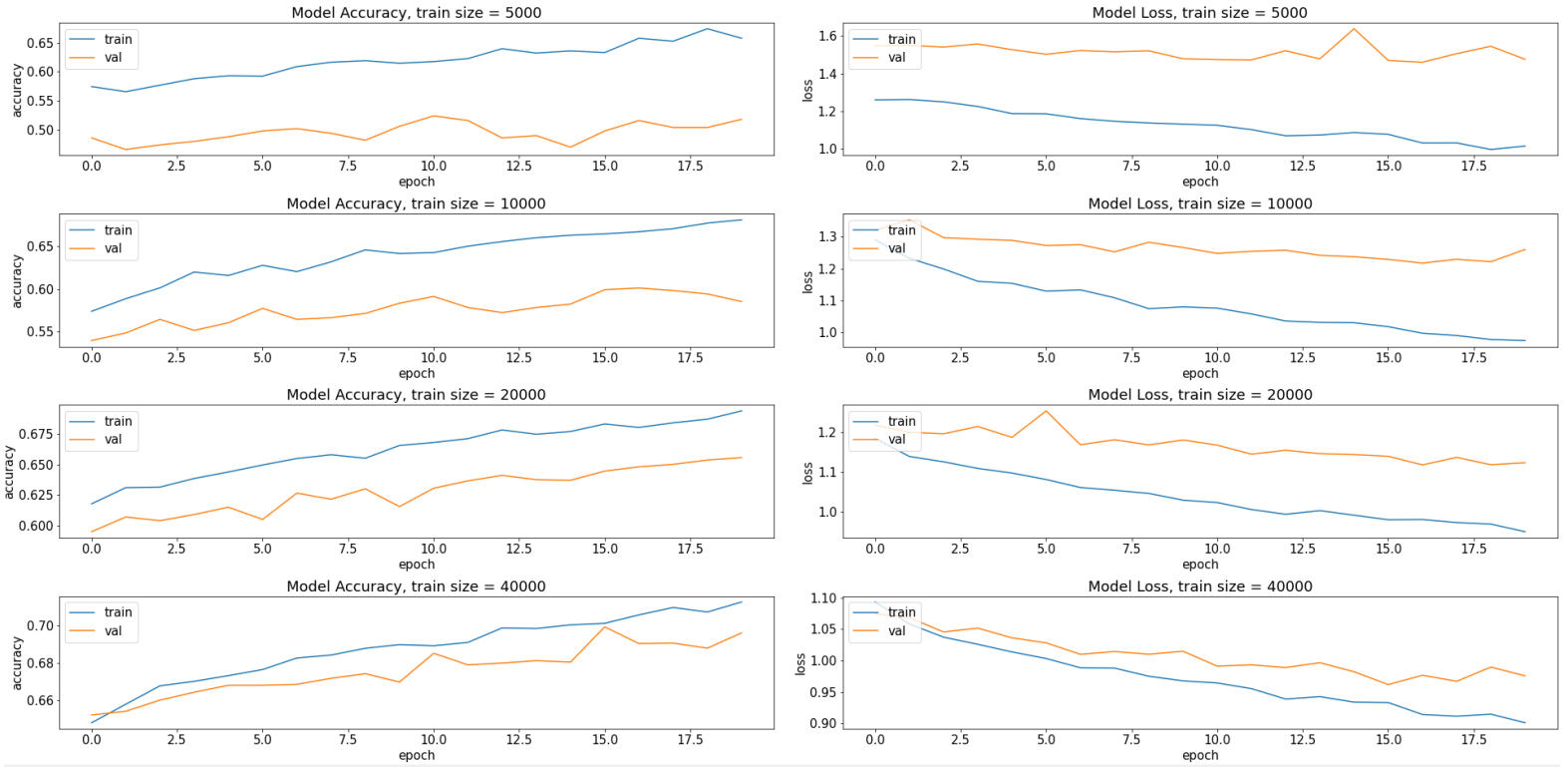

Figure. 5. Model accuracy and model loss on training and validation set for varying the train size in the range [5k, 10k, 20k, 40k]

Function model_train_sizes

```
def model_train_sizes(x_train_orig, y_train_orig, x_test, y_test,
train_size, batch_size, epochs):

    #Figure
    fig, axs = plt.subplots(4, 2)
    results = []
    count = 0

    for n in train_size:
        'Perform training for each size of n'
        #Split data
        x_train = x_train_orig[1:n]; y_train = y_train_orig[1:n]

        #Format data
        x_train = x_train.astype("float32") / 255
        #Convert class vectors to binary class matrices
        y_train = keras.utils.to_categorical(y_train, num_classes)

        #Train model
        start = time.time()
        history = model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)
        end = time.time()
        timeX = (end - start)
        print('time train = {}'.format(timeX))
```

```
        #Model prediction results
        y_preds_train = model.predict(x_train)
        y_preds_train = np.argmax(y_preds_train, axis=1) #Highest prob
        #Format
        y_train1 = np.argmax(y_train, axis=1)
        train_accuracy = accuracy_score(y_train1, y_preds_train)
        print('train accuracy = {}'.format(train_accuracy))

        #Model prediction results
        y_preds_test = model.predict(x_test)
        y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob
        #Format
        y_test1 = np.argmax(y_test, axis=1)
        test_accuracy = accuracy_score(y_test1, y_preds_test)
        print('test accuracy = {}'.format(test_accuracy))

        #Results
        d = {
            'training size' : n,
        'train time': timeX,
        'train accuracy' :  train_accuracy,
        'test accuracy' :  test_accuracy,
        }
        results.append(d)

        #Plots
        #Accuracy
        axs[count, 0].plot(history.history['accuracy'])
        axs[count, 0].plot(history.history['val_accuracy'])
        axs[count, 0].set_title('Model Accuracy, train size =
{}'.format(n))
        axs[count, 0].set(xlabel='epoch', ylabel='accuracy')
        axs[count, 0].legend(['train', 'val'], loc='upper left')
        #Loss
        axs[count, 1].plot(history.history['loss'])
        axs[count, 1].plot(history.history['val_loss'])
        axs[count, 1].set_title('Model Loss, train size = {}'.format(n))
        axs[count, 1].set(xlabel='epoch', ylabel='loss')
        axs[count, 1].legend(['train', 'val'], loc='upper left')
        #Update count
        count = count + 1

    #Show plot
    plt.show()
    #Dataframe of results
    df_results = pd.DataFrame(results)

    return df_results

#Apply Function
train_size = [5000, 10000, 20000, 40000]
df_results = model_train_sizes(x_train_orig, y_train_orig, x_test, y_test,
train_size, batch_size, epochs)
```

Part iv. Effect of varying L1 weight parameter

The effect of varying the L1 weight parameter on model accuracy, model loss and training time was investigated. This was achieved through the function *model_train_sizes* as below. The model was trained across the following range of weights; [0.0001, 0.01, 0.1, 1, 10, 50, 100, 1000]. L1 regularization penalizes weight values that are close to 0 by making them equal to 0. It has a significant effect on the accuracy. As it is increased both the training and test accuracy decrease significantly, from 60.6% and 48.3% respectfully for a weight of 0.0001 to 10.10% and 9.25% for a weight of 1000. This is also seen in Figure 6. The loss also increases dramatically as the weight is increased. Evidently, as the L1 weight is increased, more model parameters are set to zero and the model fit becomes increasingly worse.

L1 weight	train accuracy	test accuracy
0.0001	0.6065	0.4836
0.01	0.4688	0.4255
0.1	0.3250	0.3169
1.0	0.1932	0.1844
10.0	0.1048	0.1008
50.0	0.1044	0.1002
100.0	0.1038	0.1000
1000.0	0.1010	0.0925

Table 6. Effect of L1 Regularisation weight parameter on training and test accuracy

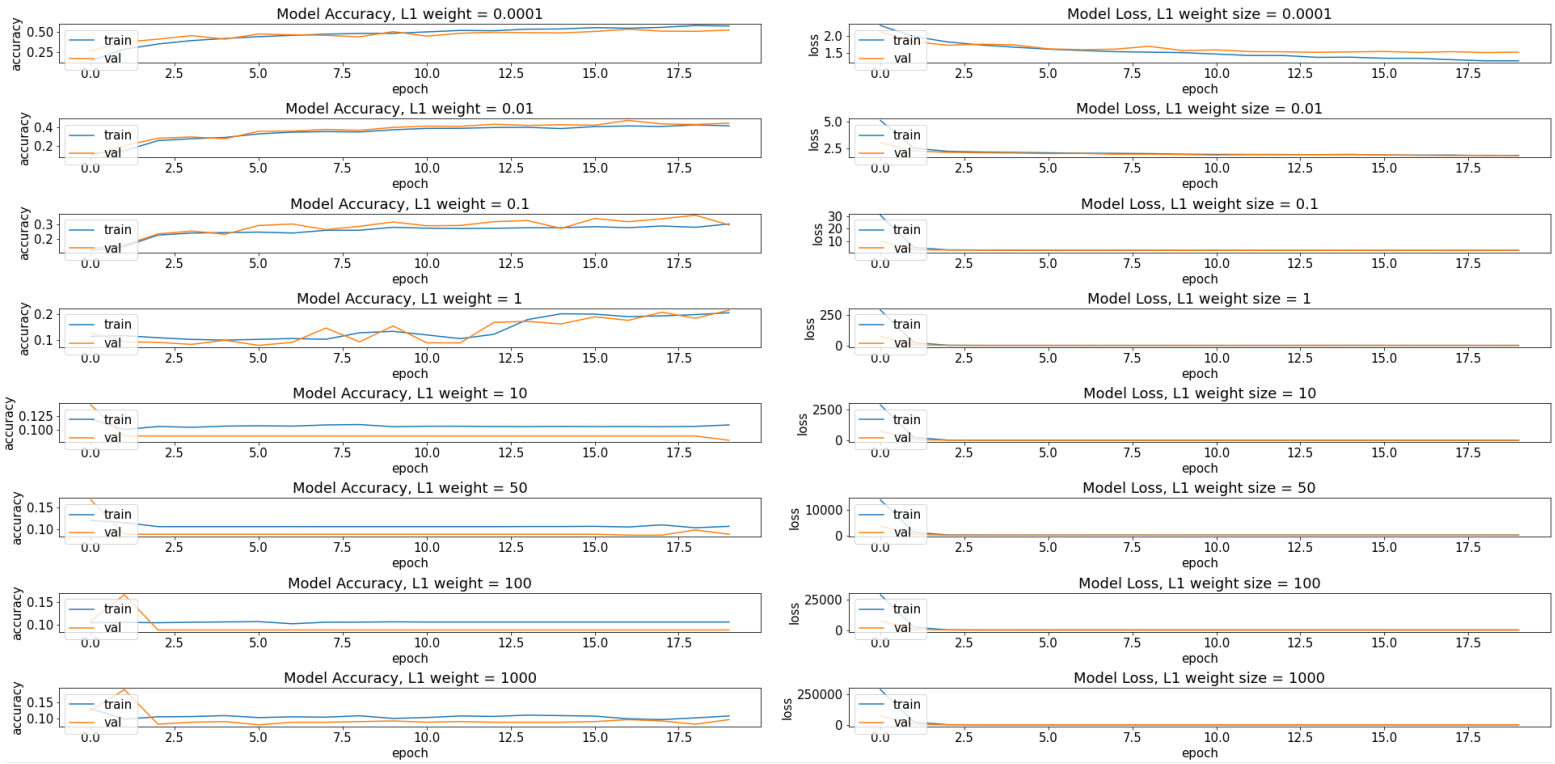

Figure 6. Effect of varying L1 weight on model accuracy and model loss for L1 weight in the range [0.0001, 0.01, 0.1, 1, 10, 50, 100, 1000]

Function model_vary_L1

```
def model_vary_L1(x_train, y_train, x_test, y_test, list_l1):
    'Train model for varying L1'
    #Figure
    fig, axs = plt.subplots(8, 2)
    count = 0
    results = []

    for l1X in list_l1:
        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), 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(l1X)))
        model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
        #model.summary()

        #Training
        history = model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)

        #Model prediction results
        y_preds_train = model.predict(x_train)
        y_preds_train = np.argmax(y_preds_train, axis=1) #Highest prob
        #Format
        y_train1 = np.argmax(y_train, axis=1)
        train_accuracy = accuracy_score(y_train1, y_preds_train)

        #Model prediction results
```

```
        y_preds_test = model.predict(x_test)
        y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob
        #Format
        y_test1 = np.argmax(y_test, axis=1)
        test_accuracy = accuracy_score(y_test1, y_preds_test)

        #Results
        d = {
            'L1 weight' : l1X,
        'train accuracy' :  train_accuracy,
        'test accuracy' :  test_accuracy,

        }
        results.append(d)

        #Plots
        axs[count, 0].plot(history.history['accuracy'])
        axs[count, 0].plot(history.history['val_accuracy'])
        axs[count, 0].set_title('Model Accuracy, L1 weight =
{}'.format(l1X))
        axs[count, 0].set(xlabel='epoch', ylabel='accuracy')
        axs[count, 0].legend(['train', 'val'], loc='upper left')
        #Loss
        axs[count, 1].plot(history.history['loss'])
        axs[count, 1].plot(history.history['val_loss'])
        axs[count, 1].set_title('Model Loss, L1 weight size =
{}'.format(l1X))
        axs[count, 1].set(xlabel='epoch', ylabel='loss')
        axs[count, 1].legend(['train', 'val'], loc='upper left')
        #Update count
        count = count + 1

    #Dataframe of results
    df_results = pd.DataFrame(results)
    #Show plot
```

```
    plt.show()


    return df_results
```

Apply function

```
list_l1 = [0.0001, 0.01, 0.1, 1, 10, 50, 100, 1000]

df_results = model_vary_L1(x_train, y_train, x_test, y_test, list_l1)
```

Part c. Model with Max pooling

The model architecture using max pooling instead of a stride to downsample is as follows;

```
model = keras.Sequential()
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'))
model.add(Dropout (0.5))
model.add(Flatten( ))
model.add(Dense(num_classes, activation = 'softmax', kernel_regularizer =
   regularizers.l1(0.0001)))
```

Model performance

The model has 37,146 parameters. The new model architecture which uses maxpooling to downsample is slightly better than that of the original architecture achieving a training accuracy of 64.11% and a test accuracy of 54.15%, an improvement of 5% on the first model. There was no difference in training time between the two models, both models took 1 min, 15 seconds to train when 5k samples were used.

Max pooling architecture

Training size	Training time	Training accuracy	Test accuracy
5k	1 min, 15 secs	64.11%	54.15%

Table 7.

Original architecture with strides

Training size	Training time	Training accuracy	Test accuracy
5k	1 min, 15.5 secs	62.27%	49.09%

Table 8.

Part d – Thinner, Deeper Network

The thinner and deeper model architecture was investigated. The model has 23,734 learnable parameters and the architecture summary is as follows;

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_88 (Conv2D)           (None, 32, 32, 8)         224
_________________________________________________________________
conv2d_89 (Conv2D)           (None, 16, 16, 8)         584
_________________________________________________________________
conv2d_90 (Conv2D)           (None, 16, 16, 16)        1168
_________________________________________________________________
conv2d_91 (Conv2D)           (None, 8, 8, 16)          2320
_________________________________________________________________
conv2d_92 (Conv2D)           (None, 8, 8, 32)          4640
_________________________________________________________________
conv2d_93 (Conv2D)           (None, 4, 4, 32)          9248
_________________________________________________________________
dropout_22 (Dropout)         (None, 4, 4, 32)          0
_________________________________________________________________
flatten_22 (Flatten)         (None, 512)               0
_________________________________________________________________
dense_22 (Dense)             (None, 10)                5130
=================================================================
Total params: 23,314
Trainable params: 23,314
Non-trainable params: 0
_________________________________________________________________
```

Model summary

To determine the effect adding more layers has on model performance, the function *model_train_sizes* was again used. The output is shown in table 9 and the resultant plots are shown in figure 7. This thinner, deeper model has a higher prediction accuracy and shorter training time than that of the original architecture, across all training sizes. Given this, it appears to be a superior model. The test accuracy is approximately 70% for all training examples. Again, for increased training size, the training time increases.

Training size	Training time	Training accuracy	Test accuracy
5k	52 secs	78.6 %	70.4 %
10k	2 min, 7 sec	77.9 %	70.5 %
20k	3 min, 4 secs	77.7 %	70.7 %
40k	6 min, 39 secs	76.7%	70.8 %

Table 9. Results of varying the training size.

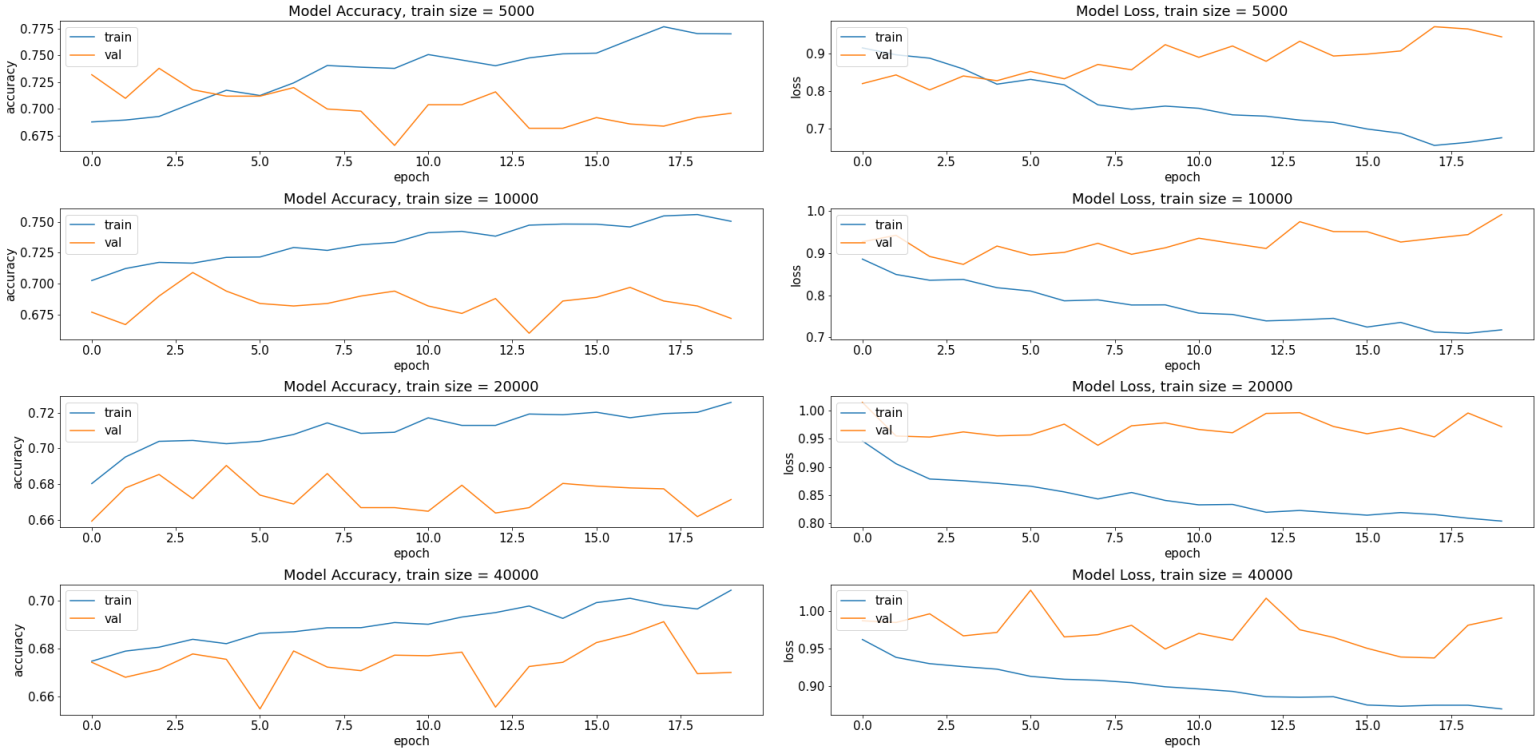

Figure 7. Accuracy and loss plots for the thinner, deeper architect across training sizes of size [5k, 10k, 20k, 40k]

Appendix – Code

```
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 30 22:06:37 2020

@author: Hannah Craddock
"""
#Imports
from PIL import Image
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from tensorflow.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
from sklearn.dummy import DummyClassifier
import time
from sklearn.metrics import accuracy_score

%matplotlib qt
plt.rc('font', size=15);

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

#*****************************************
```

```
#Part 1.


#Part a - Convolve Function
def convolve(matrix, kernel):
    ''' Convolve a nxn matrix with a kxk kernel      '''

    #Get shapes
    xKernShape = kernel.shape[0]
    yKernShape = kernel.shape[1]
    xImgShape = matrix.shape[0]
    yImgShape = matrix.shape[0]

    # Shape of Output Convolution
    xOutput = int((xImgShape - xKernShape + 2) + 1)
    yOutput = int((yImgShape - yKernShape + 2) + 1)
    output = np.zeros((xOutput, yOutput))

    # Iterate through image
    for y in range(matrix.shape[1]):
        # Exit Convolution
        if y > matrix.shape[1] - yKernShape:
            break
        for x in range(matrix.shape[0]):
            # Go to next row once kernel is out of bounds
            if x > matrix.shape[0] - xKernShape:
                break
            output[x, y] = (kernel * matrix[x: x + xKernShape, y: y +
yKernShape]).sum()

    return output

#Apply Function
matrix = np.random.randint(1,8, size = (6,6))
kernel = np.random.randint(1,8, size = (3,3))
conv_output = convolve(matrix, kernel)
```

```
#************************
#Part b
im = Image.open('sunflower.jpg')
im = im.resize((200,200))
rgb = np.array(im.convert('RGB'))
r = rgb[:,:,0] # array of R pixels
#Image.fromarray(np.uint(r)).show()


#Apply function w/ kernel1
kernel1 = np.array([[-1,-1,1],[-1,8,-1],[-1,-1,-1]])
conv_output1 = convolve(r, kernel1)
conv_output1
Image.fromarray(np.uint(conv_output1)).show()


#Apply function w/ kernel2
kernel2 = np.array([[0,-1,0],[-1,8,-1],[0,-1,0]])
conv_output2 = convolve(r, kernel2)
#Display image
Image.fromarray(np.uint(conv_output2)).show()


#************************************************************************
************************
#Part 2

#Data - split between train and test sets
(x_train_orig, y_train_orig), (x_test, y_test) =
keras.datasets.cifar10.load_data()

#Split data
n=5000
```

```
x_train = x_train_orig[1:n]; y_train = y_train_orig[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)

#Train model
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), 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(0.0001)))
    model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
    model.summary()

    #Training
    #Training params
    batch_size = 128
```

```
    epochs = 20


    #Time it
    start = time.time()
    history = model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)
    end = time.time()
    print(end - start)
    #Save model
    model.save("cifar.model")


    #Plots
    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()


#Prediction
#Training set
preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1) #Highest prob
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))
train_accuracy = accuracy_score(y_train1, y_pred)
```

```
#Test set
y_preds_test = model.predict(x_test)
y_preds_test = np.argmax(y_preds_test, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_preds_test))
print(confusion_matrix(y_test1,y_preds_test))
test_accuracy = accuracy_score(y_test1, y_preds_test)

#Accuracy
#Model prediction results
y_preds_test = model.predict(x_test)
y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob
#Format
y_test1 = np.argmax(y_test, axis=1)
test_accuracy = accuracy_score(y_test1, y_preds_test)

#*******************************************
#Part b.ii. Baseline Model
dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(x_train, y_train)

#Predict on training set
predictions_dummy = dummy_clf.predict(x_train)
y_pred_dummy = np.argmax(predictions_dummy, axis=1)

print(classification_report(y_train1, y_pred_dummy))
print(confusion_matrix(y_train1,y_pred_dummy))

#Baseline v2
y_pred_bl = 8*np.ones(len(y_train1))

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

```
#Test set
#Baseline v2
y_pred_bl = 8*np.ones(len(y_test1))

print(classification_report(y_test1, y_pred_bl))
print(confusion_matrix(y_test1, y_pred_bl))

#*********************************************************
#Part iii. Timing

def model_train_sizes(x_train_orig, y_train_orig, x_test, y_test,
train_size, batch_size, epochs):

    #Figure
    fig, axs = plt.subplots(4, 2)
    results = []
    count = 0

    for n in train_size:
        'Perform training for each size of n'
        #Split data
        x_train = x_train_orig[1:n]; y_train = y_train_orig[1:n]

        #Format data
        x_train = x_train.astype("float32") / 255
        #Convert class vectors to binary class matrices
        y_train = keras.utils.to_categorical(y_train, num_classes)

        #Train model
        start = time.time()
        history = model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)
        end = time.time()
        timeX = (end - start)
```

```
        print('time train = {}'.format(timeX))

        #Model prediction results
        y_preds_train = model.predict(x_train)
        y_preds_train = np.argmax(y_preds_train, axis=1) #Highest prob
        #Format
        y_train1 = np.argmax(y_train, axis=1)
        train_accuracy = accuracy_score(y_train1, y_preds_train)
        print('train accuracy = {}'.format(train_accuracy))

        #Model prediction results
        y_preds_test = model.predict(x_test)
        y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob
        #Format
        y_test1 = np.argmax(y_test, axis=1)
        test_accuracy = accuracy_score(y_test1, y_preds_test)
        print('test accuracy = {}'.format(test_accuracy))

        #Results
        d = {
            'training size' : n,
        'train time': timeX,
        'train accuracy' :  train_accuracy,
        'test accuracy' :  test_accuracy,
        }
        results.append(d)

        #Plots
        #Accuracy
        axs[count, 0].plot(history.history['accuracy'])
        axs[count, 0].plot(history.history['val_accuracy'])
        axs[count, 0].set_title('Model Accuracy, train size =
{}'.format(n))
        axs[count, 0].set(xlabel='epoch', ylabel='accuracy')
        axs[count, 0].legend(['train', 'val'], loc='upper left')
```

```
        #Loss
        axs[count, 1].plot(history.history['loss'])
        axs[count, 1].plot(history.history['val_loss'])
        axs[count, 1].set_title('Model Loss, train size = {}'.format(n))
        axs[count, 1].set(xlabel='epoch', ylabel='loss')
        axs[count, 1].legend(['train', 'val'], loc='upper left')
        #Update count
        count = count + 1

    #Show plot
    plt.show()
    #Dataframe of results
    df_results = pd.DataFrame(results)

    return df_results

#Apply
train_size = [5000, 10000, 20000, 40000]
df_results3 = model_train_sizes(x_train_orig, y_train_orig, x_test, y_test,
train_size, batch_size, epochs)



#***********************************************
#Part iv. Inspect varying L1 reguliser

def model_vary_L1(x_train, y_train, x_test, y_test, list_l1):
    'Train model for varying L1'
    #Figure
    fig, axs = plt.subplots(8, 2)
    count = 0
    results = []

    for l1X in list_l1:
        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), 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(l1X)))

        model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])

        #model.summary()


        #Training

        history = model.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)


        #Model prediction results

        y_preds_train = model.predict(x_train)

        y_preds_train = np.argmax(y_preds_train, axis=1) #Highest prob

        #Format

        y_train1 = np.argmax(y_train, axis=1)

        train_accuracy = accuracy_score(y_train1, y_preds_train)


        #Model prediction results

        y_preds_test = model.predict(x_test)

        y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob

        #Format

        y_test1 = np.argmax(y_test, axis=1)

        test_accuracy = accuracy_score(y_test1, y_preds_test)


        #Results

        d = {

            'L1 weight' : l1X,

        'train accuracy' :  train_accuracy,
```

```
        'test accuracy' :  test_accuracy,


        }
        results.append(d)


        #Plots
        axs[count, 0].plot(history.history['accuracy'])
        axs[count, 0].plot(history.history['val_accuracy'])
        axs[count, 0].set_title('Model Accuracy, L1 weight =
{}'.format(l1X))
        axs[count, 0].set(xlabel='epoch', ylabel='accuracy')
        axs[count, 0].legend(['train', 'val'], loc='upper left')
        #Loss
        axs[count, 1].plot(history.history['loss'])
        axs[count, 1].plot(history.history['val_loss'])
        axs[count, 1].set_title('Model Loss, L1 weight size =
{}'.format(l1X))
        axs[count, 1].set(xlabel='epoch', ylabel='loss')
        axs[count, 1].legend(['train', 'val'], loc='upper left')
        #Update count
        count = count + 1


    #Dataframe of results
    df_results = pd.DataFrame(results)
    #Show plot
    plt.show()


    return df_results


#Apply function
list_l1 = [0.0001, 0.01, 0.1, 1, 10, 50, 100, 1000]
df_results2 = model_vary_L1(x_train, y_train, x_test, y_test, list_l1)


#*************************************************************************
**
#Part c - Model
```

```
#Train model

use_saved_model = False

if use_saved_model:

    model2 = keras.models.load_model("cifar.model")

else:

    model2 = keras.Sequential()

    model2.add(Conv2D(16, (3,3), padding='same', input_shape =
x_train.shape[1:], activation='relu'))

    model2.add(Conv2D(16, (3,3), padding='same', activation='relu'))

    model2.add(MaxPooling2D(pool_size=(2, 2)))

    model2.add(Conv2D(32, (3,3), padding='same', activation='relu'))

    model2.add(Conv2D(32, (3,3), padding='same', activation='relu'))

    model2.add(MaxPooling2D(pool_size=(2, 2)))

    model2.add(Dropout(0.5))

    model2.add(Flatten())

    model2.add(Dense(num_classes, activation='softmax',
kernel_regularizer=regularizers.l1(0.0001)))

    model2.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])

    model2.summary()


    #Training

    #Training params

    batch_size = 128

    epochs = 20


    #Time it

    start = time.time()

    history = model2.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)

    end = time.time()

    print(end - start)

    #Save model

    model2.save("cifar2.model")
```

```
#Performance
#Plots
fig, axs = plt.subplots(2, 1)
#Accuracy
axs[0].plot(history.history['accuracy'])
axs[0].plot(history.history['val_accuracy'])
axs[0].set_title('Model Accuracy, train size = {}'.format(n))
axs[0].set(xlabel='epoch', ylabel='accuracy')
axs[0].legend(['train', 'val'], loc='upper left')
#Loss
axs[1].plot(history.history['loss'])
axs[1].plot(history.history['val_loss'])
axs[1].set_title('Model Loss, train size = {}'.format(n))
axs[1].set(xlabel='epoch', ylabel='loss')
axs[1].legend(['train', 'val'], loc='upper left')

#**************************
#Prediction
#Training set
preds = model2.predict(x_train)
y_preds_train = np.argmax(preds, axis=1) #Highest prob
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))
train_accuracy = accuracy_score(y_train1, y_preds_train)

#Test set
y_preds_test = model2.predict(x_test)
y_preds_test = np.argmax(y_preds_test, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_preds_test))
print(confusion_matrix(y_test1,y_preds_test))
test_accuracy = accuracy_score(y_test1, y_preds_test)
test_accuracy
```

```
#*************************************************************************
********************

#Model Part d


model3 = keras.Sequential()

model3.add(Conv2D(8, (3, 3), padding= 'same', input_shape= x_train.shape[1:
], activation = 'relu' ) )

model3.add(Conv2D(8, (3, 3), strides = (2,2) , padding= 'same', activation
= 'relu' ) )

model3.add(Conv2D(16, (3, 3), padding= 'same',  activation = 'relu') )

model3.add(Conv2D(16, (3, 3), strides = (2, 2) , padding= 'same',
activation = 'relu' ) )

model3.add(Conv2D(32, (3, 3), padding= 'same', activation = 'relu') )

model3.add(Conv2D(32, (3, 3), strides = (2,2) , padding= 'same', activation
= 'relu'))

model3.add(Dropout (0.5))

model3.add(Flatten( ))

model3.add(Dense(num_classes, activation = 'softmax', kernel_regularizer =
regularizers.l1(0.0001)))

model3.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])

model3.summary()


#Training

#Training params

batch_size = 128

epochs = 20


#Time it

start = time.time()

history3 = model3.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)

end = time.time()

print(end - start)

#Save model

model.save("cifar.model")
```

```
#**************

# Effect on train size


def model_train_sizesII(x_train_orig, y_train_orig, x_test, y_test,
train_size, batch_size, epochs):


    #Figure

    fig, axs = plt.subplots(4, 2)

    results = []

    count = 0


    for n in train_size:

        'Perform training for each size of n'

        #Split data

        x_train = x_train_orig[1:n]; y_train = y_train_orig[1:n]


        #Format data

        x_train = x_train.astype("float32") / 255

        #Convert class vectors to binary class matrices

        y_train = keras.utils.to_categorical(y_train, num_classes)


        #Train model

        start = time.time()

        history = model3.fit(x_train, y_train, batch_size=batch_size,
epochs=epochs, validation_split=0.1)

        end = time.time()

        timeX = (end - start)

        print('time train = {}'.format(timeX))


        #Model prediction results

        y_preds_train = model.predict(x_train)

        y_preds_train = np.argmax(y_preds_train, axis=1) #Highest prob

        #Format

        y_train1 = np.argmax(y_train, axis=1)

        train_accuracy = accuracy_score(y_train1, y_preds_train)
```

```
        print('train accuracy = {}'.format(train_accuracy))

        #Model prediction results
        y_preds_test = model.predict(x_test)
        y_preds_test = np.argmax(y_preds_test, axis=1) #Highest prob
        #Format
        y_test1 = np.argmax(y_test, axis=1)
        test_accuracy = accuracy_score(y_test1, y_preds_test)
        print('test accuracy = {}'.format(test_accuracy))

        #Results
        d = {
            'training size' : n,
        'train time': timeX,
        'train accuracy' :  train_accuracy,
        'test accuracy' :  test_accuracy,
        }
        results.append(d)

        #Plots
        #Accuracy
        axs[count, 0].plot(history.history['accuracy'])
        axs[count, 0].plot(history.history['val_accuracy'])
        axs[count, 0].set_title('Model Accuracy, train size =
{}'.format(n))
        axs[count, 0].set(xlabel='epoch', ylabel='accuracy')
        axs[count, 0].legend(['train', 'val'], loc='upper left')
        #Loss
        axs[count, 1].plot(history.history['loss'])
        axs[count, 1].plot(history.history['val_loss'])
        axs[count, 1].set_title('Model Loss, train size = {}'.format(n))
        axs[count, 1].set(xlabel='epoch', ylabel='loss')
        axs[count, 1].legend(['train', 'val'], loc='upper left')
        #Update count
        count = count + 1
```

```
    #Show plot
    plt.show()
    #Dataframe of results
    df_results = pd.DataFrame(results)

    return df_results

#Apply
train_size = [5000, 10000, 20000, 40000]
df_resultsII = model_train_sizesII(x_train_orig, y_train_orig, x_test,
y_test, train_size, batch_size, epochs)
```