Part 1 – Convolution Basics:

1-a) Convolute function: The convolute function accepts 2d input array and kernel array, performs convolution and returns the result array. Multiple 'for' loops are used, where inner loops are used to increment indices i & j and outer loops are used to increment variables temp_step_i and temp_step_j. Simple multiplication operation is performed on the inp_val and ker_val where these values are assigned with help of indices mentioned above. The results are first stored in a few temp variables then finally appended in final result_array which is the output of this function.

```
# (i_m , i_n) (k_m , k_n) are m x n dimensions of input array and kernel array
# loop through individual values and perform convolution to get output array
temp_step_i=0
for x in range (i_m-k_n+1):
    temp_arr=[]
    temp_step_j=0
    for y in range (i_n - k_n+1):
        temp_res = 0
        for i in range(k_m):
            for j in range(k_n):
                inp_val = input_arr[i+temp_step_i][j+temp_step_j]
                ker_val = kernel[i][j]
                res_val = ker_val*inp_val
                temp_res=temp_res+res_val
        temp_arr.append(temp_res)
        temp_step_j=temp_step_j+STRIDE      #STRIDE is defined as 1
    result_arr.append(temp_arr)
    temp_step_i = temp_step_i+STRIDE        #STRIDE is defined as 1
return result_arr
```

1-b) Kernel application on image: A simple image of a shop is used where first red colour values are extracted then the given kernels K1 = [[-1,-1,-1],[-1,8,-1],[-1,-1,-1]] and K2 = [[0,-1,0],[-1,8,-1],[0,-1,0]] are applied on red colour array by using the convolute function above.

1.1 Original image

1.2 Using red colour

1.3 Convolution with K1

1.4 Convolution with K2

Image 1.2 is created by using only red colour channel of the original image. Thus, we can see bright pixels where the colour was red & white (white colour contains max values of all three colours in rgb) and dark pixels where there is no presence red colour, i.e., near door, window and background. After convolution with given kernel K1 the output image 1.3 shows different edges present in the image. There are also a few additional edges that are not visible to the naked eye but present in the output, especially in the white section. Convolution with K2 results in image 1.4 which is similar to image 1.2 but the pixels are darker when compared to 1.2. Also, the boundaries where colour change takes place are much sharper, which can be seen in the door and window section, that have gradient colour finish in the original image.

Part 2 – CNN on CIFAR10 image dataset:

2-a) CNN architecture: The architecture of the CNN model can be represented by the figure given below.

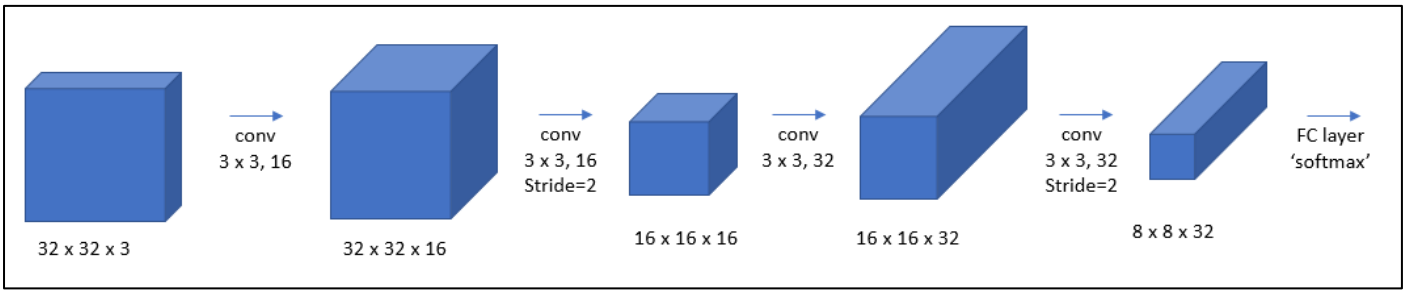

Fig 2-a1: CNN architecture

The input array of the image is a 32 x 32 array over 3 channels. 'Same' padding is used during all the convolutions. The significance of using 'same' padding here is the output array dimensions will be same to that of input when stride is 1 and the dimensions will be halved when stride is 2.

1) In the first convolution a 3 x 3 x 3 kernel is applied on the input across **16 channels** which results in 32 x 32 x 16 block.
2) Then again 3 x 3 x 16 kernel is applied across **16 channels** but with **stride 2**, so the output block has size of 16 x 16 x 16.
3) In the next convolution 3 x 3 x 16 kernel is applied across **32 channels**, so the output block has more depth. The size of output block becomes 16 x 16 x 32
4) In the next convolution 3 x 3 x 32 kernel is applied across **32 channels** with **stride 2**, so the size is reduced to 8 x 8 x 32
5) Then we apply dropout because we don't use any regularization in convolution.
6) Then the we use flatten operation to convert multidimensional array to a 1-d vector.
7) The vector from step 6 is used as an input in the fully connected dense layer with 'softmax' (multiclass logistic regression) activation function and L1 regularization.

2-b1) Model parameters: The parameters of different layers of the given model are presented in the table below. The calculation of number of parameters is shown in column *'Kernel weights + biases'*. In this model the parameters are increasing layer by layer. The most number of parameters can be seen in the final layer i.e., Dense layer where the input array is of length 2048 and we are predicting for 10 classes so the total number of parameters are 2048 x 10 + 10 biases = 20490

Layer	Kernel weights + biases	Parameters	Output Shape
conv2d (Conv2D)	3x3x3x16 + 16	448	(None, 32, 32, 16)
conv2d_1 (Conv2D)	3x3x16x16 + 16	2320	(None, 16, 16, 16)
conv2d_2 (Conv2D)	3x3x16x32 + 32	4640	(None, 16, 16, 32)
conv2d_3 (Conv2D)	3x3x32x32 + 32	9248	(None, 8, 8, 32)
dropout (Dropout)		0	(None, 8, 8, 32)
flatten (Flatten)		0	(None, 2048)
dense (Dense)	2048x10 + 10	20490	(None, 10)
	Total Parameters	**37146**	

Table 2-b1: Model Parameters

Accuracy: After completion of final epoch, the model shows an accuracy of 55.5% on training set and 50% on validation set. Higher training accuracy is a sign of overfitting.

Baseline Comparison: If we consider baseline model which predicts most common class, then accuracy of this model would be close to 10% if we assume the test data has datapoints equally distributed among all 10 classes. Compared to 50% accuracy of the given model we can say that the given model is somewhat better than a baseline model.

2-b2) Model performance: The performance of the model after every epoch completion is recorded in the graph below. From this graph we can see that after 10 epochs the accuracy of model on training data is more than accuracy on validation data. This might have happened because the model is ***'overfitting'*** and is unable to predict properly on the validation data. The model weights after completion of 6th / 7th epoch will be much better as they show similar performance on both training and test data.

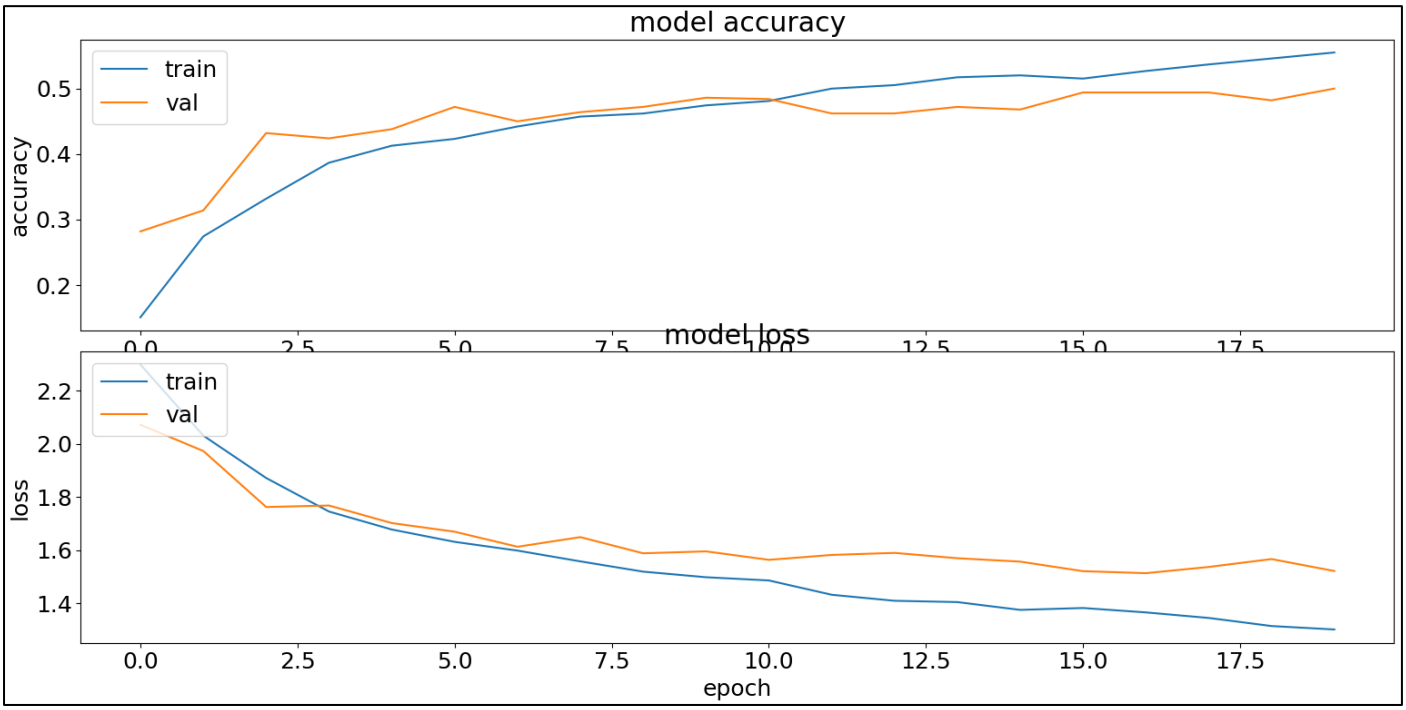

Fig 2-b2: Model Performance History (5K training data & L1=0.0001)

2-b3) Varying amount of training data: The same model is trained for different sizes of training data, i.e., 5K, 10K, 20K and 40K images respectively. The performance of these models is represented by graphs below.

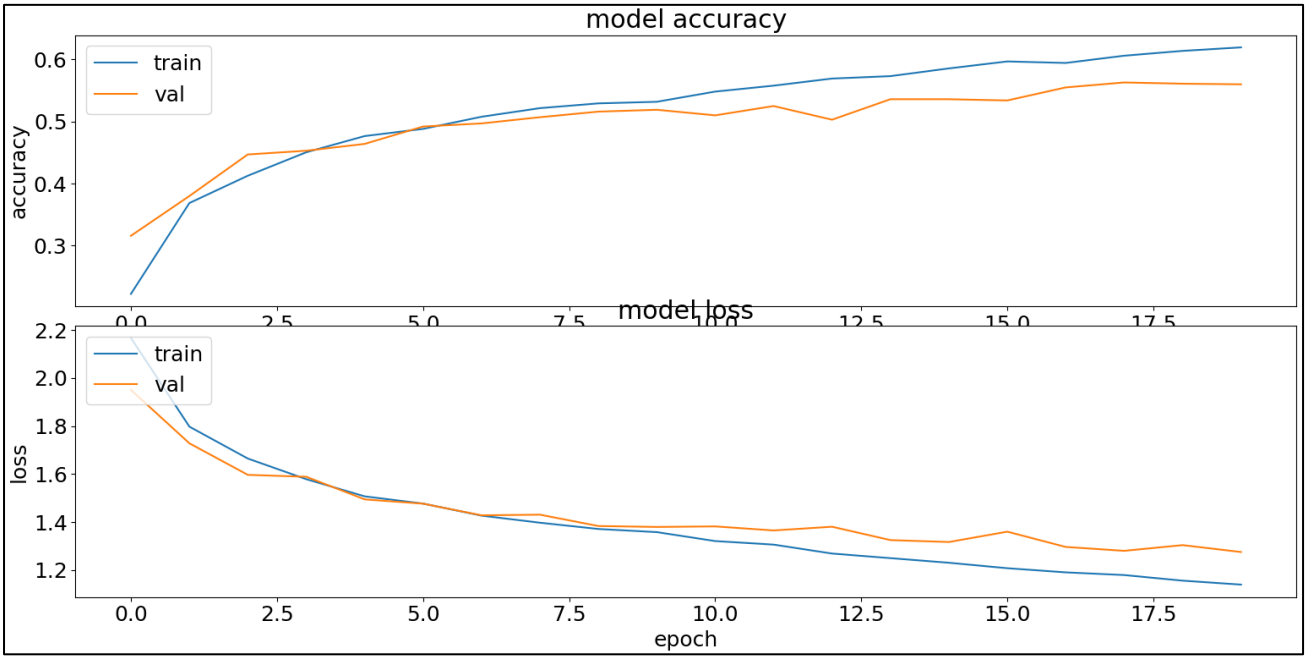

Fig 2-b31: Model Performance History (10K training data)

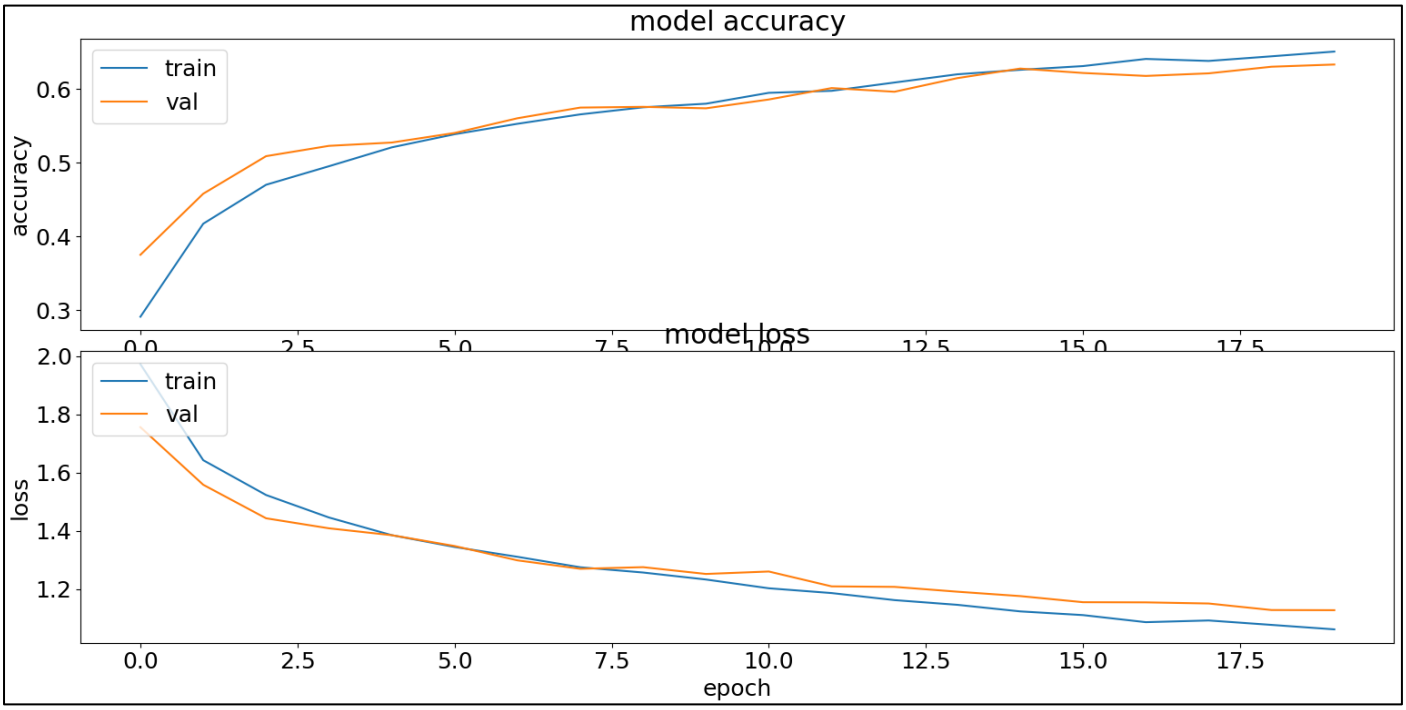

Fig 2-b32: Model Performance History (20K training data)

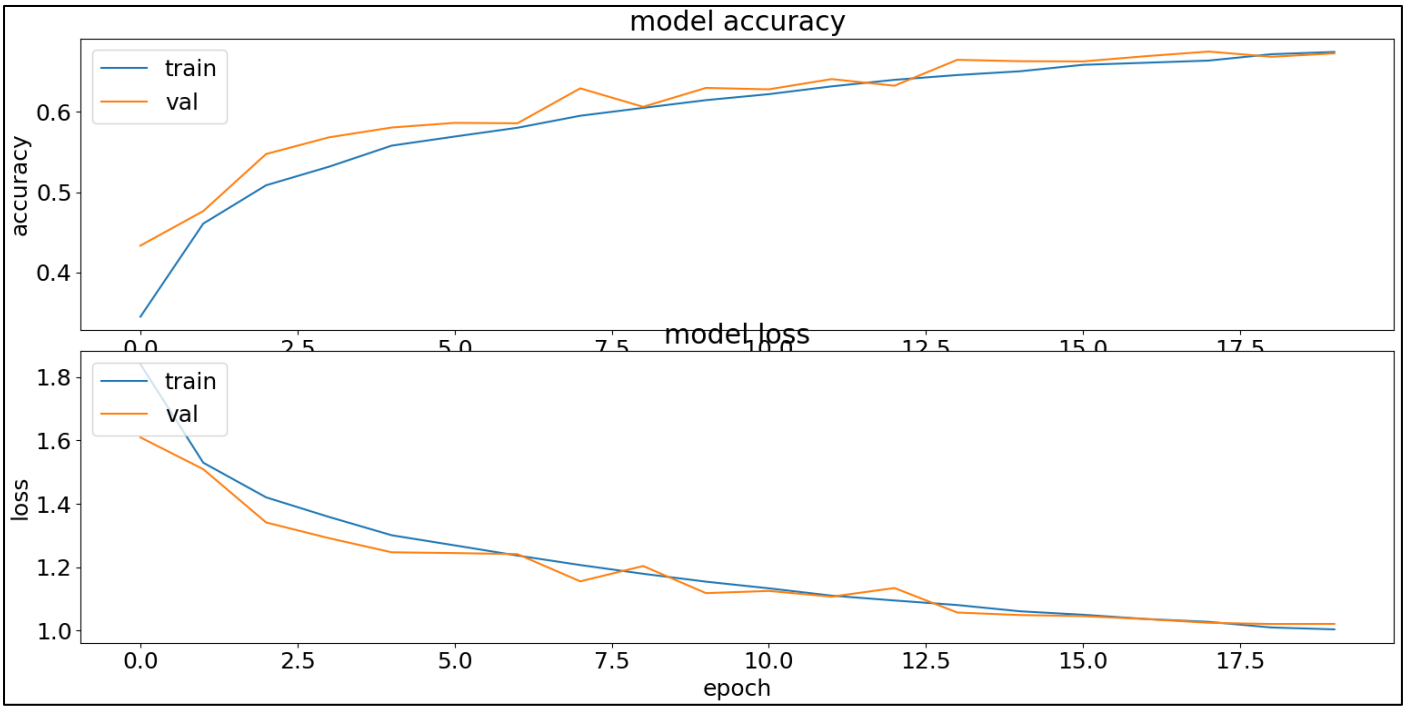

Fig 2-b33: Model Performance History (40K training data)

From the figures above we can say that increasing the data to 10K images is not that effective the validation accuracy might have increased to 55% but the model clearly shows sign of overfitting. For model with 20K images the validation accuracy after final epoch 63%. There is still probability of overfitting in the final few epochs. The model with 40K training data, gives validation accuracy of about 67% and there aren't any visible signs of overfitting as the accuracy of training and test data are very close to each other. After looking at all the performances above we can say that CNN performs much better on large amount of training data.
Time taken for training models (on laptop with 960M GPU) for different size of training data is given below:
5K - 32.34 secs **10K - 52.73 secs** **20K - 79.15 secs** **40K - 143.21 secs**

As expected, the time increases with an increase in amount of training data.

2-b4) Changing L1 weight parameter: The dense layer of the given model utilizes L1 regularization. We can compare the effects of different values of L1 weight parameter on the performance of the model. Here values of L1 used are [0, 0.0001, 0.01, 0.1]. The performance of L1=0.0001 is shown in the '*Fig 2-b2*'. For rest of the L1 performances are shown below.

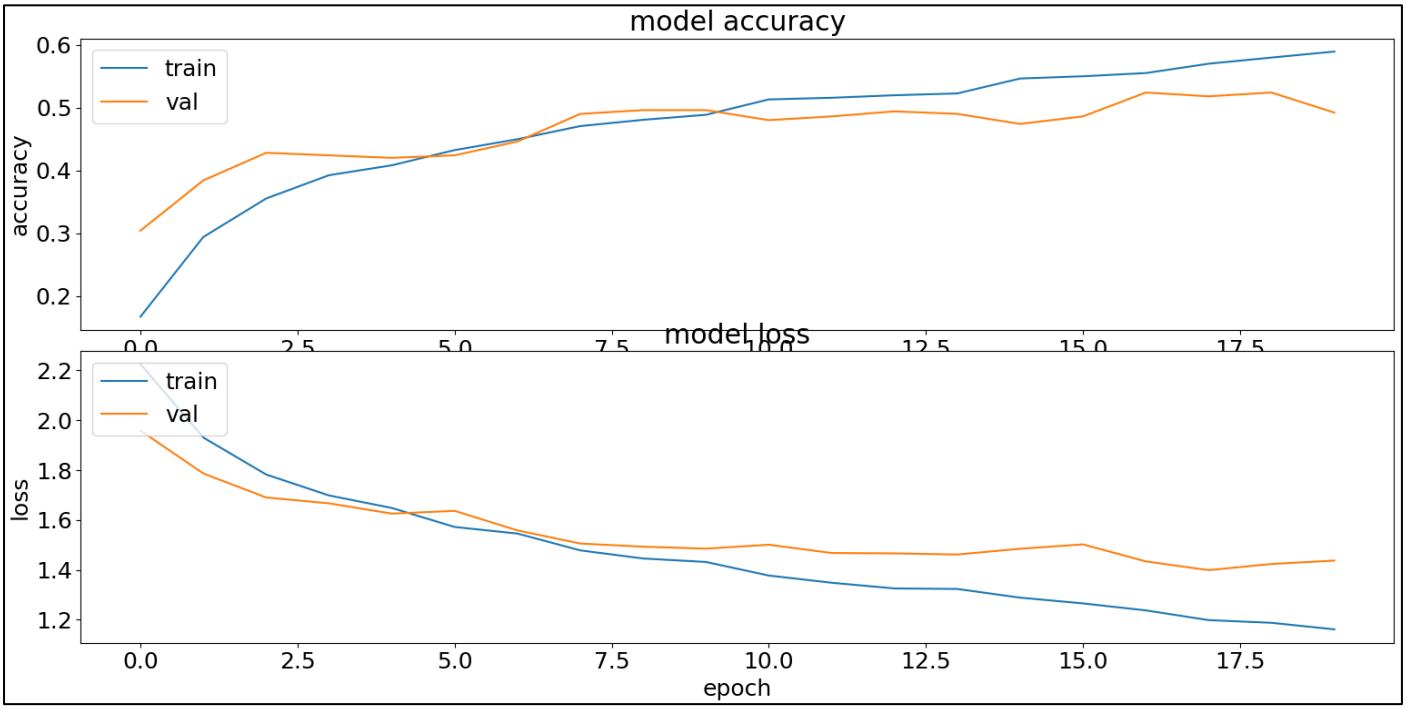

Fig 2-b41: Model Performance History (L1 = 0)

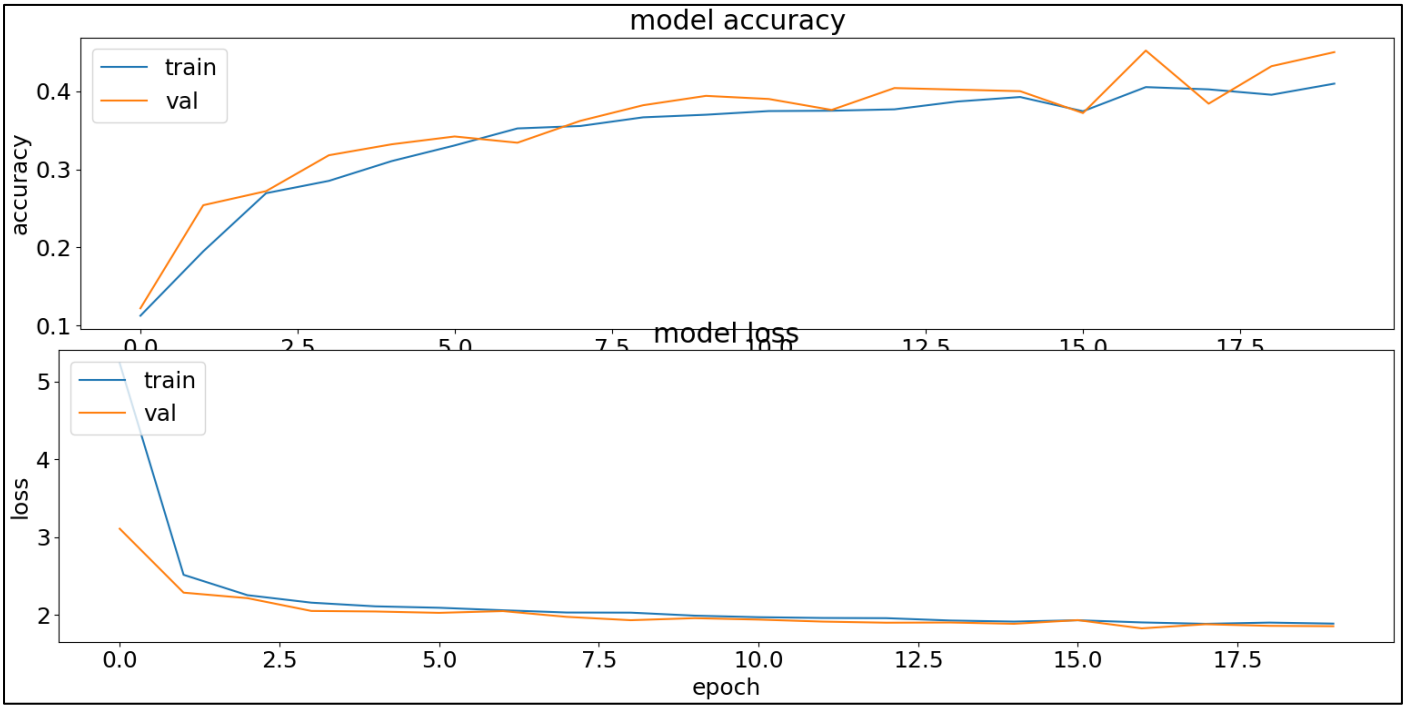

Fig 2-b42: Model Performance History (L1 = 0.01)

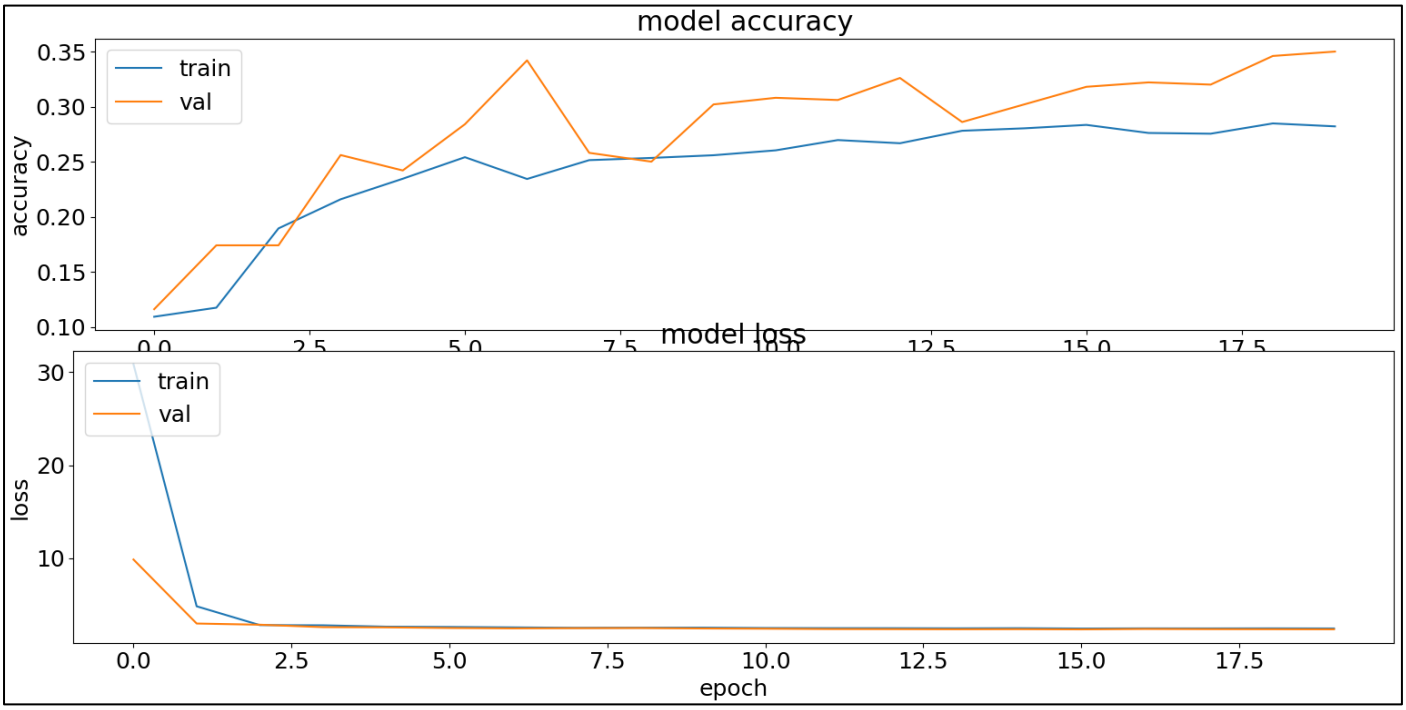

Fig 2-b43: Model Performance History (L1 = 0.1)

From the graphs above we can judge that as we increase the weight parameter of L1 the accuracy of the model decreases. When L1 parameter is close to 0 the validation accuracy is around 50% which reduces to 35% when the parameter increases towards 0.1. Also, the overfitting problem persists in all of the variations of L1 weight parameters. From the analysis above we can say that increasing the amount of training data is more beneficial for the model in terms of increasing accuracy and reducing overfitting.

2-c) Using max-pooling for downsampling: As stated in the section 2-c, both the layers with stride = 2 are replaced by same layers with same size kernels and removing the strides (stride =1 by default). An additional max-pool layer is also added with pool-size (2,2) so the output dimension is halved after each max-pool layer. The parameters of the model are shown in the table below.

Layer	Kernel weights + biases	Parameters	Output Shape
conv2d (Conv2D)	3x3x3x16 + 16	448	(None, 32, 32, 16)
conv2d_1 (Conv2D)	3x3x16x16 + 16	2320	(None, 32, 32, 16)
max_pooling2d (MaxPooling2D)		0	**(None, 16, 16, 16)**
conv2d_2 (Conv2D)	3x3x16x32 + 32	4640	(None, 16, 16, 32)
conv2d_3 (Conv2D)	3x3x32x32 + 32	9248	(None, 16, 16, 32)
max_pooling2d_1 (MaxPooling2D)		0	**(None, 8, 8, 32)**
dropout (Dropout)		0	(None, 8, 8, 32)
flatten (Flatten)		0	(None, 2048)
dense (Dense)	2048x10 + 10	20490	(None, 10)
	Total Parameters	**37146**	

Table 2-c1: Model Parameters (with MaxPooling)

Training time for model with maxpooling on 5K training images= **33.40 secs**
The time taken is just 1.1 seconds more than the model without maxpooling. This increment might be less in this case, but in-general maxpooling requires more time than strided downsampling because of two reasons.

1) No. of calculations in convolution where stride is 1 **>** No. of calculations in convolution where stride is 2
2) Additional calculations required in maxpooling layer. This layer is not required when stride is 2

The performance of the model is not too different when compared to previous model. The validation accuracy score is **53%**, that is a small increment of 3% when compared with the previous model. Also overfitting can be seen which is similar to the previous model.

The performance history of the model is shown in the figure below:

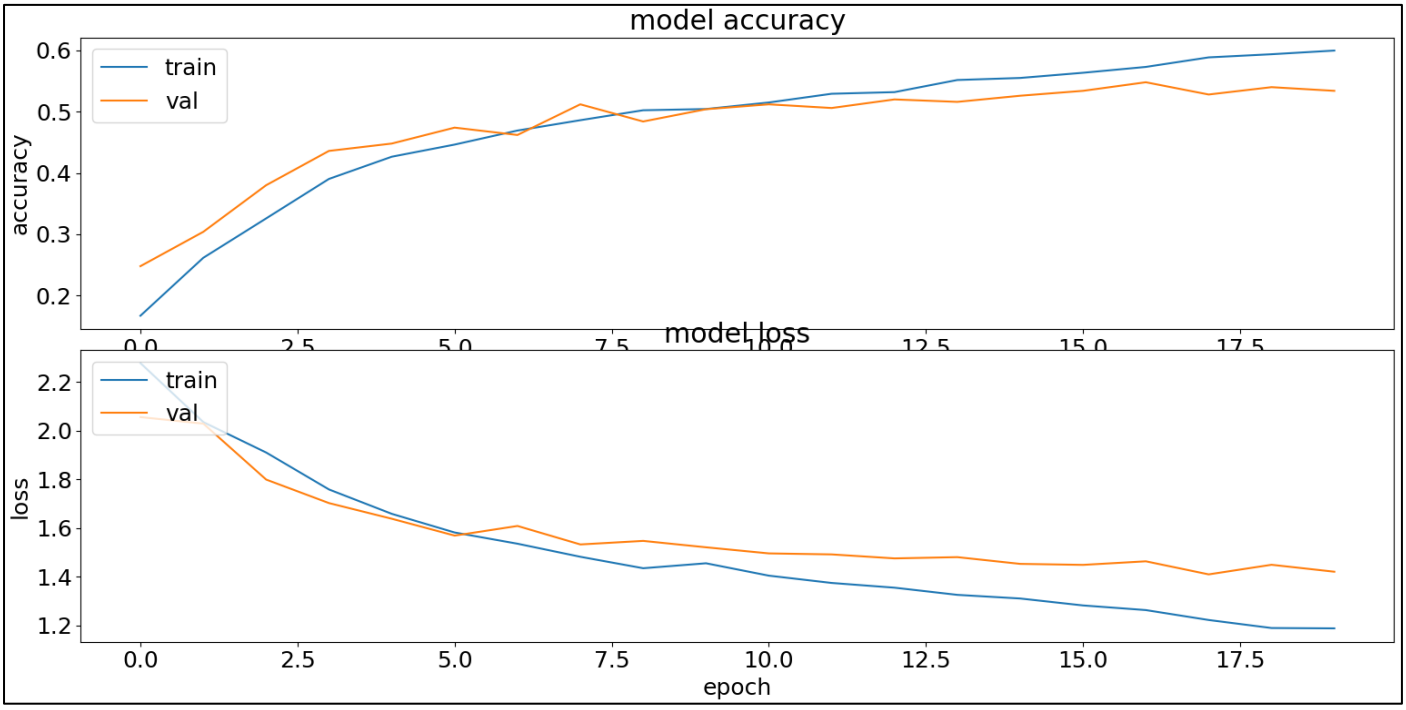

Fig 2-c1: Model Performance History (with Max-Pooling)

Appendix:

Part 1 contains code for the simple convolution function and kernel application on sample image.

Part 2 contains the given code along with the modified model that uses max-pooling.

Part 1 code begins here:

```
Name: Omkar Pramod Padir
# Student Id: 20310203
# Course: Machine Learning CS7CS4
# Week 8 Assignment Part1


import numpy as np

# Part i-a starts here

def Convolute(input_arr, kernel):

    result_arr = []

    # Define step size
    STRIDE = 1

    # Get the dimensions of input and kernel
    i_m = len(input_arr)
    i_n = len(input_arr[0])

    k_m = len(kernel)
    k_n = len(kernel[0])

    # loop through individual values and perform convolution to get output array
    temp_step_i=0
    for x in range (i_m-k_n+1):

        temp_arr=[]
        temp_step_j=0
        for y in range (i_n - k_n+1):

            temp_res = 0
            for i in range(k_m):
                for j in range(k_n):

                    inp_val = input_arr[i+temp_step_i][j+temp_step_j]
                    ker_val = kernel[i][j]

                    res_val = ker_val*inp_val

                    temp_res=temp_res+res_val

            temp_arr.append(temp_res)
```

```
            temp_step_j=temp_step_j+STRIDE

        result_arr.append(temp_arr)
        temp_step_i = temp_step_i+STRIDE


    return result_arr


# Part i - b starts here

from PIL import Image

im = Image.open('Images|Shop.png') # Shop.png
rgb = np.array(im.convert('RGB'))
r = rgb [:,:,0] # array of R pixels

Image.fromarray(np.uint8(r)).show()
Image.fromarray(np.uint8(r)).save("Images|Shop_red.png")

# Define Kernels
K1 = [[-1,-1,-1],[-1,8,-1],[-1,-1,-1]]
K2 = [[0,-1,0],[-1,8,-1],[0,-1,0]]

# Output of rgb array after convolution ; parameters are inputArray, Kernel and Stride
res1 = Convolute(r,K1)
Image.fromarray(np.uint8(res1)).show()
Image.fromarray(np.uint8(res1)).save("Images|Shop_K1.png")

res2 = Convolute(r,K2)
Image.fromarray(np.uint8(res2)).show()
Image.fromarray(np.uint8(res2)).save("Images|Shop_K2.png")
```

Part 2 code begins here:

```
# Name: Omkar Pramod Padir
# Student Id: 20310203
# Course: Machine Learning CS7CS4
# Week 8 Assignment Part 2


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
```

```
from datetime import datetime
import time

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()

start = time.time()
now = datetime.now()
current_time = now.strftime("%H:%M:%S")
print("Start Time =", current_time)


n=5000        #5000,10000,20000,40000


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)

max_pool_flag = True

use_saved_model = False
if use_saved_model:
    model = keras.models.load_model("cifar.model")
elif max_pool_flag == False :
    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(1))) #0.0001,0.01,
0.1, 0,1
```

```
    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")

    print("End Time =", datetime.now().strftime("%H:%M:%S"))
    print("Total Time Taken (secs): ", (time.time() - start))

    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()

else :  # This model uses maxpool instead of strides
    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')) #, strides=(2,2)
    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')) #, strides=(2,2)
    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)))
#0.0001,0.01, 0.1, 0,1
    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")

    print("End Time =", datetime.now().strftime("%H:%M:%S"))
    print("Total Time Taken (secs): ", (time.time() - start))

    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))
```