Assignment 8 - Hanwen Liang – SID:23359826

i.a

The code below is the implementation of convolution:

```
# convolve input array to kernel
def convolve(input_array, kernel):
    # get kernel height and width
    k_h, k_w = kernel.shape[0], kernel.shape[1]
    # add padding to keep output the same size as input
    pad_img = np.pad(input_array, (((k_h - 1) // 2, (k_h - 1) // 2), ((k_w - 1) // 2, (k_w - 1) // 2)))
    # compute convolution
    ret = np.zeros((input_array.shape[0], input_array.shape[1]))  # store result
    for i in range(input_array.shape[0]):
        for j in range(input_array.shape[1]):
            temp_convolution = (kernel * pad_img[i: i + k_h, j: j + k_w]).sum()
            ret[i, j] = temp_convolution
    return ret
```

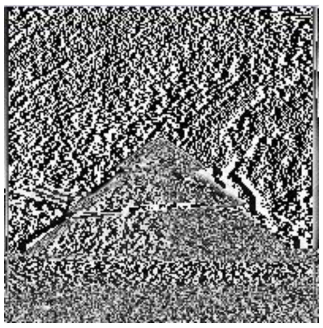
i.b

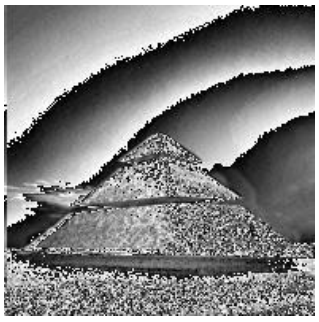
The image below has been reshaped by 200x200.

Two Kernels

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

Convolve input image with Kernel 1 (left) & Kernel 2(right)





ii.a

The ConvNet consists of several key components:

- Input Layer (32, 32, 3)
  It represents input images consisting of 32x32 pixels and 3 colour channels for RBG.
- Convolutional Layers (4 layers)

```
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'))
```

The first Layer has 16 output filters and a 3x3 kernel. It takes the training data (32, 32, 3) as its input and the convolution output remains the same size as the input due to the padding being set by 'same'. A rectified linear unit (ReLU) is used to activate input features. The resulting output dimension of the first layer is (32, 32, 16).

The Layer 2 also has 16 output filters and a 3x3 kernel. It takes the output of layer 1 (32, 32, 16) as input and it adds padding to ensure the same spatial dimension of output as the input. It also uses ReLU, a rectified linear unit, as its activation function. By setting stride (2, 2), move the filter by 2 pixels at a time horizontally and vertically to downsample output. The resulting output dimension of the second layer is (16, 16, 16).

For the third layer, it has 32 output filters and a 3x3 kernel. It takes the output of layer 2 (16, 16, 16) as input, and it adds padding to keep the same output size as the input. Again it uses ReLU to activate function. The resulting output dimension of the third layer is (16, 16, 32).

The last layer of convolution has 32 output filters and a 3x3 kernel. It takes the output of layer 3 (16, 16, 32) as input and it adds padding to retain the same output size as the input. It also uses ReLU, a rectified linear unit, as its activation function. By setting stride (2, 2), move the filter by 2 pixels at a time horizontally and vertically to downsample the output. The resulting output dimension of the third layer is (8, 8, 32).

- Dropout Layer
  `model.add(Dropout(0.5))`
  During training, around 50% of input parameters will be randomly set to zero to prevent overfitting. The resulting outputs of dropout layer retain the same size as the input (8, 8, 32).
- Flatten layer
  `model.add(Flatten())`
  It converts outputs from preceding layers into a 1D array and this is necessary for transition to fully connected layers. The resulting output of flatten layer is an array of 2048 elements.
- Dense Layer (Output)
  ```
  model.add(Dense(num_classes, activation='softmax', kernel_regularizer=
  regularizers.l1(0.0001)))
  ```
  In the dense layer, each node is connected with nodes of the previous layer and it predicts the probability of each class. The resulting output shows probabilities of 'num_class' (10 outputs in this case) by using the softmax activation function. It uses L1 regularization to prevent overfitting, with rate of 0.0001. The resulting output is 10 probabilities of predicted classes.

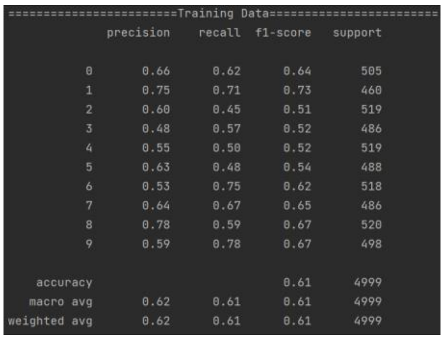
ii.b (i)

Based on Keras report, there are a total of 37146 parameters in this model. The fourth convolutional layer contains the most parameters (9248) and this is calculated by the formula below:

$$Prameters = (Number\ of\ Input\ Channels * Kernel\ Width * Kernel\ Height + Bias) * Number\ of\ Output\ Channels$$

The parameters of the fourth layer = (32 * 3 * 3 + 1) * 32 = 9248

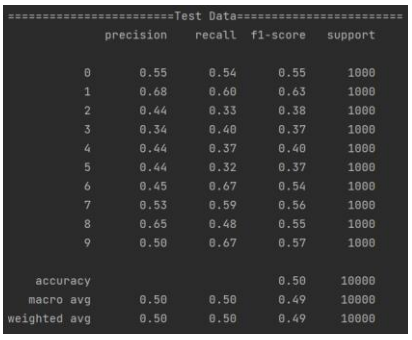
The below shows that training data performs better than test data, with a slightly higher average accuracy of 61% than the 50% of test data.



```
=======================Training Data=======================
              precision    recall  f1-score   support

           0       0.66      0.62      0.64       505
           1       0.75      0.71      0.73       460
           2       0.60      0.45      0.51       519
           3       0.48      0.57      0.52       486
           4       0.55      0.50      0.52       519
           5       0.63      0.48      0.54       488
           6       0.53      0.75      0.62       518
           7       0.64      0.67      0.65       486
           8       0.78      0.59      0.67       520
           9       0.59      0.78      0.67       498

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

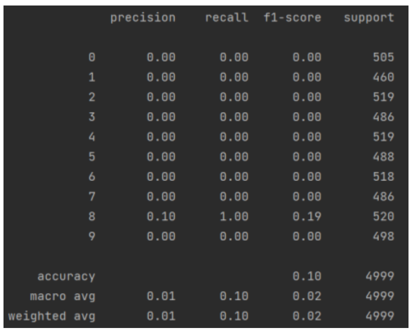


```
=======================Test Data=======================
              precision    recall  f1-score   support

           0       0.55      0.54      0.55      1000
           1       0.68      0.60      0.63      1000
           2       0.44      0.33      0.38      1000
           3       0.34      0.40      0.37      1000
           4       0.44      0.37      0.40      1000
           5       0.44      0.32      0.37      1000
           6       0.45      0.67      0.54      1000
           7       0.53      0.59      0.56      1000
           8       0.65      0.48      0.55      1000
           9       0.50      0.67      0.57      1000

    accuracy                           0.50     10000
   macro avg       0.50      0.50      0.49     10000
weighted avg       0.50      0.50      0.49     10000
```

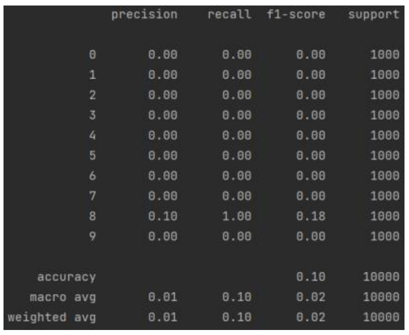
The below shows that comparison against the most frequent baseline predictor, with training data on the left, and test data on the right. It is apparent that the baseline predictor poorly performs across all classes, with only 10% accuracy overall. It fails to provide meaningful prediction as it always predicts the most common class which is the 8 in this case with its precision of 10% and f1 score of 19%. Therefore, it is recommended to train a more complex model to address the issue.



```
              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
```

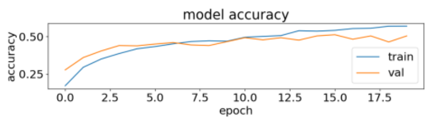


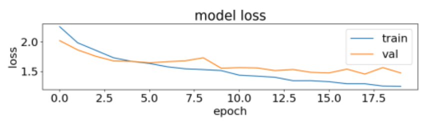
```
              precision    recall  f1-score   support

           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
```

ii.b (ii)





This model tends to be overfitting. By observing the model accuracy plot, both accuracies of training data and test data initially rise but after epoch 10 the line of training data surpasses that of test data, showing a dramatic difference afterwards. This indicates that the model poorly performs for unseen data and the model tends to memorise training data instead of learning its pattern.

This can also be proved by the second plot. Initially, both losses of training data and test data are decreasing. However, the loss of test data ceases to decline after 8 or 9 epochs, remaining at a high level afterwards. The training data, in contrast, continues to decrease, leading the gap between the two datasets to be substantial afterwards. Consequently, the trained model fits too closely to the training data.

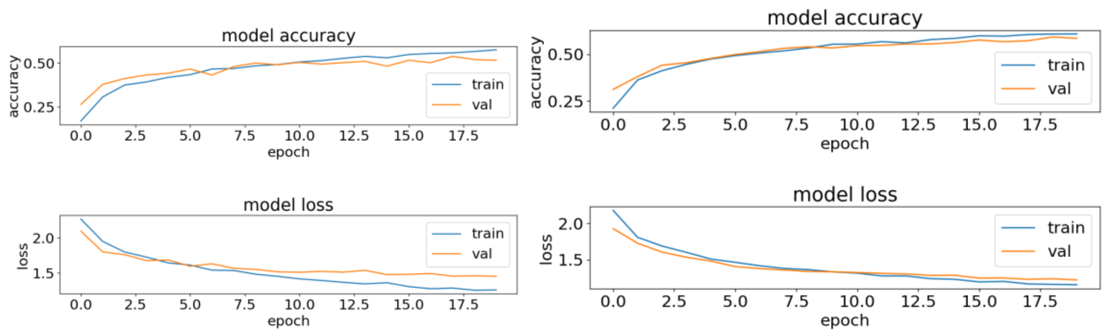
ii.b (iii)

Amount of Training Data	Accuracy of Training Data	Accuracy of Test Data	Time for Training (s)
5K	0.63	0.50	11.3337
10K	0.68	0.57	23.3685
20K	0.69	0.62	42.7667
40K	0.72	0.67	82.1953

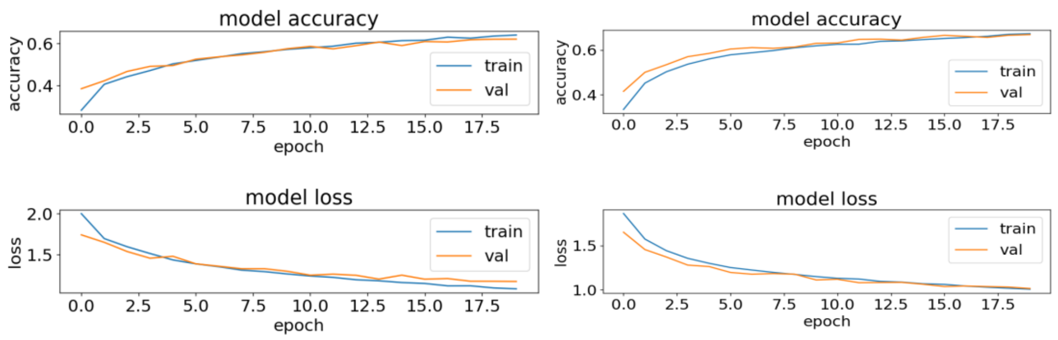
With the amount of training data increasing from 5K to 40K, the training time increases by nearly 8 times. Both accuracies of training data and test data increase as well. Notably, the latter accuracy of test data climbs dramatically by almost 20%, leading to similar accuracy to training data.

Plots of Accuracy and Loss of Varying Training Data:

When amount of training data = 5K (left) & 10K (right)



When amount of training data = 20K (left) & 40K (right)



By observing the above plots, the lines of training data and test data tend to better fit each other with increasing training data. Especially when training data is 40K, two lines of training data and test data tend to overlap after 15 epochs, indicating a great prediction under unknown data.

ii.b (iv)

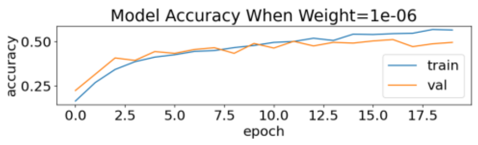
Weight Parameter of L1	Accuracy of Training Data	Accuracy of Test Data

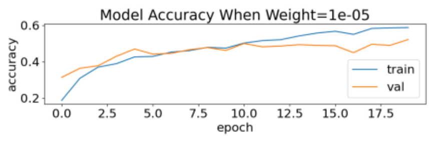
10^{-6}	0.63	0.50
10^{-5}	0.65	0.50
10^{-4}	0.60	0.50
10^{-3}	0.55	0.48
0.01	0.44	0.42
0.1	0.40	0.38
0	0.63	0.51
1	0.10	0.10
5	0.10	0.10
10	0.10	0.10

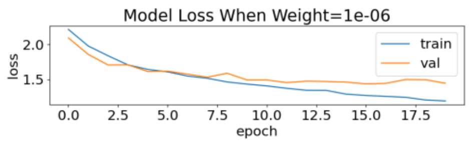
The above table shows that a certain range of L1 weights can effectively affect both the accuracies of training data and test data. For example with the L1 weight increasing from 10^{-4} to 0.1, both accuracies of training data and test data decrease accordingly. However, the accuracies of both datasets remain stable at both ranges of L1 weights, the range of 10^{-6} to 10^{-4} with training data accuracy around 0.6 and test data accuracy equal to 0.5, and the range of 1 to 10 with both accuracies of training data and test data equal to 0.1. When L1 weight is equal to 0, both accuracies of training data and test data climb to 0.63 and 0.51 respectively.

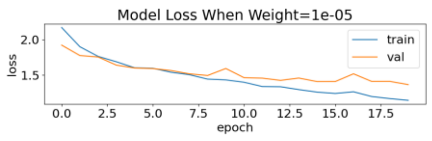
Plots of Accuracies with Varying L1 Weights:

Weight = 10^{-6}(left below) and Weight = 10^{-5}(right below)

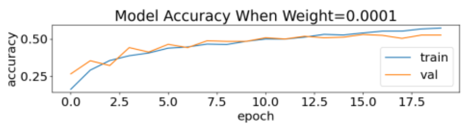


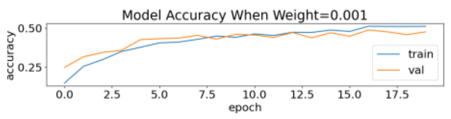


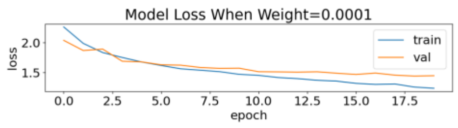


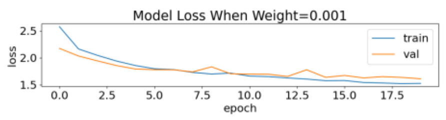


Weight = 10^{-4}(left below) and Weight = 10^{-3}(right below)

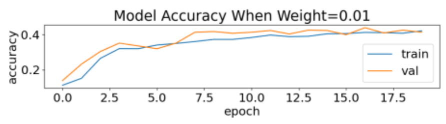


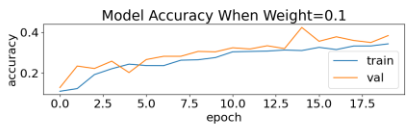


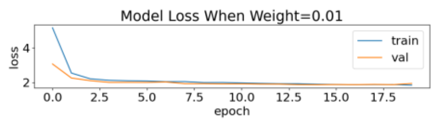


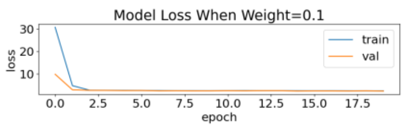


Weight = 10^{-2}(left below) and Weight = 10^{-1}(right below)

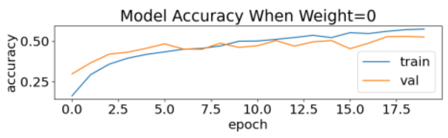


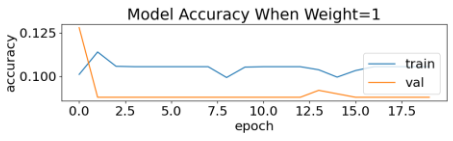


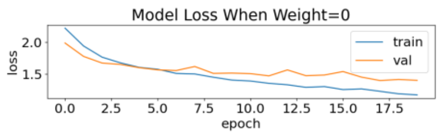


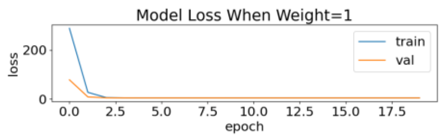


Weight = 0 (left below) and Weight = 1 (right below)

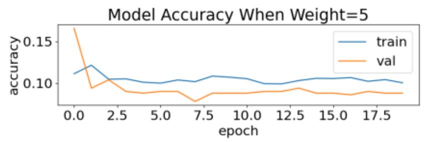


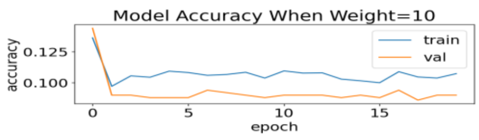


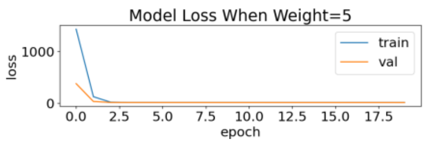


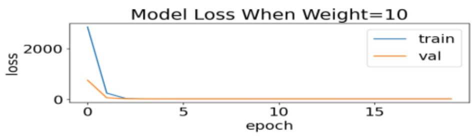


Weight = 5 (left below) and Weight = 10 (right below)









The above plots indicate that increasing L1 weights can prevent overfitting. When L1 weight is set by 10^{-6}, the accuracy line of training data is highly above that of test data. With increasing L1 weights from 10^{-4} to 0.1, both lines of accuracy and loss are about to overlap, especially when L1 weight is equal to 0.01, although both their accuracies decline (drop 0.12 for test data, 0.2 for training data). Overall, the model tends to perform better for unseen data with increasing L1 weight. Whereas, the accuracy lines of two datasets become apart and the losses are about to increase afterwards.

Both techniques including increasing training data and increasing L1 weight can prevent overfitting. It is suggested to apply the former method because the resulting accuracy is higher than the latter one.

ii.c (i)

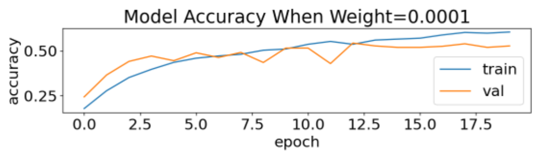
Replace the given code as below:

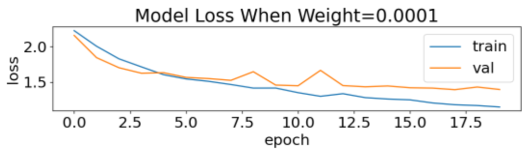
```
model.add(Conv2D(16, (3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))  # add max pooling
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))  # add max pooling
```

ii.c (ii)

Net Architecture	Number of Parameters	Accuracy of Training Data	Accuracy of Test Data	Training Time
Initial ConvNet	37146	0.63	0.50	11.3337
Modified ConvNet	37146	0.66	0.54	17.0903

By observing the table above, the total number of parameters of modified ConvNet remains the same as the initial model with the number of 37146. Whereas, both accuracies and training time have changed accordingly. The training time increases by around 6 seconds and the accuracies of both training data and test data increase by 0.03 and 0.04 respectively.





There are several potential reasons for increasing training time. Adding two extra Max Pooling layers and the resulting higher complexity leads to more computation and more execution time. This also requires additional convolutional processes which take extra computing time.

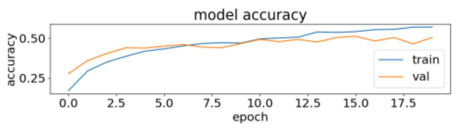
ii.d

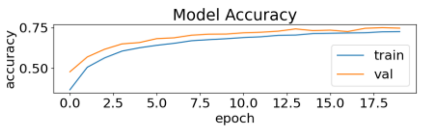
Net Architecture	**Number of Epoch**	**Amount of Training Data (K)**	**Number of Parameters**	**Accuracy of Training Data**	**Accuracy of Test Data**	**Training Time**
Modified ConvNet	20	5	23314	0.52	0.44	8.1622
Initial ConvNet	20	5	37146	0.63	0.5	11.3337
Modified ConvNet	20	50	23314	0.68	0.65	83.1577
Initial ConvNet	20	50	37146	0.77	0.74	160.0671

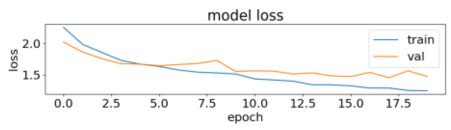
Firstly, I have compared the modified ConvNet which is thinner and deeper, with the initial ConvNet, using a stable epoch number of 20 and a different amount of training data (5K and 50K). By observing the above table, `the resulting accuracies of both test data and training data dramatically increase. For the initial model, it achieves relatively good accuracies for test data (0.74) and training data (0.77) when increasing the amount of training data to 50K. As for the modified model, it also performs better (Accuracy: 0.65 for test data, 0.68 for training data) when increasing the training data amount. The resulting training time for both initial and modified models has increased to 160s and 83s respectively.

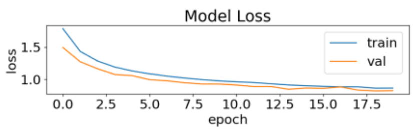
Plots of Initial Model:

When epoch=20, training data are 5K (left below) and 50K (right below)









From the above plots, it is true that increasing the amount of training data can improve accuracies and optimise losses. The initial model is more appropriate as it only needs to iterate 20 epochs to reach such a good accuracy.

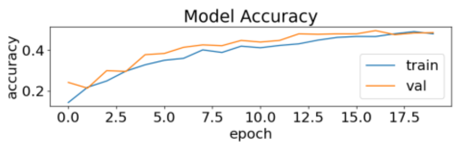
In order to improve the accuracy of modified ConvNet, there are a few more training being taken.

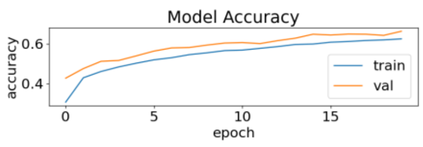
Net Architecture	Number of Epoch	Amount of Training Data (K)	Number of Parameters	Accuracy of Training Data	Accuracy of Test Data	Training Time (s)
Modified ConvNet	30	50	23314	0.71	0.66	111.0761
	65	50	23314	0.75	0.69	209.6535
	80	50	23314	0.78	0.7	257.0558
	100	50	23314	0.79	0.7	331.1557

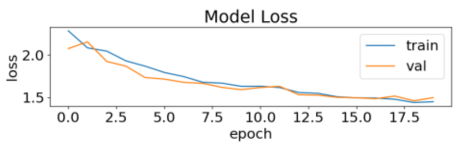
From the above table, with a constant amount of training data being used, increasing epoch numbers can improve model accuracies. When the epoch is set by 100 the model accuracy reaches 0.7, which is the same as when the epoch is equal to 80. Although both accuracies have been improved, it is suggested to use epoch 65 as the gap between test data and training data is relatively smaller, even with a relatively good accuracy of 0.69. And the training time of epoch 65 is almost 50 seconds less than the time when epoch is equal to 80.

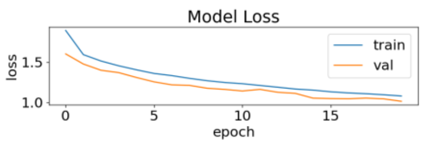
Plots of Modified Model:

When epoch=20, training data are 5K (left below) and 50K (right below)

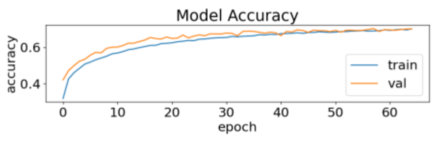


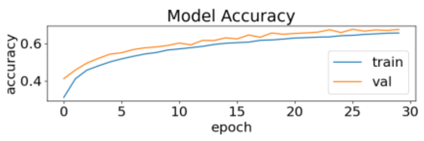


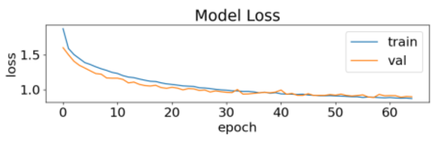


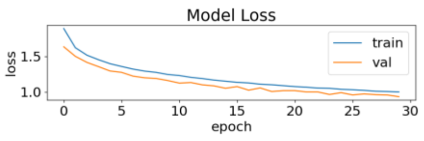


When training data=50K, epoch=30 (left below) & epoch=65 (right below)

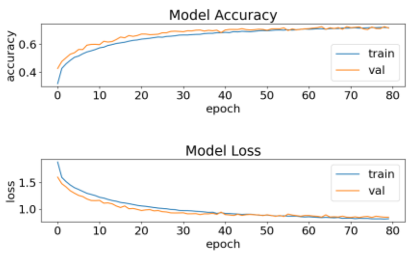


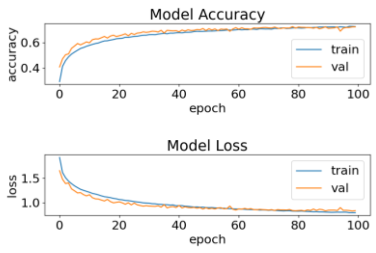






When training data=50K, epoch=80 (left below) & epoch=100 (right below)





By observing the above plots, the gaps between accuracies and losses are to be smaller with increasing epochs. This is reasonable because the model can be iterated more times with increasing epochs. However, it is not suggested to apply too many iterations as it may lead to overfitting.

Appendix

Convolution Code

```
import numpy as np
from PIL import Image


# convolve input array to kernel
def convolve(input_array, kernel):
    # get kernel height and width
    k_h, k_w = kernel.shape[0], kernel.shape[1]

    # add padding to keep output the same size as input
    pad_img = np.pad(input_array, (((k_h - 1) // 2, (k_h - 1) //
2), ((k_w - 1) // 2, (k_w - 1) // 2)))

    # compute convolution
    ret = np.zeros((input_array.shape[0], input_array.shape[1]))
# store result
    for i in range(input_array.shape[0]):
        for j in range(input_array.shape[1]):
            temp_convolution = (kernel * pad_img[i: i + k_h, j: j +
k_w]).sum()
            ret[i, j] = temp_convolution

    return ret


# convolve image input with kernels
def convolution(img):
    img = Image.open(img).resize((200, 200))
    img_rgb = np.array(img.convert('RGB'))
    img_r = img_rgb[:, :, 0]  # single channel

    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(convolve(img_r, kernel1))).show()
    Image.fromarray(np.uint8(convolve(img_r, kernel2))).show()
```

CNN Code

```
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
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()
n = 5000 * 10
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)
```

```
# for w in [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1, 5, 10]:
optional = True
if not optional:
    for w in [1e-4]:
        model = keras.Sequential()
        model.add(Conv2D(16, (3, 3), padding='same',
input_shape=x_train.shape[1:], activation='relu'))

        # ii(c).i replace stride with max pooling
        # model.add(Conv2D(16, (3, 3), strides=(2, 2),
padding='same', activation='relu'))
        model.add(Conv2D(16, (3, 3), padding='same',
activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))  # add max
pooling

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

        # ii(c).i replace stride with max pooling
        # model.add(Conv2D(32, (3, 3), strides=(2, 2),
padding='same', activation='relu'))
        model.add(Conv2D(32, (3, 3), padding='same',
activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))  # add max
pooling 2x2
        model.add(Dropout(0.5))
        model.add(Flatten())
        model.add(Dense(num_classes, activation='softmax',
kernel_regularizer=regularizers.l1(w)))

else:
    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'))
```

```python
    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(1e-4)))

    # try weights [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 0, 1] default 1e-4

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


batch_size = 128
epochs = 100

# record start time
t1 = time.time()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)

# record end time
t2 = time.time()

delta_t = format(t2 - t1, '.4f')
print(f'Time Taken: {delta_t}')

model.save("cifar.model")
plt.subplot(211)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title(f'Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='lower right')
plt.subplot(212)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title(f'Model Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.tight_layout()
plt.show()

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(f'========================Training
Data========================')
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(f'========================Test
Data========================')
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1, y_pred))

# Determine the most common label in the training data
most_common_label = np.argmax(np.bincount(np.argmax(y_train,
axis=1)))

# Evaluate the baseline classifier for training data
baseline_predictions_train = np.full((len(x_train),),
most_common_label)
print("========================Baseline Classifier Training
Data========================")
print(classification_report(y_train1,
baseline_predictions_train))
print(confusion_matrix(y_train1, baseline_predictions_train))

# Evaluate the baseline classifier for test data
baseline_predictions_test = np.full((len(x_test),),
most_common_label)
print("========================Baseline Classifier Test
Data========================")
print(classification_report(y_test1, baseline_predictions_test))
print(confusion_matrix(y_test1, baseline_predictions_test))
```