Machine Learning Assignment Week 8

Name: Masanari Doi

Student number: 19313167

(i)

(a)

```
def convolve(array, kernel):
   if (array.size<kernel.size): # array should be larger than kernel
       return 0
   convolved = []               # array for convolved img
   sum = 0
   size = 0
   kSize = math.sqrt(kernel.size)  # kernel size. if the kernel is 3*3, the size is
3
   kSize = int(kSize)
   arrSize = math.sqrt(array.size) # priginal array size. if it is 3*3, the size is
3
   arrSize = int(arrSize)
   size = arrSize - kSize + 1
   print(size)
   for i in range(size):
       for j in range(size):
           for k in range(kSize):
               for l in range(kSize):
                   sum = sum + array[k+i,l+j]*kernel[k,l]
           convolved.append(sum)
           sum = 0
   convolved = np.array(convolved)
   convoSize = math.sqrt(convolved.size)
   convoSize = int(convoSize)
   convolved = convolved.reshape(-1, convoSize)
   return convolved
```

The code above is used to make a function that takes an array and a kernel and returns a convolved array. Firstly, it checks whether the array size is larger than kernel size as the array should be so. The main algorithm used to convolve the kernel to array is the code below.

```
for i in range(size):
       for j in range(size):
           for k in range(kSize):
               for l in range(kSize):
                   sum = sum + array[k+i,l+j]*kernel[k,l]
           convolved.append(sum)
           sum = 0
```

Take a 5×5 array 1 2 3 4 5 a 3×3 kernel 1 2 3
6 7 8 9 10 4 5 6
2 3 4 5 6 7 8 9
7 8 9 10 11
3 4 5 6 7 and as an example.

In this for-loop, l in range(kSize):, when i=0, j=0, and k=0,
sum = 1*1 + 2*2 + 3*3, and then when k = 1,
sum = 1*1 + 2*2 + 3*3(←from k=0) + 6*4 + 7*5 + 8*6 and then when k = 2,
sum = 1*1 + 2*2 + 3*3 + 6*4 + 7*5 + 8*6 + 2*7 + 3*8 + 4*9, and when the k-loop finishes, the sum is added to the array called convolved that is initialised before the for-loop.
Nextly, when i=0, j=1, and k&l = 0~2,
sum = 2*1 + 3*2 + 4*3 + 7*4 + 5*8 + 9*6 + 3*7 + 4*8 + 5*9. It can be seen that when j changes, the kernel moves to the right side position in the original array.

Additionally, when i=1, j=0, and k&l = 0~2,
sum = 6*1 + 7*2 + 8*3 + 2*4 + 3*5 + 4*6 + 7*7 + 8*8 + 9*9. In this case, when i changes, the kernel moves down a level.
Therefore, we can say that i and j are used for the kernel's horizontal and vertical movement in the array, so when the all 4 for-loop above finishes, the kernel finishes moving in the array in horizontal and vertical way for calculation..

(b)

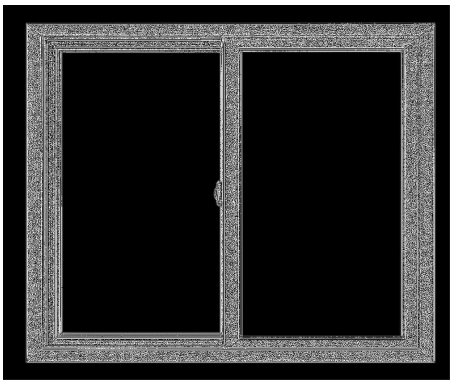
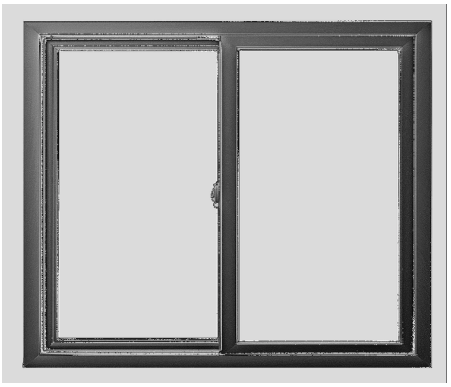
```
im = Image.open("/Users/doimasanari/Desktop/ML/week8/pics.jpg/wk8.jpg")
rgb = np.array(im.convert("RGB"))
r=rgb[:,:,0] # array of R pixels

kernel1 = [-1, -1, -1, -1, 8, -1, -1, -1, -1]
kernel1 = np.array(kernel1)
kernel1 = kernel1.reshape(-1,3)
convolvedR = convolve(r, kernel1)
Image.fromarray(np.uint8(convolvedR)).show()

kernel2 = [0, -1, 0, -1, 8, -1, 0, -1, 0]
kernel2 = np.array(kernel2)
kernel2 = kernel2.reshape(-1,3)
convolvedR = convolve(r, kernel2)
Image.fromarray(np.uint8(convolvedR)).show()
```

The code above is used to load the image and display a convolved image.
rgb = np.array(im.convert("RGB")) and r=rgb[:,:,0] are used to load the image as three RGB arrays and select one of these arrays to work with. Then, convolvedR = convolve(r, kernel1) is used to call the function made in (a). Finally, Image.fromarray(np.uint8(convolvedR)).show() is used to display the convolved image.

The picture above is the original picture. Then, the figure below on the left is the one that kernel 1 is applied to, and for the right one kernel 2 is applied.

It can clearly be seen that the left one is darker than the right one because the kernel1 has more -1 than kernel2.

(ii)

(a)

```
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)))
```

The code above is the downloaded code which illustrates the architecture of ConvNet. I will inspect how each line works as a convolutional layer.

Line 1

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

It is a 2D Convolutional Layer using a 3x3 kernel with 16 output channels. Using “same” padding in convolutional layers means output will be the same size as input. Moreover, this uses ReLU(Reflected Linear Unit) activation function which is a linear function producing input if it is positive, if it is negative it produces 0 . i.e. $g(x) = \{ \begin{matrix} x & x \geq 0 \\ 0 & x < 0 \end{matrix} \}$

Line 2

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

As the same as the Line 1, it is the 2D Convolutional Layer uses a 3x3 kernel with 16 output channels, ‘relu’ activation and padding = “same”. However, the difference this code shows is that stride 2 is used to downsample the image.

Line 3 & 4

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

Line 3 is the 2D convolutional Layer using the same structure of Line 1 except for the number of output channels. It uses 32 channels.
In the same sense, Line 4 is the same as Line 2 except for 32 output channels.

Line 5

```
model.add(Dropout(0.5))
```

This Dropout is a regularisation technique for neural network models used to avoid overfitting in a way that it randomly sets input to 0 during training.

Line 6

```
model.add(Flatten())
```

It flattens the input if the input is output from a convolution layer.

Line 7

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

It uses a nonlinear function, soft max which is a multi-class logistic regression model in order to get final output from Convolutional neural network features. The output is 10x1 as the code given defines num_class = 10. This 10 is corresponding to values 0-9. These values are the elements of the output vector that are the probability of each class.

(b)

(i)

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

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

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

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

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

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 10)                20490

=================================================================
Total params: 37,146
Trainable params: 37,146
Non-trainable params: 0
```

It has 37146 parameters in total. Dense layer has the most parameters, which is 20409.
The accuracy of the test data on the performance on the training data is 0.63, and that of the test data on the performance on the test data is 0.51 as below.

	precision	recall	f1-score	support
0	0.72	0.61	0.66	505
1	0.78	0.70	0.74	460
2	0.53	0.57	0.55	519
3	0.59	0.38	0.46	486
4	0.59	0.48	0.53	519
5	0.49	0.70	0.57	488
6	0.68	0.66	0.67	518
7	0.62	0.71	0.66	486
8	0.74	0.72	0.73	520
9	0.64	0.76	0.70	498
accuracy			0.63	4999
macro avg	0.64	0.63	0.63	4999
weighted avg	0.64	0.63	0.63	4999

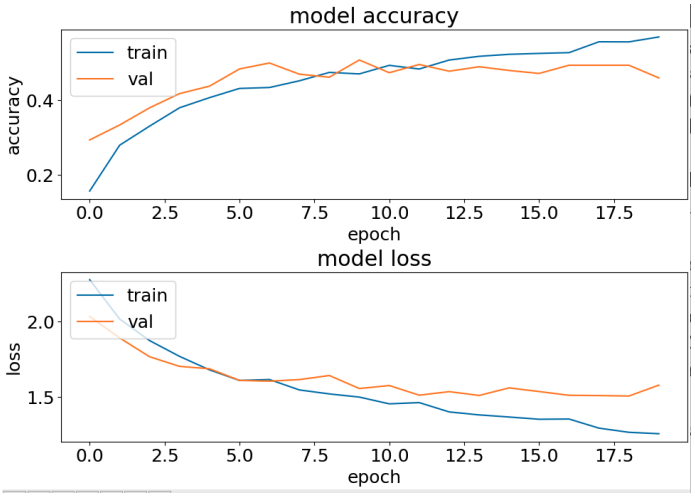
	precision	recall	f1-score	support
0	0.60	0.55	0.57	1000
1	0.71	0.59	0.64	1000
2	0.42	0.45	0.43	1000
3	0.36	0.22	0.27	1000
4	0.45	0.36	0.40	1000
5	0.36	0.52	0.42	1000
6	0.58	0.56	0.57	1000
7	0.51	0.61	0.56	1000
8	0.63	0.64	0.64	1000
9	0.56	0.65	0.60	1000
accuracy			0.51	10000
macro avg	0.52	0.51	0.51	10000
weighted avg	0.52	0.51	0.51	10000

```
dummy = DummyClassifier().fit(x_train, y_train)

ydummy = dummy.predict(x_train)
y_pred = np.argmax(ydummy, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print("for training data, confusion matrix for baseline classifiers that always
predicts the most frequent class")
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))
```

Furthermore, when I compared the performance on the test data and performance on the training data against a simple baseline, I got an accuracy of 0.1 for both performances. The code above is used for this comparison. Comparing the performance with a simple baseline, the training data and test data, 0.63 and 0.51, have a greater (more accurate) percentage.

(ii)



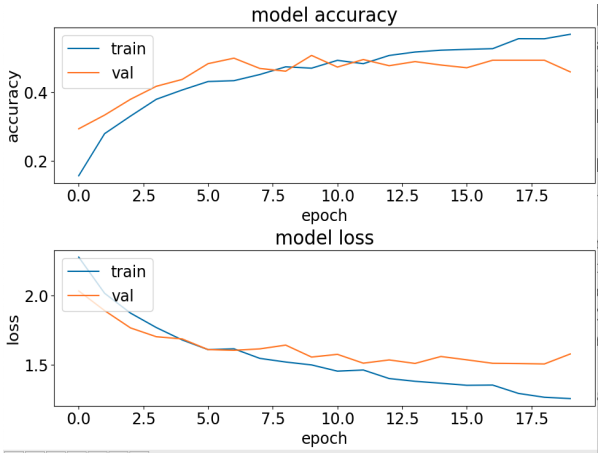
The graph above is plotting the accuracy on the upper graph and loss on the graph below for y-axis, and epoch for both graphs for x-axis. It can be seen that when epoch changes from 0 to 2.5, both accuracy and loss moves dramatically. Since the accuracy increases and the loss decreases, it would be true that it is under-fitting at the first few epochs. In terms of over-fitting, it is shown that after epoch is around 15.0, the accuracy and loss start fluctuates. Therefore, it is probably true that it starts over-fitting and there is no improvement after epochs around 15.0.

(iii)

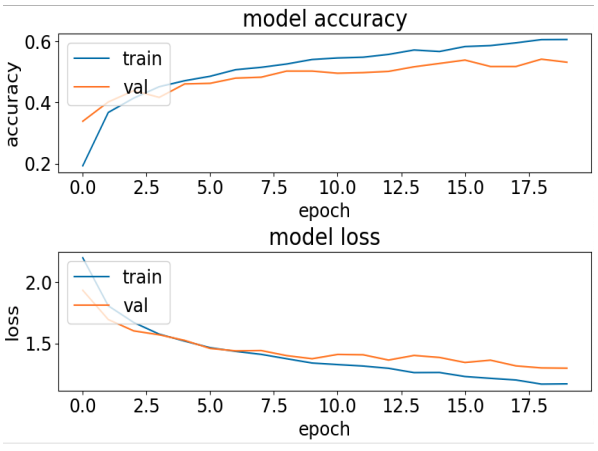
Amount of training data	Time (s)	Accuracy on training data	Accuracy on test data
5000	83	0.64	0.51
10000	159	0.67	0.56
20000	343	0.68	0.61
40000	591	0.73	0.68

From the table above, it can clearly be seen that the more training data are used, the more accurate results are produced. Moreover it would be true that the time taken to train the network is almost proportional to the number of the training data. For example, the time used for 10k training is almost twice as much as the time used for 5k training.
The graphs below show each data's history variable. These graphs illustrate that the data which has more training data produces the more smooth graph.i.e. The gap between train and val is decreased when n is bigger.

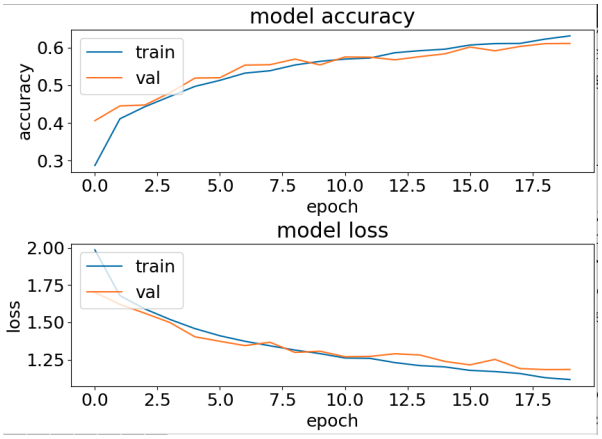
Graphs



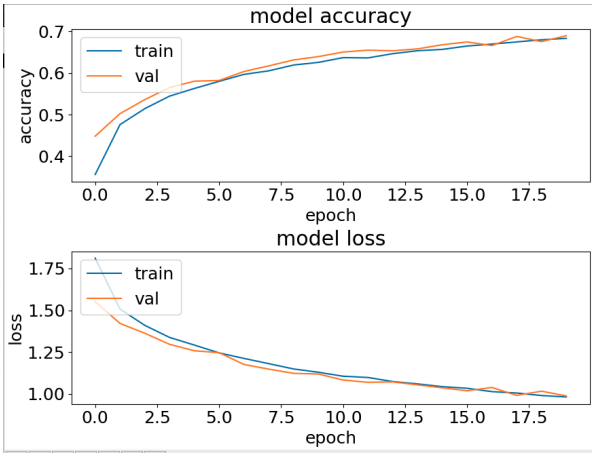
n = 5k



n = 10k



n = 20k

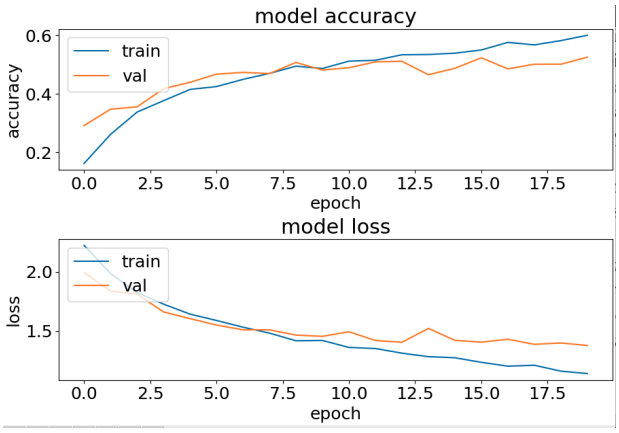


n = 40k

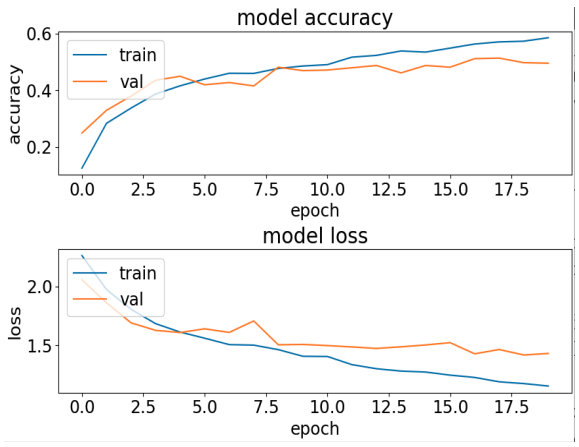
(iv)

L1	Accuracy on training data	Accuracy on test data
0.00000001	0.65	0.50
0	0.61	0.50
0.1	0.36	0.36
1.0	0.24	0.24
10	0.10	0.10

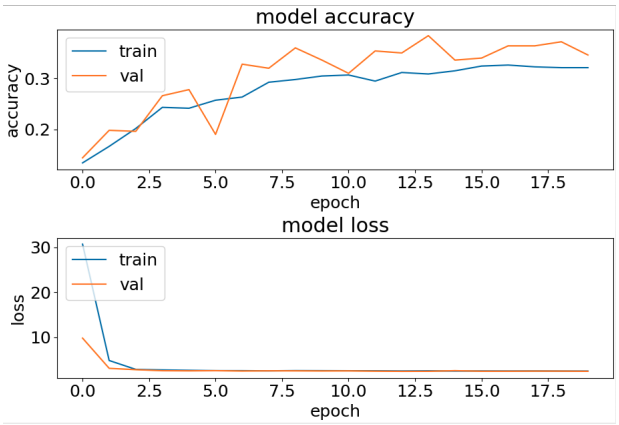
Graphs



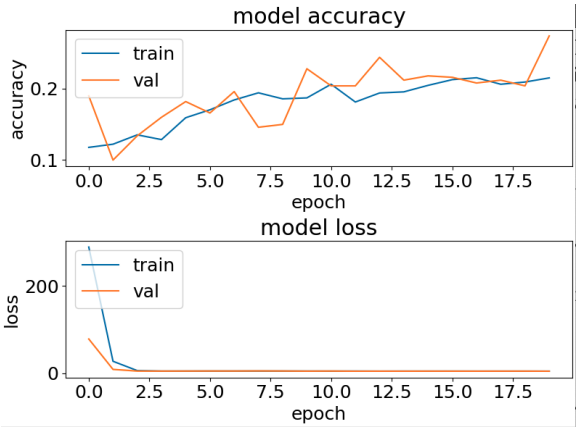
L1 = 0.00000001



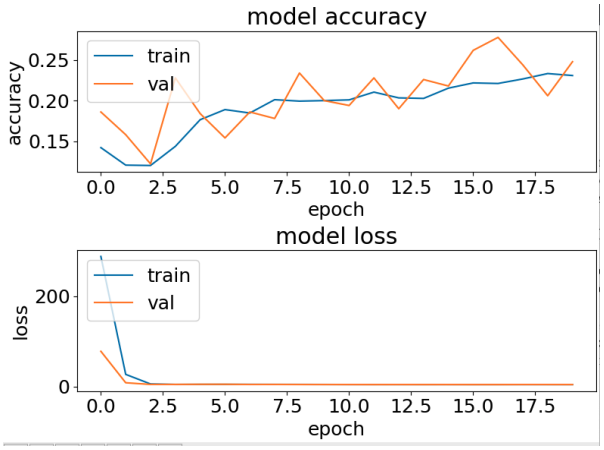
L1 = 0



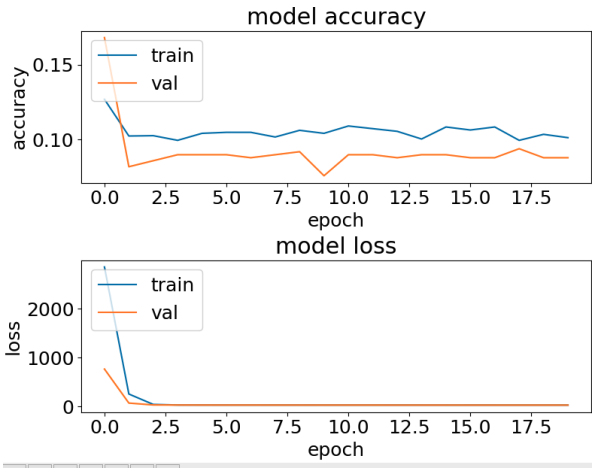
L1 = 0.1



L1 = 1.0



L1 = 1.0



L = 10

From the table and the graph above, it can clearly be seen that the loss decreases as L1 increases although accuracy decreases as L1 increases. It can be true that even though the loss decreases, the accuracy is not improved. Moreover, when seeing the graph L1 = 10, it looks fluctuating all the time except for the first few epochs, and it reaches the accuracy 0.1, which is the same as the baseline output. We can see that the less L1 we use, the more accuracy we get. However, when comparing L1 = 0.00000001 and L1 = 0, their accuracy on the test data is the same, which looks overfitting. Despite the fact that the test data we got from (iii) has the accuracy 0.68 as the highest, the data here has only 0.5 as the highest. Therefore, in terms of managing over-fitting, and accuracy, it is probably true that increasing the amount of training data is more effective than manipulating L1 value.

(c)

(i)

```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

I used the code above to replace the 16 channel strided layer with a 16 channel same layer followed by a (2,2) max-pool layer, and similarly for the 32 channel strided layer. The code for this question is as below. Compared to the code mentioned in (ii) (a), model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu')) and model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu')) were replaced with model.add(MaxPooling2D(pool_size=(2, 2))).

```
model.add(Conv2D(16, (3,3), padding='same',
input_shape=x_train.shape[1:],activation='relu'))
   model.add(MaxPooling2D(pool_size=(2, 2))) # for q c i and ii
   model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
   model.add(MaxPooling2D(pool_size=(2, 2))) # for q c i and ii
   model.add(Dropout(0.5))
   model.add(Flatten())
   model.add(Dense(num_classes,
activation='softmax',kernel_regularizer=regularizers.l1(0.0001)))
```

(ii)

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

max_pooling2d (MaxPooling2D  (None, 16, 16, 16)        0
)

conv2d_1 (Conv2D)            (None, 16, 16, 32)        4640

max_pooling2d_1 (MaxPooling  (None, 8, 8, 32)          0
2D)

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

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 10)                20490

=================================================================
Total params: 25,578
Trainable params: 25,578
Non-trainable params: 0
```

Keras says the ConvNet has 25,578 parameters.

	Time (s)	**Accuracy on training data**	**Accuracy on test data**
Original	83	0.64	0.51
Max-pool layer used	59	0.61	0.51

Compared to the original work, the time taken to train the network and the prediction accuracy on the training and test data is lower. It can be because of the downsampling. It can

be seen that the number of the parameters is lower than the original one, which is 37146 got in Q (ii) (b) (i). Interestingly, there is not much of a difference of accuracy between the original one and max-pooled one.

(d)

```
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'))
```

The code above represents the layers given. It lets us try using a thinner and deeper ConvNet Architecture. The result I got is as below.

Amount of training data	Time (s)	Accuracy on training data	Accuracy on test data
5000	65	0.52	0.45
10000	105	0.58	0.52
20000	221	0.62	0.57
40000	445	0.67	0.62
60000	518	0.68	0.64

The table derived from Q (ii) (b) (iii) to compare this with the table above.

Amount of training data	Time (s)	Accuracy on training data	Accuracy on test data
5000	83	0.64	0.51
10000	159	0.67	0.56
20000	343	0.68	0.61
40000	591	0.73	0.68

It can be seen that, if the tinner and deeper layers are used, the time taken to train the network becomes lower. However, compared to the table we got in Q (ii) (b) (iii), accuracy is lower as well. The question says “When trained for long enough on the full dataset this should achieve prediction accuracy above 70% “. Despite the fact that the data in Q (ii) (b) (iii) needs 40k data to achieve prediction accuracy 70%, the data with more layer one needs more training data.

Appendix

Q(i)

```
from cmath import sqrt
import numpy as np
import math
from PIL import Image

# make a function that takes an n × n array and a k × k kernel, convolves the
kernel to the input array and returns the result
def convolve(array, kernel):
   if (array.size<kernel.size): # array should be larger than kernel
       return 0
   convolved = []               # array for convolved img
   sum = 0
   size = 0
   kSize = math.sqrt(kernel.size)  # kernel size. if the kernel is 3*3, the size is
3
   kSize = int(kSize)
   arrSize = math.sqrt(array.size) # original array size. if it is 3*3, the size is
3
   arrSize = int(arrSize)
   size = arrSize - kSize + 1       # convolved array size is calculated from
original array size - kernel size + 1
   print(size)
   for i in range(size):
       for j in range(size):
           for k in range(kSize):
               for l in range(kSize):
                   sum = sum + array[k+i,l+j]*kernel[k,l]
           convolved.append(sum)
           sum = 0

   convolved = np.array(convolved)
   convoSize = math.sqrt(convolved.size)
   convoSize = int(convoSize)
   convolved = convolved.reshape(-1, convoSize)
   return convolved

# the comment below was used to check if the conv function works correctly

# array = [1,0,1,0,-1,1,0,1,0,-1,1,0,1,0,-1,1,0,1,0,-1,1,0,1,0,-1]
# kernel1 = [1,2,1,1,1,3,2,1,1]
# array = np.array(array)
# array = array.reshape(-1,5)
# kernel1 = np.array(kernel1)
# kernel1 = kernel1.reshape(-1,3)
# print(array)
# print(kernel1)
# newArr1 = convolve(array, kernel1)
# print(newArr1)

# kernel2 = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2]
# kernel2 = np.array(kernel2)
# kernel2 = kernel2.reshape(-1,4)
# print(kernel2)

# newArr2 = convolve(array, kernel2)
# print(newArr2)

im = Image.open("/Users/doimasanari/Documents/ML/week8/pics.jpg/wk8.jpg")
rgb = np.array(im.convert("RGB"))
```

```
r=rgb[:,:,0] # array of R pixels

print(r)
print(r.size)

kernel1 = [-1, -1, -1, -1, 8, -1, -1, -1, -1]
kernel1 = np.array(kernel1)
kernel1 = kernel1.reshape(-1,3)
print(kernel1)
convolvedR = convolve(r, kernel1)
Image.fromarray(np.uint8(convolvedR)).show()

kernel2 = [0, -1, 0, -1, 8, -1, 0, -1, 0]
kernel2 = np.array(kernel2)
kernel2 = kernel2.reshape(-1,3)
print(kernel2)
convolvedR = convolve(r, kernel2)
Image.fromarray(np.uint8(convolvedR)).show()

# sum = sum + array[j+i,k]*kernel[j,k]
# sum = sum + array[j,k+1]*kernel[j,k]
```

Q (ii) (a) ~ (c)

```
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 sklearn.dummy import DummyClassifier


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 = 5k or 10k or 20k or 40k
n=5000
# n=10000
# n=20000
# n=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)

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(MaxPooling2D(pool_size=(2, 2))) # for q c i and ii
   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(MaxPooling2D(pool_size=(2, 2))) # for q c i andii
   model.add(Dropout(0.5))
   model.add(Flatten())
   model.add(Dense(num_classes,
activation='softmax',kernel_regularizer=regularizers.l1(0.0001)))
   model.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
   model.summary()

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

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

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

dummy = DummyClassifier().fit(x_train, y_train)

ydummy = dummy.predict(x_train)
y_pred = np.argmax(ydummy, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print("for training data, confusion matrix for baseline classifiers that always
predicts the most frequent class")
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))
```

```
ydummy = dummy.predict(x_test)
y_pred = np.argmax(ydummy, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print("for test data, confusion matrix for baseline classifiers that always
predicts the most frequent class")
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))
```

Q (ii) (d)

```
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 sklearn.dummy import DummyClassifier


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 = 5k or 10k or 20k or 40k or 60k
# n=5000
n=10000
# n=20000
# n=40000
# n=60000
x_train = x_train[1:n]; y_train=y_train[1:n]
#x_test=x_test[1:500]; y_test=y_test[1:500]

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
print("orig x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

use_saved_model = False
if use_saved_model:
    model = keras.models.load_model("cifar.model")
else:
    model = keras.Sequential()
    model.add(Conv2D(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.compile(loss="categorical_crossentropy", optimizer='adam',
metrics=["accuracy"])
   model.summary()

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

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

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

dummy = DummyClassifier().fit(x_train, y_train)

ydummy = dummy.predict(x_train)
y_pred = np.argmax(ydummy, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print("for training data, confusion matrix for baseline classifiers that always
predicts the most frequent class")
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

ydummy = dummy.predict(x_test)
y_pred = np.argmax(ydummy, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print("for test data, confusion matrix for baseline classifiers that always
predicts the most frequent class")
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))
```