**Machine Learning – Week 8 Assignment**

1. (a)

I first extract the shape attribute of the input array to get its size. We know it’s an n x n array so column number and row number will be equal. The same applies for the k x k kernel. I subtract the size of the kernel from the input array to compute the size of the convolved output array. Next I loop through each row and for each column in that row I apply the convolution and store the summation result in the convolved output array. As it was stated that the function takes only an input array and kernel as parameters, I assumed that the padding for the convolution was 0 and that the stride was 1.

I took the two examples in the slides, but added a column to each to make column and row size equal. I then used these as inputs to test my solution.

Graphical user interface

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

My output matches indicating the convolution works.

A picture containing text, electronics

Description automatically generated

(b)

I took a picture of a pentagon from the internet and resized it to be 200x200 pixels.

**Input image: Input kernel:**

Shape, polygon

Description automatically generated A picture containing object, clock, watch

Description automatically generated

Next, I load the image as three RGB arrays as asked and pass it to my convolution function along with the first specified input kernel. The output can be seen below.

Shape

Description automatically generated

The edges of the pentagon in the convolved image appear noisy and there is a black background using this kernel.

I applied the same process using same input image and the second kernel specified.

**Input image: Input kernel:**

Shape, polygon

Description automatically generated A picture containing object, clock, watch

Description automatically generated

The output can be seen below.

Shape

Description automatically generated

The edges of the pentagon in the convolved image appear clearer than in the first, with no noise and there is a white background using this kernel.

1. (a)

The figures needed to answer this question are shown in the code given to us below.

A screenshot of a computer

Description automatically generated

As we can see as we go through the layers, the number of channels increases and the height decreases. This architecture is similar to the one given in the notes:

**(Conv × n + Maxpool) × m**

Where n is the number of layers and m is the number of times the process is repeated. Instead of max pooling, down sampling using strides of length 2 is applied. We can see from the code that the block of convolving using strides of length 2 repeated twice, meaning m = 2.

The input layer has a in shape of 32. 16 is passed to the filter parameter meaning there are 16 output filters. It uses a 3 x 3 kernel for the convolution.

The next convolution layer is for down sampling. It is passed 16 for the filter parameter meaning there are 16 output filters. It again uses a 3 x 3 kernel for the convolution. It uses a stride of 2 which will help reduce the size of the output.

The next convolution layer then had 32 output filters and uses a 3 x 3 kernel. Down sampling is applied again, like before with a stride of 2.

The next layer is a dropout layer which applies regularisation with a rate of 0.5. The output stays the same size here.

We must flatten the last convolved output before it can be used as input to the dense layer. This is the flatten layer.

The last layer is the dense layer/fully connected layer. This is the softmax layer which maps the features to the final output. There are 10 classes which is passed to the Dense constructor, so the number of output nodes will be 10.

This leaves us with 7 layers in total.

(b)

(i)

The resulting output from running the code can be seen below:

Text

Description automatically generated

Here we can see that there are 37,146 parameters in total with all of them being trainable parameters. The layer with the most parameters is the dense/fully connected layer. If we have an input with m elements and n outputs the dense layer will have m x n parameters. We know that each output is a function of a weighted sum of all of the inputs, meaning we need all inputs as parameters as well as all outputs, as there can be more than one output. This means this figure will always be quite large as it grows rapidly.

For the training data the accuracy was around 63%.

Graphical user interface, text, application

Description automatically generated

For the test data, the accuracy was around 49/50%.

Graphical user interface, application

Description automatically generated

The training data has a higher accuracy by around 13% compared to the test data, but this is expected as the model should make more accurate predictions on the training data compared to other inputs, otherwise the training data is not well suited to the model.

I used a most common label baseline classifier which predicted with an accuracy of 10% as can be seen below.

Graphical user interface, text, application

Description automatically generated

Comparing this to the test data results we can see that training even a simple model can go a long way when it comes to predictions. There is an increase of 40% in correct predictions.

(ii)

For each epoch run, every sample in the training data can update the parameters of the model. A full run through of the dataset is performed. Epoch’s can be used to generate learning curves like shown above, in order to distinguish whether a model overfits or underfits.

The plot giving the models accuracy and loss can be seen below:

Chart, line chart

Description automatically generated

An argument could be made saying that overfitting may be occurring from the loss graph as the training loss decreases after each iteration. However, it is not clear that the validation loss will begin to increase as more epoch’s occur.

The training loss does decrease and seems to stable out towards the last few epoch’s, however it still seems to trend downwards. The validation loss decreases and does not fluctuate too much while keeping close enough to the training loss curve. This could indicate a good fit. The accuracy is quite high for both the validation and training curves, which leads me to conclude that there may be a small amount of overfitting occurring but to me, it seems like an good fit.

(iii)

For 5k training points:

This was the initial amount of points, which yielded an accuracy of 63% for the training data and 50% for the test data. The shortest epoch time was 3 seconds and the longest was 6 seconds. The output graph can be seen above in part (ii).

For 10k training points:

This figure yielded an accuracy of 66% for the training data and 56% for the test data. The shortest epoch time was 5 seconds and the longest was 10 seconds. The output graph can be seen below.

Chart, line chart

Description automatically generated

For 20k training points:

This figure yielded an accuracy of 67% for the training data and 61% for the test data. The shortest epoch time was 15 seconds and the longest was 17 seconds. The output graph can be seen below.

Chart, line chart

Description automatically generated

For 40k training points:

This figure yielded an accuracy of 73% for the training data and 68% for the test data. The shortest epoch time was 29 seconds and the longest was 33 seconds. The output graph can be seen below.

Line chart

Description automatically generated with medium confidence

It can be seen clearly that as we increase the number of datapoints being used, the accuracy of the model increases. For 5k training points the accuracy on the test data was 50%. This increased slowly with each addition of the training points. The massive increase from 50% to 68% for 40k training points shows that the more training data . This is to be expected as the more training data we feed the model, the more sources it has to draw accurate predictions from. The accuracy of the model when it came to the training data also increased, but not by as much as the test data. It began at 63% for 5k datapoints and increased to 73% for 40k datapoints. This shows an increase in accuracy of 18% for the test data and 10% for the training data.

The time taken to train the model increases as the number of datapoints increases. For 5k datapoints the average epoch was around 4.5 seconds. This increased to 31 seconds when using the 40k datapoints. This is expected as it must take the extra data into account when training.

From the loss graphs generated, initially I thought overfitting may begin to occur as we increase the number of datapoints. I made this assumption because I thought the validation loss curve may increase. However in the last graph using 40k datapoints we can see that the 2 curves for the training and validation have a tiny gap and seem to follow the same trend. This indicates a well fitted model.

(iv)

For L1 = 1:

The training data accuracy was 10% and the test data accuracy was 10%.

For L1 = 0.1:

The training data accuracy was 35% and the test data accuracy was 34%.

For L1 = 0.01:

The training data accuracy was 48% and the test data accuracy was 44%.

For L1 = 0.001:

The training data accuracy was 57% and the test data accuracy was 48%.

For L1 = 0.0001:

The training data accuracy was 63% and the test data accuracy was 50%.

For L1 = 0.00001:

The training data accuracy was 65% and the test data accuracy was 49%.

For L1 = 0:

The training data accuracy was 64% and the test data accuracy was 51%.

The L1 penalty clearly plays a big effect on the models predictions. For high values close to 1 for the penalty, it is too large as it means the model takes fewer risks. The weight 0 seems to be the best fit as it has the highest accuracy but this may not be smart as no penalty is applied. We want a lower value for the penalty closer to 0 but not exactly 0. This helps the model produce more accurate results and adapt to the training data more efficiently.

For managing overfitting, the number of datapoints used is important. However a combination of both the L1 penalty and an optimal amount of datapoints would be key when attempting to optimise the models fit. L1 regularisation is used to combat overfitting. This is done by shrinking the parameters towards 0 which can help eliminate some features.

(c)

(i)

I added the following line of code which specified a max (2,2) pool layer:

model.add(MaxPooling2D(pool\_size=(2, 2)))

for max pooling instead of the original 2 stride down sampling line:

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

I added this change for each convolution. The code can be seen in the appendix.

(ii)

The accuracy of this model on the training data was 60% and on the test data was 51%. The test data accuracy here is 1% higher than using the down sampling method but the training data percentage is down by 3%. This is not show much of a difference between the two methods.

The model parameters can be seen below:

Text

Description automatically generated

There are 25,578 total parameters and all of them trainable once again. The dense layer is again the one with the most parameters with 20490. The input layer had a parameter value of 448. conv2d\_1 had 4640 parameters. It can be noted that there are less parameters than with the previous method. The total number of parameters dropped from 37,146 to 25,578.

Max pooling layers do not have any parameters as we can see clearly. They are used to reduce computational load and they reduce parameters in doing so. It involves picking the maximum section of each feature map.

The longest amount of time taken on an epoch was 4 seconds and shortest amount of time was 2 seconds. If we compare this to our initial use of down sampling where we recorded a longest epoch time of 6 seconds and shortest of 3 seconds, we can see there is a speed increase with our second solution. This is most likely because there are less parameters for the model to learn from meaning the computational load is less.

**Appendix:**

**Part i):**

import numpy as np

from PIL import Image

# Assume padding = 0 and stride = 1

def convolution(nArray, kKernal):

k = kKernal.shape[0]

n = nArray.shape[0]

print("K = ",k)

print("N = ",n)

resultSize = n - k + 1

output = np.zeros((resultSize, resultSize))

for i in range(resultSize):

for j in range (resultSize):

toConvolute = nArray[i: (i + k), j: (j + k)]

convoluted = toConvolute \* kKernal

output[i][j] = convoluted.sum()

return output

# testInput = [10,10,10,0,0,0],[10,10,10,0,0,0],[10,10,10,0,0,0],[10,10,10,0,0,0],[10,10,10,0,0,0],[10,10,10,0,0,0]

# testKernel = [1,0,-1],[1,0,-1],[1,0,-1]

# nArray = np.array(testInput)

# kKernel = np.array(testKernel)

# convolved = convolve2D(nArray, kKernel)

# print(convolved)

# testInput = [0,0,0,10,10,10],[0,0,0,10,10,10],[0,0,0,10,10,10],[0,0,0,10,10,10],[0,0,0,10,10,10],[0,0,0,10,10,10]

# nArray = np.array(testInput)

# kKernel = np.array(testKernel)

# convolved = convolve2D(nArray, kKernel)

# print(convolved)

im = Image.open('test.jpg')

im = im.resize((200,200))

rgb = np.array(im.convert('RGB'))

r = rgb[:,:,0]

# testKernel = [-1,-1,-1],[-1,8,-1],[-1,-1,-1]

# kKernel = np.array(testKernel)

# output = convolution(r, kKernel)

# Image.fromarray(np.uint8(output)).show()

testKernel = [0,-1,0],[-1,8,-1],[0,-1,0]

kKernel = np.array(testKernel)

output = convolution(r, kKernel)

Image.fromarray(np.uint8(r)).show()

**Part ii):**

import statistics

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers, regularizers

from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization

from keras.layers import Conv2D, MaxPooling2D, LeakyReLU

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.utils import shuffle

import matplotlib.pyplot as plt

plt.rc('font', size=18)

plt.rcParams['figure.constrained\_layout.use'] = True

# Model / data parameters

num\_classes = 10

input\_shape = (32, 32, 3)

# the data, split between train and test sets

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.cifar10.load\_data()

n = 5000

x\_train = x\_train[1:n]

y\_train = y\_train[1:n]

#x\_test=x\_test[1:500]; y\_test=y\_test[1:500]

# Scale images to the [0, 1] range

x\_train = x\_train.astype("float32") / 255

x\_test = x\_test.astype("float32") / 255

print("orig x\_train shape:", x\_train.shape)

# convert class vectors to binary class matrices

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

use\_saved\_model = False

if use\_saved\_model:

model = keras.models.load\_model("cifar.model")

else:

model = keras.Sequential()

model.add(Conv2D(16, (3, 3), padding='same',

input\_shape=x\_train.shape[1:], activation='relu'))

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

model.add(MaxPooling2D(pool\_size=(2, 2)))

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)))

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))

# most common label

most\_common\_label = statistics.mode(y\_train1)

most\_common\_result = most\_common\_label \* np.ones(y\_train1.shape)

print(classification\_report(y\_train1, most\_common\_result))

print(confusion\_matrix(y\_train1, most\_common\_result))