**CMPT 412**

**Project1**

**PART 1: Forward Pass**

The following results were obtained by running **test\_components.py**

**Q 1.1: Inner product layer**

Ans: In order to complete the inner\_product.py and from a completely connected layer, I implemented the following equation **f (x) = W x + b** within the ***inner\_product\_forward(input,layer,param)*** function. This is the matrix multiplication of the weights W with the input followed by the addition of the biases, to obtain the desired output. As a result of the well functioning layer we obtain the result as shown in fig1.

A screenshot of a computer game

Description automatically generated

**FIG1: INNER PRODUCT LAYER**

**Q1.2: Pooling Layer**

In this part we fill in the code within **pooling\_layer.py**. The basic idea is to reduce the size of the feature maps by replacing a local region of the feature map with aggregated stats, and this was implemented by the use of the following formula.The result is shown below(fig2).

A group of squares with different colors

Description automatically generated

**FIG 2: POOLING LAYER**

**Q1 1.3 Convolution Layer**

In this part of the assignment, I filled in code within the **conv\_layer.py**. We mainly utilized the function provided to us ***im2col\_conv\_batch(*** ) which enabled in making the computation a lot faster. This was then followed by the implementation of the following formula **f (X, W, b) = X ∗ W + b** to obtain the desired output. The results are shown below in fig3 and fig4.

A group of squares with different colors

Description automatically generated

**FIG 3 Convolution1**

A group of black squares with different colored squares

Description automatically generated

**FIG 4: Convolution 2**

**Q 1.3 ReLu:**

For ReLu we used the following formula **f (x) = max(x, 0)** and implemented it within **relu.py**

**PART2: Back Propagation**

**Q 2.1 Relu:**

Implemented within **relu.py** under the ***relu\_backward(output, input\_data, layer)*** function.

**Q 2.2** I**nner Product layer:**

Implemented within **inner\_product.py** under the ***inner\_product\_backward(output, input\_data, layer, param).***

**PART3: Training**

**Q 3.1 Training**

After running **train\_lenet.py** we obtain the following accuracy for **2000** iterations.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The above two snippet and the copy of the output depict that we obtained a test accuracy of **95%.**

**Q 3.2 Test the network:**

After modifying the test\_network.py I was able to obtain the following confusion matrix.

A screenshot of a computer

Description automatically generated

It is important to note that the confusion matrix may differ slightly every time we run test\_network.py The image provided above depicts the output I received when I ran it. In this image we can notice that the most confusing pairs are the following:

**8 and 5**: It seems as though the network had a hard time differentiating the 5’s from 8. One reason for this is probably because the curve at the bottom of 5 is very similar to 8 and if the rest of the strokes of 5 aren’t written with care it may result in such a confusion.

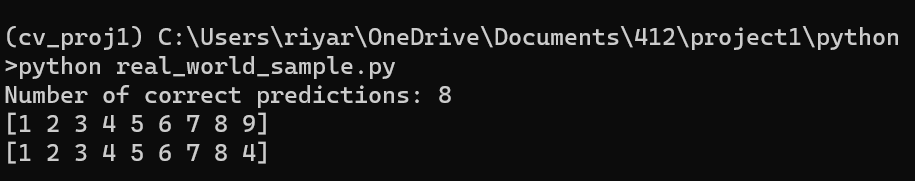
**9 and 4**: It seems that the network identified 4 as 9 multiple times, this is possibly because there are two ways of writing 4, with the two lines connecting and the top and without connecting. I believe that when it was written with the two straight lines connected at the top the network confuses it for a 9 as they may foster a similar look.

**Q 3.3 Real world testing:**

For this part I created a new python script called **real\_world\_sample.py** . I stored my samples within the images folder, under another folder called samples. I made sure to resize my images to (28,28) prior to saving them in my folder.

The following are the samples I used in this part:





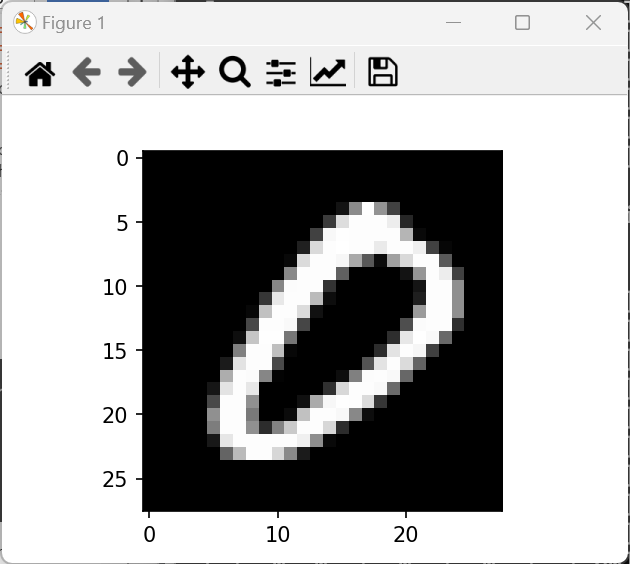
The above snippet is the output I received, the first list is the actual list of elements and the second list depicts the predictions. The output shows that we got 8 out of 9 correct, this shows that the network has a good accuracy. And it confirms the confusion pair shown above that the network may get confused between **9 and 4**.

**PART4: Visualization**

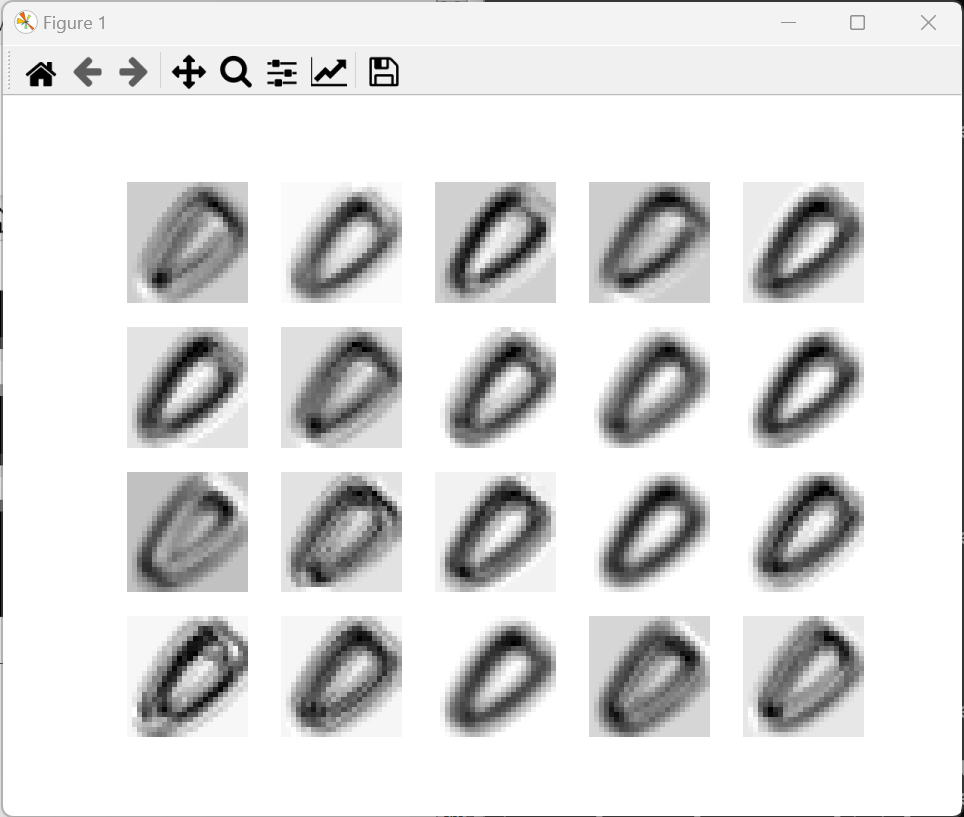
**Q 4.1:**

For this part of the assignment we implemented the code within **vis\_data.py** and obtained the following outputs.

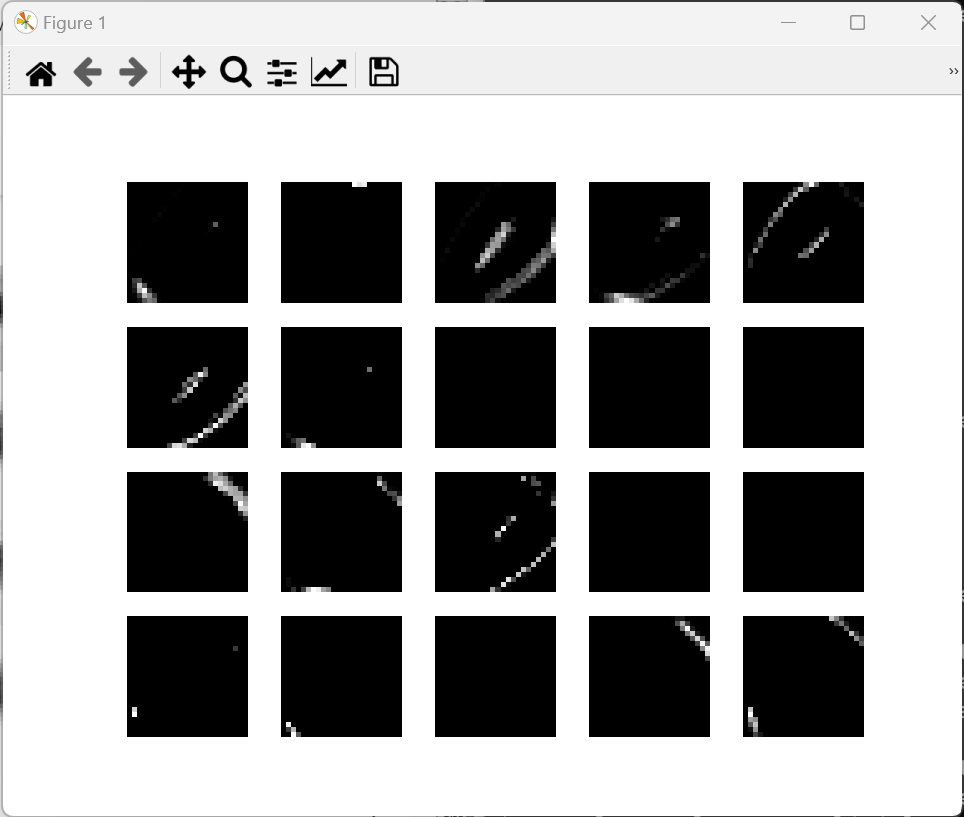
**Input :**



**Convolution layer:**



**ReLu Layer:**



**Q 4.2 Explanation:**

In the above images we notice that the convolution layer as compared to the original input image has certain aspects filtered out and blurry but still portrays the digits in a legible manner. We can also notice that grayscale shades seem to be inverted, with the number being darker as opposed to the background, this is probably due to transpose.

On the other hand we notice that in the ReLu layer we notice just a few specs, this is all of the aspects that got filtered out(negative pixels) in the convolution layer. Thus depicting that our network has learnt to identify the essential features such as the curves and the straight lines within the different layers.

**PART5: Image Classification**

In this part of the assignment I created a new python script called **ec.py** within the python folder. In this script, I mainly utilised the inbuilt **OpenCV** python functions to implement my code. I utilised the inbuilt functions to read the image and convert it to grayscale. I then used **cv2.threshold( )** to binarize the image and lastly, I utilized **cv2.findContours( ),** to find the individual numbers within the single image. For each image I had to tweak the values of the **bounding box** separately, however all the changes were being applied to each bounding box within the image. This did make it slightly troublesome, as each contour found by the function were of different shapes and some did not even capture the entire digit and required tweaking. I also noticed that **cv2.findContours( )** did not find the numbers in order, and so I had to input the individual bounding boxes one by one, to ensure that the prediction was made in order of the contours found.

On running ec.py we get the following images are my results of this part:

image1:

A screenshot of a computer

Description automatically generated

image2:

A close up of a number

Description automatically generated

Image3:

A close up of a black letter

Description automatically generated with medium confidence

Image4:

A screenshot of a number

Description automatically generated