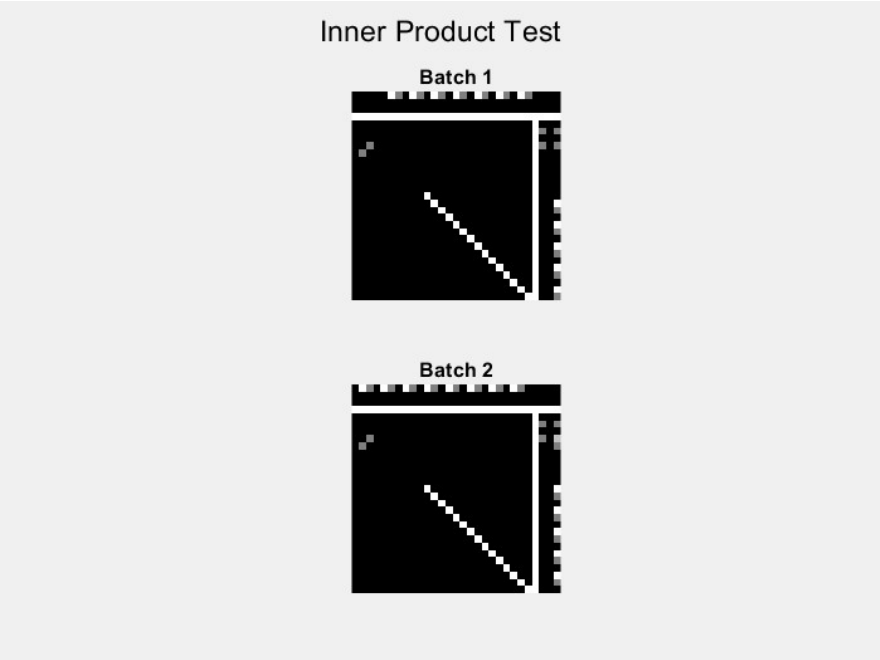
Q 1.1 Inner Product Layer

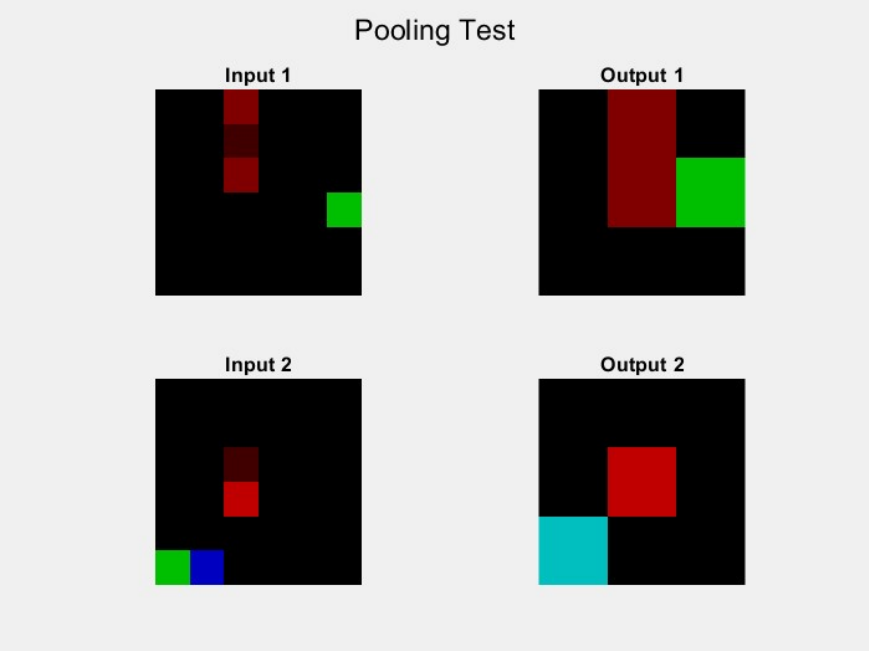
[output] = inner_product_forward(input, layer, param)



The input data given to the function is initially processed by extracting dimensions from the input and parameter structs such as the number of input neurons, batch size, and number of output neurons. The forward calculation is carried out by adding the bias to the dot product of the input data and the weight matrix. The layer's output is obtained by a linear combination of the inputs and the weight. The function returns a struct containing the output data.

Q 1.2 Pooling Layer

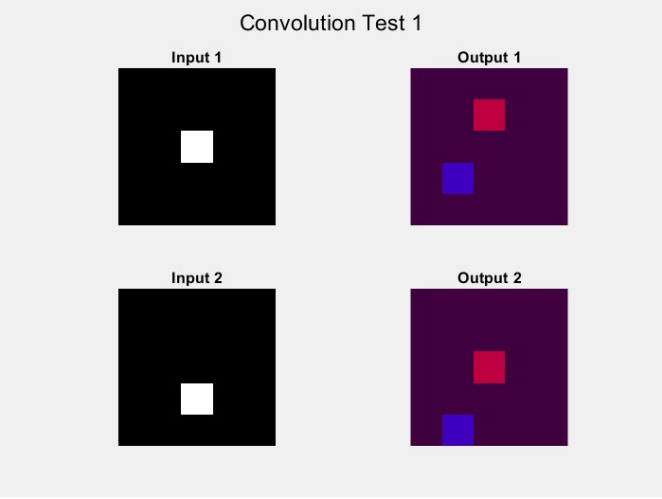
[output] = pooling_layer_forward(input, layer)

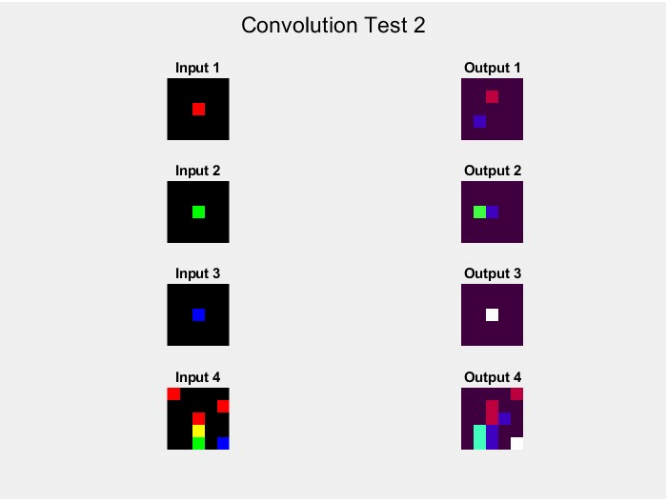


The function takes input data and layer information and returns an output struct with the output data. It extracts kernel size, padding, stride, and dimensions from the input layer struct, computes the output height and width, creates the output struct and iterates over each image in the batch. It reshapes the input data, performs max pooling on it, stores the output image in a column vector in the output struct, which is returned. The implementation is beneficial in the CNN's forward pass as it aids in retrieving the pooling layer's output using the max pooling approach.

Q 1.3 Convolution Layer

[output] = conv_layer_forward(input, layer, param)





The function conv_layer_forward() is a convolutional layer implementation that I used in the forward pass of a CNN. Its main purpose is to perform the computations required to obtain the output of the convolutional layer. The function takes in the input data, layer information and the parameter struct as input and returns the output struct containing the output data. The function extracts the height, width, number of channels and batch size of the input from the input struct. It also extracts the kernel size, padding, stride, and number of filters from the layer struct. Using these values, it then calculates the output shape. It then creates a struct for the input, with the same shape as the original input. The function then iterates over each batch and reshapes the input into columns to prepare for convolution. It then performs a matrix multiplication between the reshaped input and the weights, adds the bias and reshapes the output back into a column vector and stores it in the output struct. Finally, the function sets the output shape, height, width, channel, and batch size, and returns the output struct.

Q 1.4 ReLU

[output] = relu_forward(input)

The function returns the output struct containing the output data after accepting the input data as input. The function applies the ReLU function elementwise to the input data and transfers the input shape to the output struct. ReLU is defined as max(x,0), which indicates that it maintains each element in the input data if it is larger than zero and sets it to zero otherwise. Height, width, channel, and batch size are also assigned same values for the output struct as they are for the input struct.

Q 2.1 ReLU Backwards

The backward pass for the ReLU (Rectified Linear Unit) activation function takes the output of the forward pass, which includes the activations of the current layer and the downstream derivatives, and the input to the forward pass as its inputs. It doesn't take any parameters as ReLU layer doesn't have any parameters. It then calculates the input gradients using the output.diff and the input.data. It multiplies the output.diff with the part of the input.data that is greater than 0. It then assigns the result to input_od which is the output of the function.

Q 2.2 Inner Product layer

The function inner_product_backward() takes in the output data, input data, layer struct, and parameters struct as inputs and returns the parameter gradients and input_od(input data gradient) as output.

The function initializes the input_od as the same size as the input data, bias gradient as the same size as the bias parameter and the weight gradient as the same size as the weight parameter. The function then loops through each input in the batch and accumulates the gradient of the bias parameter. Next, it computes the gradient of the weight parameter by taking dot product of input data and transpose of output data diff. Then, it computes the gradient of the input by taking dot product of weight parameter and output data diff.

Q 3.1 Training

The actual network updates are carried out by the script train lenet.m, which also specifies the optimization settings. The network is loaded and trained using this script for 3000 iterations. On my initial run the training of data was taking too long to finish each iteration, and so after a few optimizations and refactoring of my previous functions, the script train_lenet.m finished all of its iterations (3000) in roughly 40 minutes with a 97% accuracy.

Q 3.2 Test the network

```
The top two confused pairs of classes are class 1 with a confusion rate of 60 and class 7 with a confusion rate of 55
    54     0     0     0     0     1     0     0     0     0
     0    60     0     0     0     0     0     1     0     0
     1     0    41     1     0     0     0     2     0     0
     0     0     0    49     1     1     0     0     1     0
     0     0     0     0    48     0     0     0     0     2
     0     0     0     1     0    51     0     0     0     0
     0     0     0     0     0     0    46     0     0     0
     0     1     1     0     0     0     0    55     0     1
     0     0     0     0     0     0     0     0    40     0
     0     0     0     0     0     0     0     0     0    41
```

The test network.m script runs a pre-trained convolutional neural network (CNN) on the MNIST dataset. The CNN architecture and pre-trained weights are put into the script, and the test dataset is routed through the network. A confusion matrix is used to assess the network's performance and identify the pairings of classes that the network has the greatest trouble distinguishing. The script is run several times, and the confusion matrix is averaged to achieve a more accurate result. Finally, the top two confused pairings of classes are presented, along with their corresponding confusion rates. In my case, I have found 1 and 7 with the highest confusion rate. However, with each run I would get different max confusion pairs but in most cases 1 is a common digit that I had gotten from multiple runs.

Q 3.3 Real-world testing

```
The predicted class for image 1.png is 4
The predicted class for image 3.png is 8
The predicted class for image 4.png is 8
The predicted class for image 5.png is 3
The predicted class for image 6digit.png is 8
The predicted class for image 8.png is 0
```

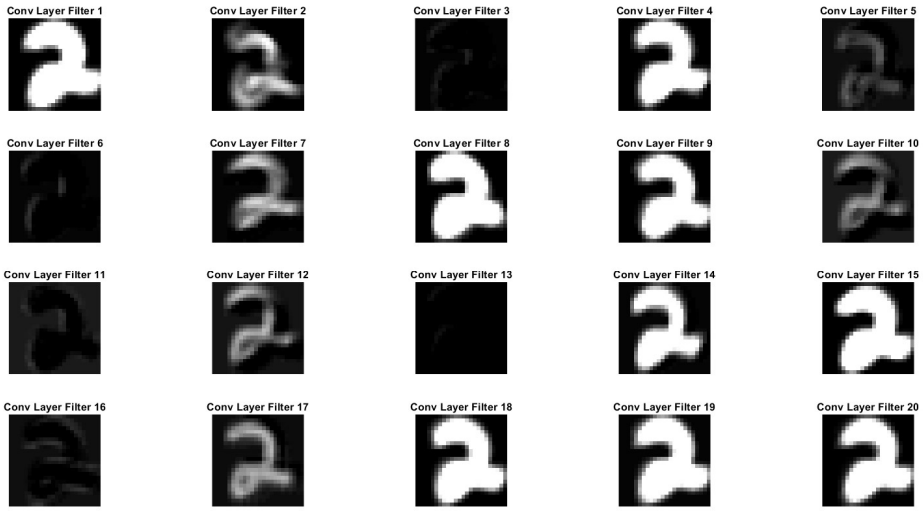
The script in real_pred.m loads all images (downloaded 6 images of digits) in the current folder, normalizes the pixel values of the images, reshapes the images to 2D arrays, and runs them through a convolutional neural network (CNN) using the function "convnet_forward". It then uses the function "max" to get the class prediction of the image with the highest probability and prints out the name of the image and the class prediction.

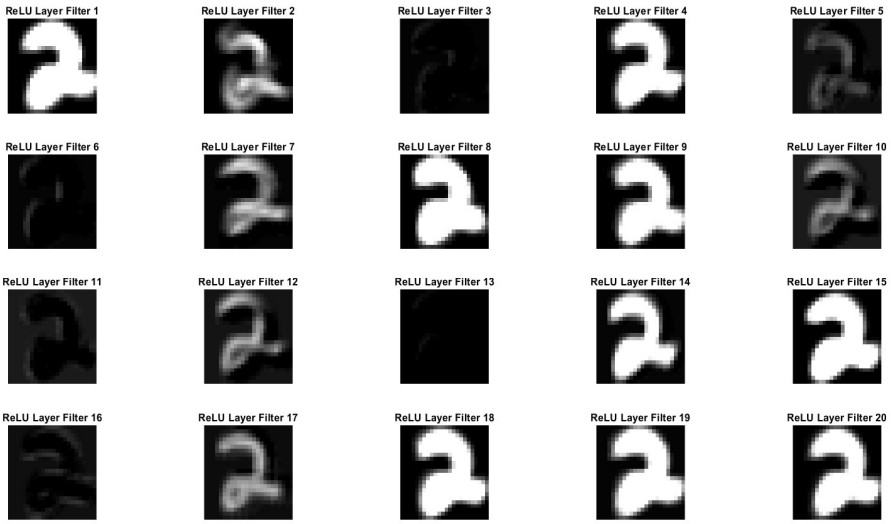
The predictions from the code might be wrong from the actual images provided as the images in the current folder are not similar to the images that the network was trained on. Also, the network may not have seen similar images during training and thus may not be able to generalize well to new images. Also, the image preprocessing steps (resizing, converting to grayscale, and normalizing) might have altered the image in a way that negatively impacts the network's ability to classify the image correctly. Additionally, if the image has a different size than the input size of the network, the reshaping step may cause loss of information which can also lead to incorrect predictions.

Visualization

Q 4.1

Original Image taken from dataset





The script vis.m loads a pre-trained CNN and the MNIST dataset, then it selects an image (2 in our case) from the test set, passes it through the network and visualizes the output of the second layer (CONV) and the third layer (ReLU). The script creates two figures, one for each layer, it loops over the number of filters and displays the output of each filter. The script is used to visualize the features learned by the network.

As shown in the images the two layers looks very similar to one another this is because the ReLU function only allows positive values to pass through and sets all negative values to zero. Since most of the values in the output of a CONV layer are positive, many of the values will not be affected by the ReLU function, and therefore the output of the ReLU layer will look similar to the output of the CONV layer.

Q 4.2

The feature maps created by the second layer (CONV) differ from the original picture because they are the consequence of applying a series of filters to the original image. These filters are intended to recognise certain aspects in a picture, such as edges, shapes, textures, and so on. The feature maps depict the positions and strengths of certain characteristics in the picture. They are also often smaller in size than the original image since they have been convolved with the filters.

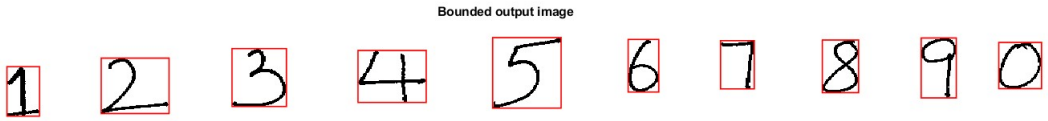
The feature maps created by the third layer (ReLU) differ from the original picture as well because they are the result of an activation function (ReLU) being applied to the feature maps generated by the second layer. The ReLU activation function is used to induce nonlinearity into the network and to reduce the number of negative values in feature maps. The ReLU layer's output will have the same structure as the input, but the values will be bounded to 0 if they are negative and maintained unchanged if, they are positive.

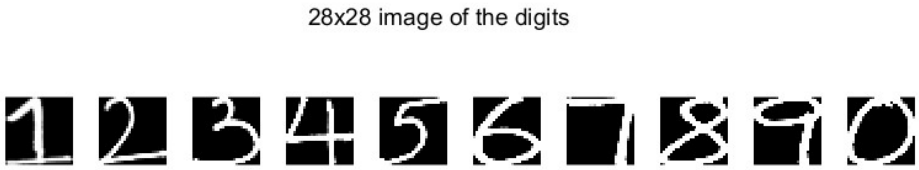
Part 5 Image Classification

The script ec.m script is a digit recognition program that uses the pre-trained model to predict digits in an image. The script starts by loading the lenet model, setting the batch size to 1, and specifying the directory of the images to be processed. For each image in the directory, the script reads and pre-processes the image by converting it to grayscale, binarizing it, removing small noise and detecting bounding boxes around connected components. The script then crops and resizes these regions, stores them as pre-processed digits, and passes them through the network to get predictions. The script then displays the original image with bounding boxes, the pre-processed digits, and the final predictions. The script also provides the name of the image and the

predicted digits as output. The script runs through all the 4 images from the images folder, but we will be looking at the first two image classification below.

IMAGE 1

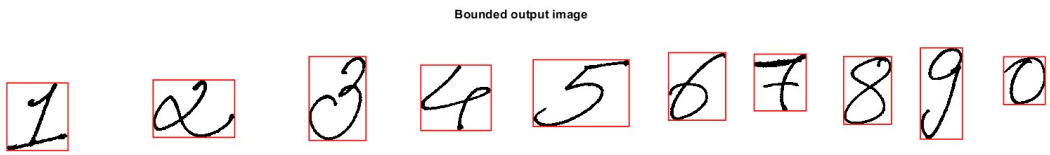





```
image1.JPG
Predicted digits :-      8     5     6     3     5     9     6     2     3     3
```

When the script finishes running, we see the above classification given for image 1. The script puts red coloured rectangular boundaries on each digit it recognizes then resizes them in 28 by 28 gray-scaled cropped versions and outputting them as a figure. As was described in 3.3 our recognition of the digits is not correct for all cases. We can see that our predictions are off for image 1.

IMAGE 2 -



```
image2.JPG
Predicted digits :-      5     4     3     8     5     5     3     3     3     6
```

The same thing is happening on image 2 as was discussed previously but we have a small, improved result for our prediction. We are getting two correct classifications i.e., 3 and 5 is correctly being predicted by our model on image 2.

Possible reasons for errors in the recognition:

Overfitting is one possibility which could occur as the model that is trained on has a limited set of data and is not able to generalize to new examples. Variations in font styles could be another reason which could lead to poor performance if the model was trained on a specific font style and the image contains digits in a different font style. Another reason could be that the images had small digits where the model might have difficulty recognizing small or low-resolution digits.