Project 5: Recognition using Deep Networks

Overview

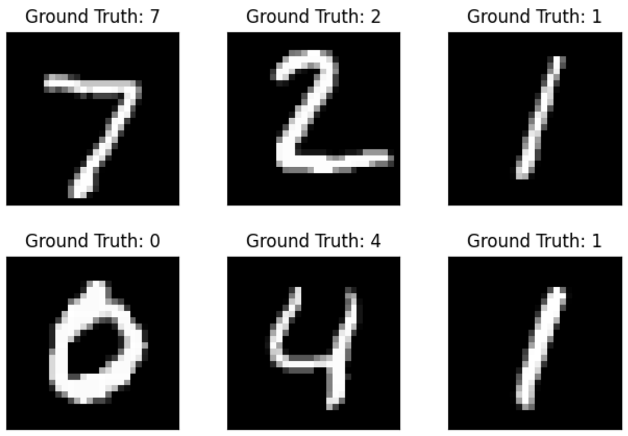
The aim of this project was to build, train, analyse and modify a deep network for digit recognition using MNIST dataset. Additionally, we also examined the network by analysing the different layers and seeing their effect on the images. We then used this trained model to identify the greek symbols alpha, beta, gamma by using the digit network as a digital embedding space. The last step of this project was to experiment with various different dimensions that would affect the performance of the model and try to optimise it.

Tasks

Build and train a network to recognise digits

A. Get the MNIST digit data set:

Plot of the first six example digits using matplotlib pyplot package:



B. Make your network code repeatable

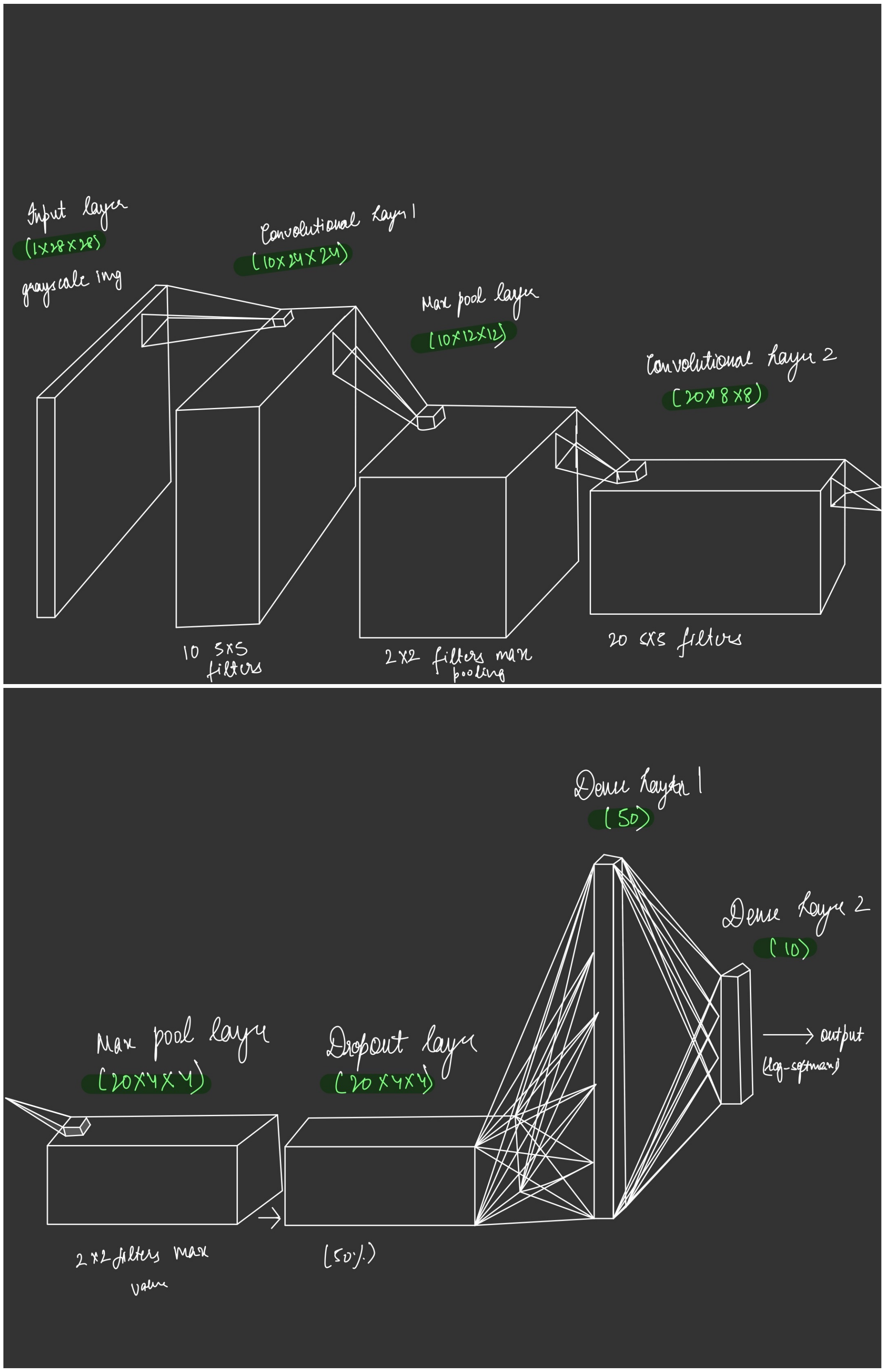
Made the code repeatable using torch.manual_seed(random_seed) with random_seed value 1. Additionally, turned CUDA off using torch.backends.cudnn.
enabled = False.

C. Build a network model

The network model is built with the following layers:

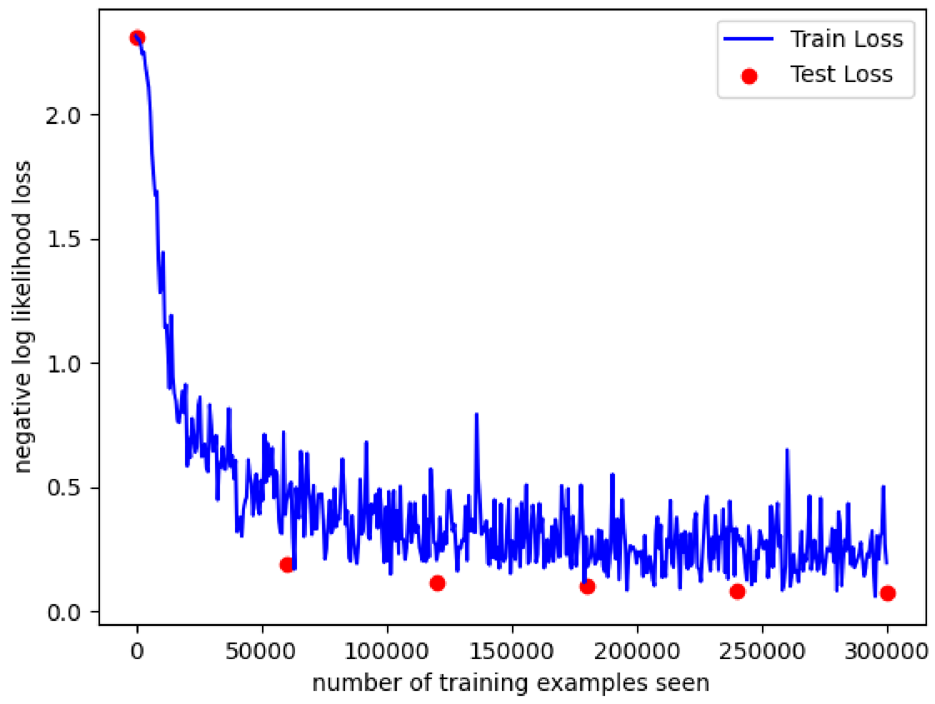
- A convolution layer with 10 5x5 filters
- A max pooling layer with a 2x2 window and a ReLU function applied.
- A convolution layer with 20 5x5 filters
- A dropout layer with a 0.5 dropout rate (50%)
- A max pooling layer with a 2x2 window and a ReLU function applied
- A flattening operation followed by a fully connected Linear layer with 50 nodes and a ReLU function on the output
- A final fully connected Linear layer with 10 nodes and the log_softmax function applied to the output.

The following is the diagram of the built network:



D. Train the model

The model is trained for 5 epochs and evaluated on both the training and test sets after each epoch. The following shows the plot for the training and testing errors:

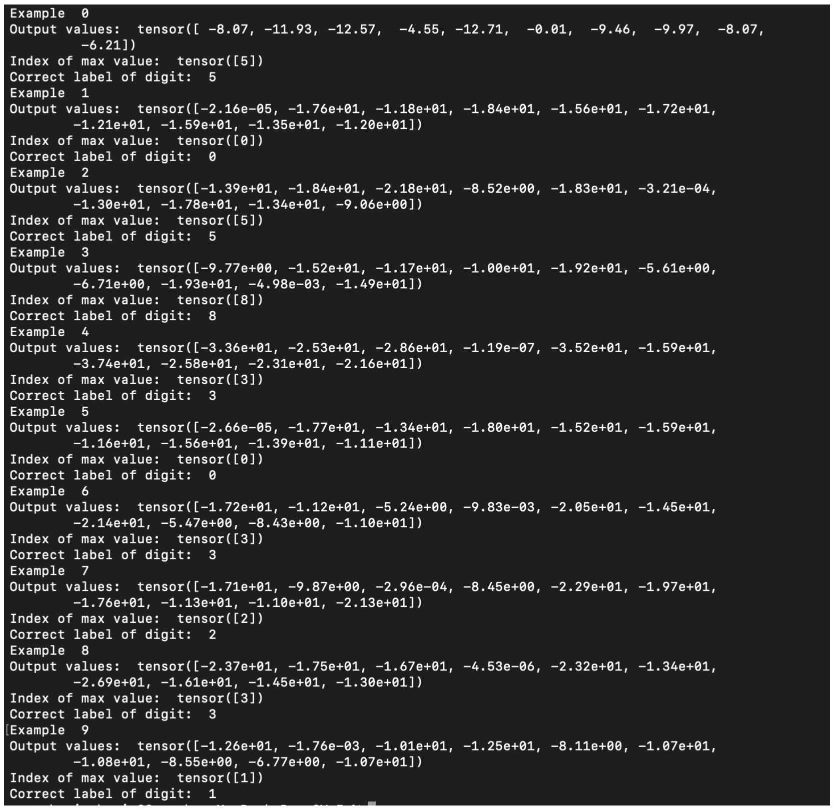


E. Save the network to a file

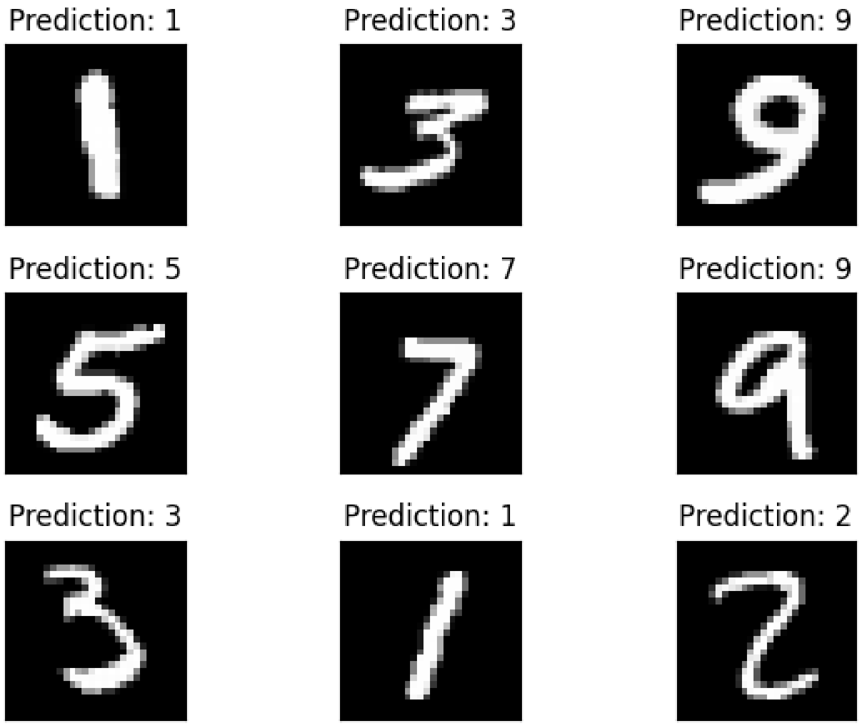
The network was saved to a file using: torch.save(network.state_dict(), 'model.pth')

F. Read the network and run it on the test set

The model was run on the first 10 examples of the test set. The first image below shows the 10 output values, the max output value index and the correct label of the digit for each of the first 10 examples. The second image shows the first 9 digits and their model predictions plotted using matplotlib. As we can see the network correctly classifies all the 10 examples.

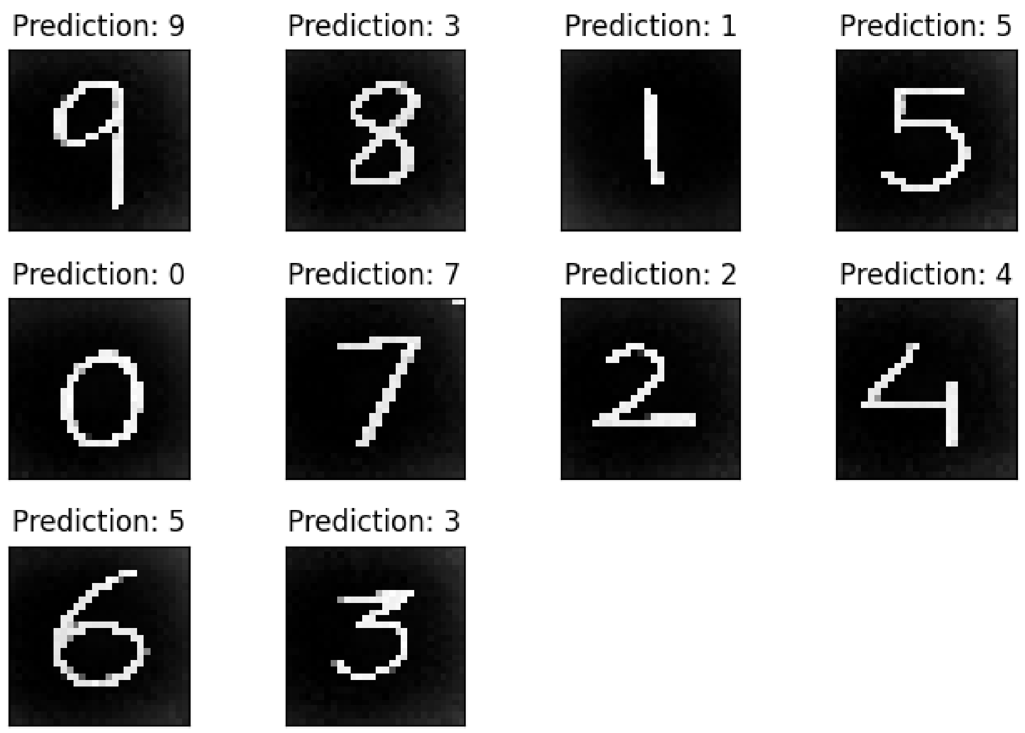


```
Example  0
Output values:  tensor([ -8.07, -11.93, -12.57,  -4.55, -12.71,  -0.01,  -9.46,  -9.97,  -8.07,
         -6.21])
Index of max value:  tensor([5])
Correct label of digit:  5
Example  1
Output values:  tensor([-2.16e-05, -1.76e+01, -1.18e+01, -1.84e+01, -1.56e+01, -1.72e+01,
        -1.21e+01, -1.59e+01, -1.35e+01, -1.20e+01])
Index of max value:  tensor([0])
Correct label of digit:  0
Example  2
Output values:  tensor([-1.39e+01, -1.84e+01, -2.18e+01, -8.52e+00, -1.83e+01, -3.21e-04,
        -1.30e+01, -1.78e+01, -1.34e+01, -9.06e+00])
Index of max value:  tensor([5])
Correct label of digit:  5
Example  3
Output values:  tensor([-9.77e+00, -1.52e+01, -1.17e+01, -1.00e+01, -1.92e+01, -5.61e+00,
        -6.71e+00, -1.93e+01, -4.98e-03, -1.49e+01])
Index of max value:  tensor([8])
Correct label of digit:  8
Example  4
Output values:  tensor([-3.36e+01, -2.53e+01, -2.86e+01, -1.19e-07, -3.52e+01, -1.59e+01,
        -3.74e+01, -2.58e+01, -2.31e+01, -2.16e+01])
Index of max value:  tensor([3])
Correct label of digit:  3
Example  5
Output values:  tensor([-2.66e-05, -1.77e+01, -1.34e+01, -1.80e+01, -1.52e+01, -1.59e+01,
        -1.16e+01, -1.56e+01, -1.39e+01, -1.11e+01])
Index of max value:  tensor([0])
Correct label of digit:  0
Example  6
Output values:  tensor([-1.72e+01, -1.12e+01, -5.24e+00, -9.83e-03, -2.05e+01, -1.45e+01,
        -2.14e+01, -5.47e+00, -8.43e+00, -1.10e+01])
Index of max value:  tensor([3])
Correct label of digit:  3
Example  7
Output values:  tensor([-1.71e+01, -9.87e+00, -2.96e-04, -8.45e+00, -2.29e+01, -1.97e+01,
        -1.76e+01, -1.13e+01, -1.10e+01, -2.13e+01])
Index of max value:  tensor([2])
Correct label of digit:  2
Example  8
Output values:  tensor([-2.37e+01, -1.75e+01, -1.67e+01, -4.53e-06, -2.32e+01, -1.34e+01,
        -2.69e+01, -1.61e+01, -1.45e+01, -1.30e+01])
Index of max value:  tensor([3])
Correct label of digit:  3
Example  9
Output values:  tensor([-1.26e+01, -1.76e-03, -1.01e+01, -1.25e+01, -8.11e+00, -1.07e+01,
        -1.08e+01, -8.55e+00, -6.77e+00, -1.07e+01])
Index of max value:  tensor([1])
Correct label of digit:  1
```



G. Test the network on new inputs

The model was tested out on 10 handwritten digits. The images which were written in black ink on white background, were read, resized to 28x28, normalised, converted to grayscale, and the colours were inverted as the images contained in the dataset were numbers written in white ink on a black background. The following image shows the prediction of the network for the 10 digits:

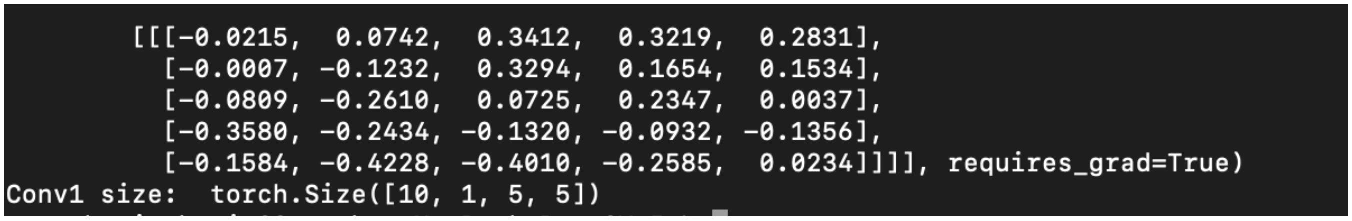


We can see that the network worked well for 8/10 digits. There could be various reasons for the same ranging from the input images varying in size, intensities, thickness of lines etc from the training images. This problem can be rectified by including more images in the dataset.

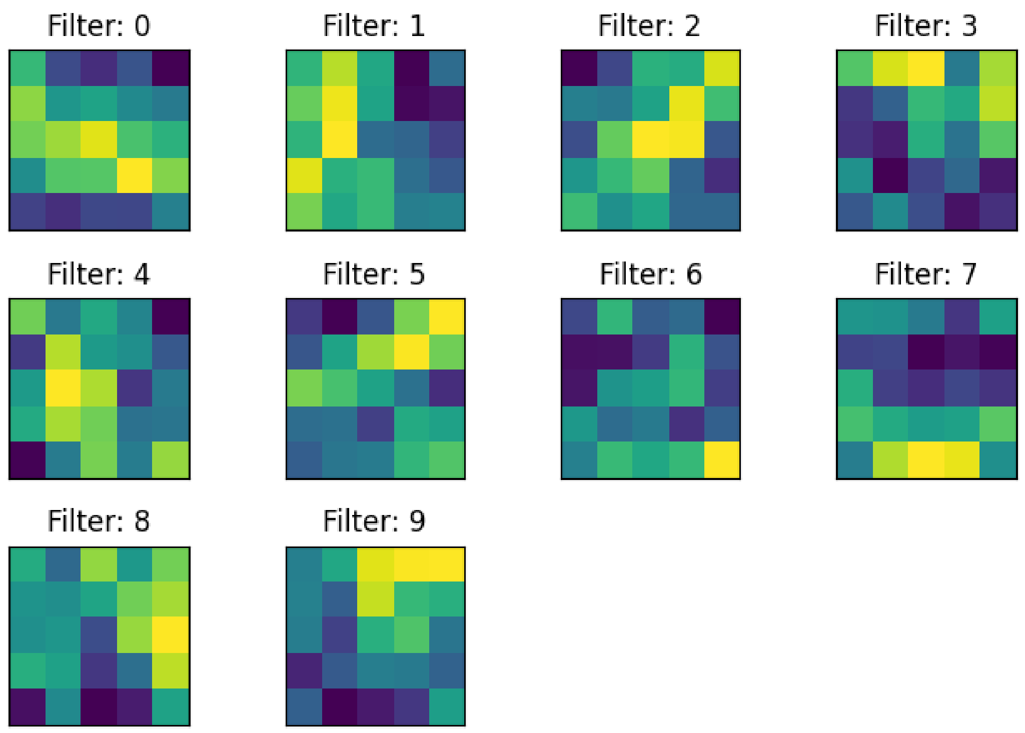
Examine your network

A. Analyse the first layer

Got the weights of the first layer using model.conv1.weight. The result is a tensor with the shape [10,1,5,5] as can be seen from the first screenshot. The second image shows the 10 filters using pyplot.

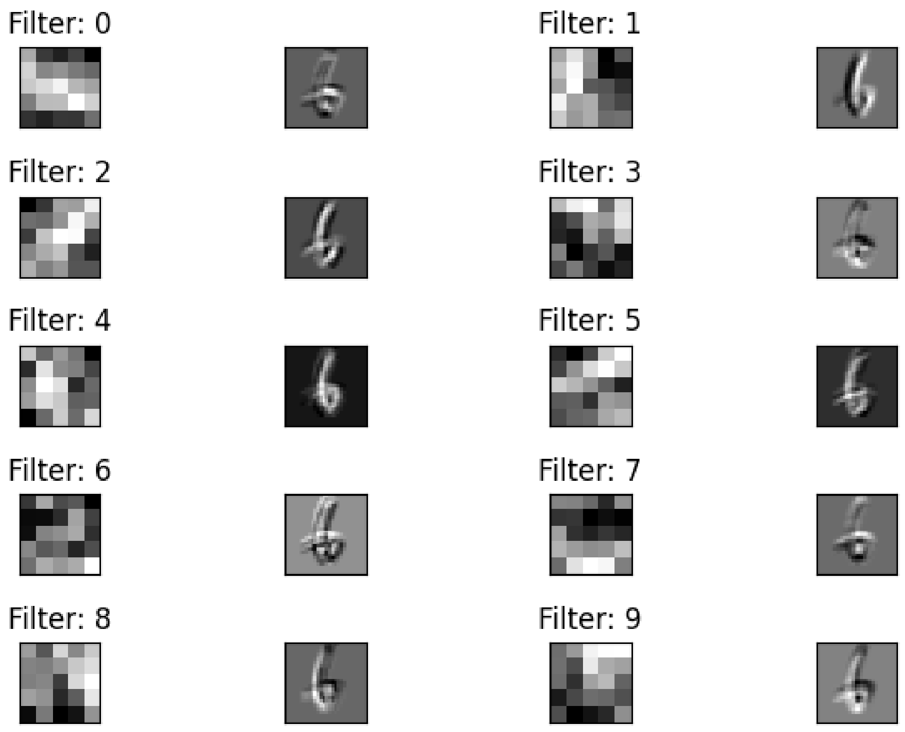


```
        [[[-0.0215,  0.0742,  0.3412,  0.3219,  0.2831],
          [-0.0007, -0.1232,  0.3294,  0.1654,  0.1534],
          [-0.0809, -0.2610,  0.0725,  0.2347,  0.0037],
          [-0.3580, -0.2434, -0.1320, -0.0932, -0.1356],
          [-0.1584, -0.4228, -0.4010, -0.2585,  0.0234]]]], requires_grad=True)
Conv1 size:  torch.Size([10, 1, 5, 5])
```



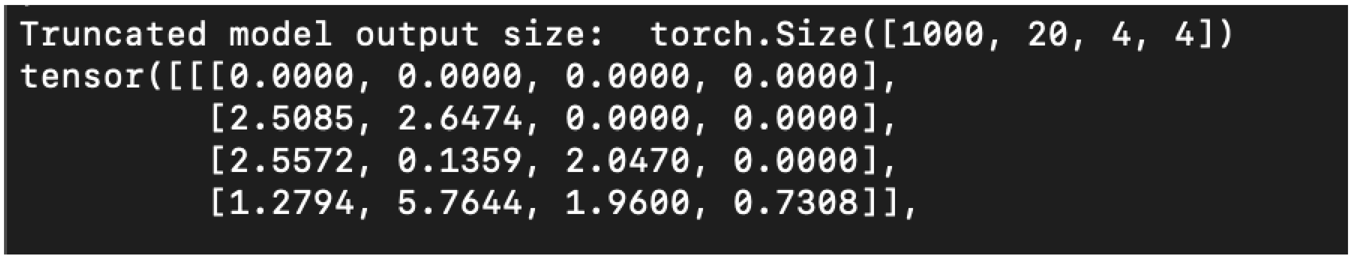
B. Show the effect of the filters

The following plot shows the the 10 filters applied to the first training example image using OpenCV filter2D function. We can see from the results that in all the different filters, a gradient along a different direction is highlighted, hence helping us recognise edges along various different directions, and eventually shapes which help us recognise digits.

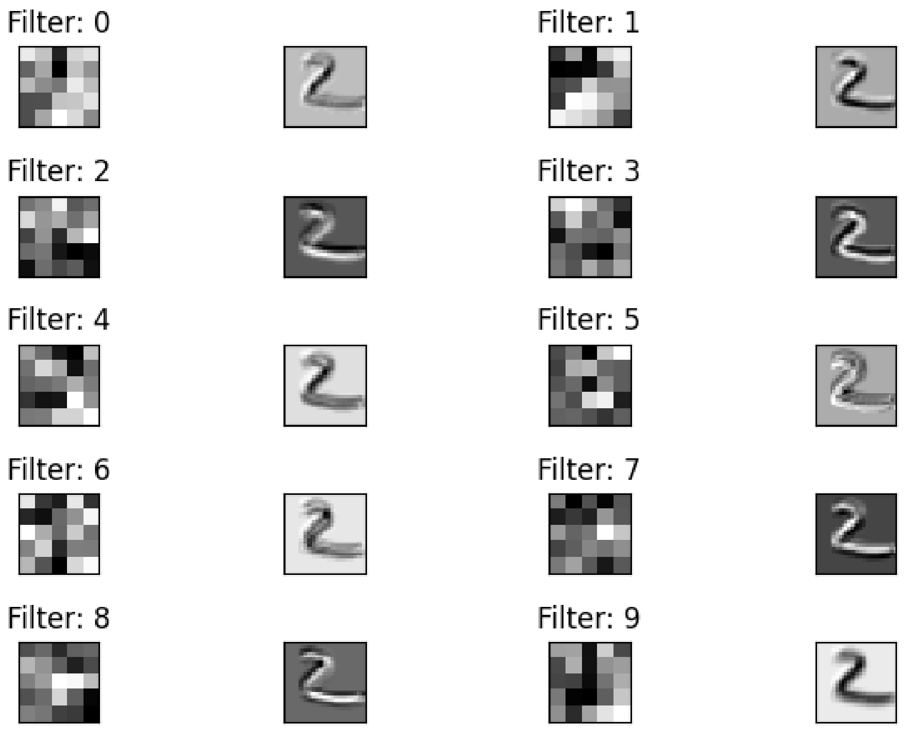


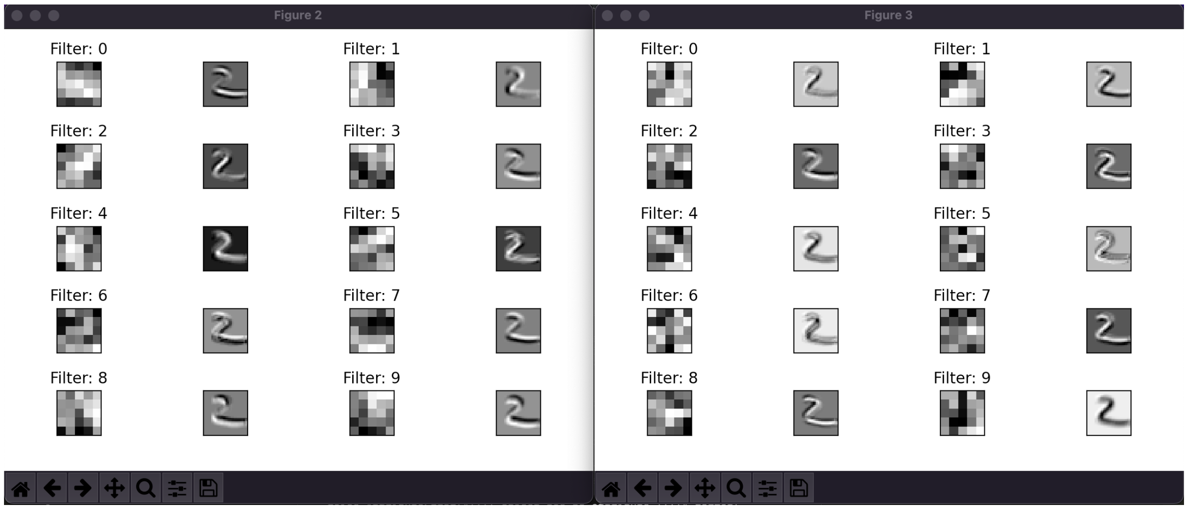
C. Build a truncated model

Built a truncated new model which only contains the first two convolutional layers, by creating a subclass of the Neural Network class that we created in the previous tasks. We can see that the size of the output channels after applying this truncated model to our dataset is [1000,20,4,4]. The second image shows the effects after the second convolution layer on the first example image. The third image shows the effects of the first convolution layer compared to the effects of the first and the second convolution layers.



```
Truncated model output size:  torch.Size([1000, 20, 4, 4])
tensor([[[0.0000, 0.0000, 0.0000, 0.0000],
         [2.5085, 2.6474, 0.0000, 0.0000],
         [2.5572, 0.1359, 2.0470, 0.0000],
         [1.2794, 5.7644, 1.9600, 0.7308]],
```

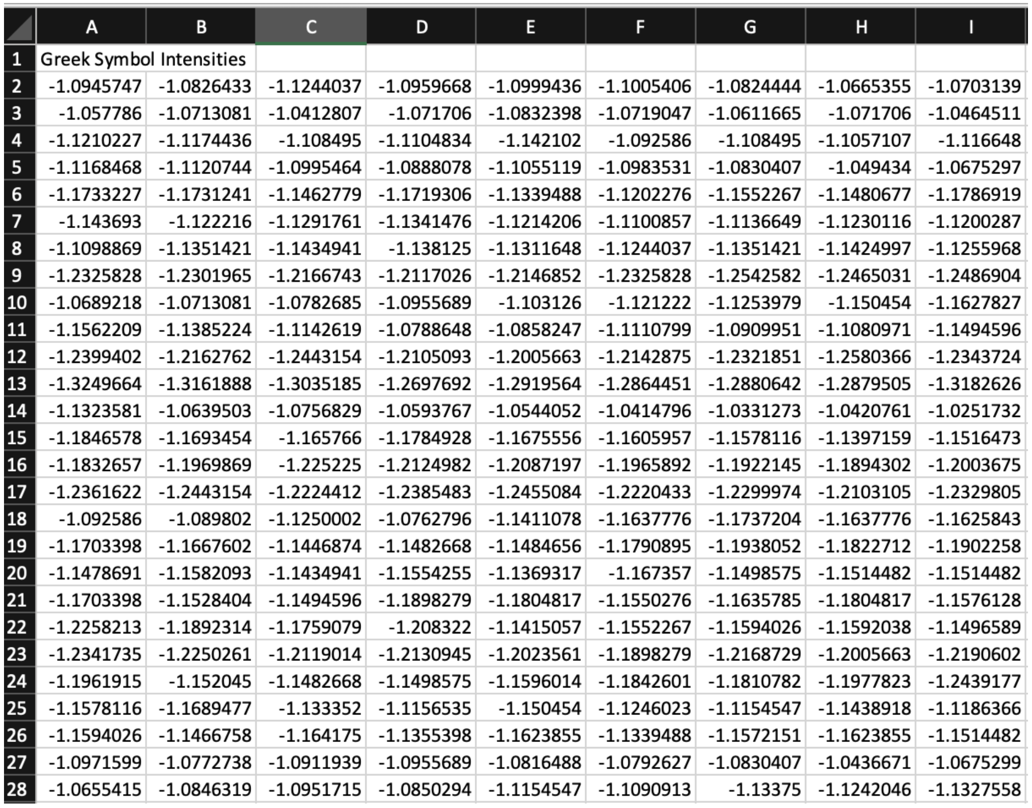




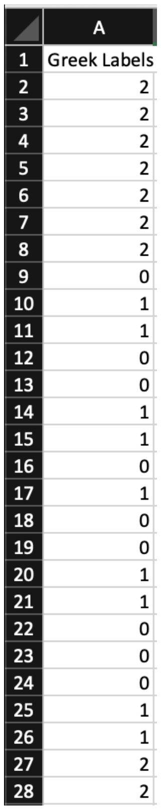
Create a digit embedding space

A. Create a greek symbol data set

Wrote a program to read in the 27 greek symbol images from the given dataset, scaled them down to 28x28, converted them to greyscale, inverted the intensities, and wrote out the data set as two CSV files: one file containing a header row and 27 data rows with 784 intensity values in each row and the other file containing a header row and 27 data rows with just the category (alpha = 0, beta = 1, gamma = 2). The following images show a screenshot of both the aforementioned csv files respectively:



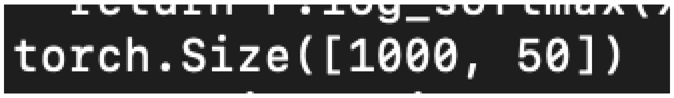
	A	B	C	D	E	F	G	H	I
1	Greek Symbol Intensities								
2	-1.0945747	-1.0826433	-1.1244037	-1.0959668	-1.0999436	-1.1005406	-1.0824444	-1.0665355	-1.0703139
3	-1.057786	-1.0713081	-1.0412807	-1.071706	-1.0832398	-1.0719047	-1.0611665	-1.071706	-1.0464511
4	-1.1210227	-1.1174436	-1.108495	-1.1104834	-1.142102	-1.092586	-1.108495	-1.1057107	-1.116648
5	-1.1168468	-1.1120744	-1.0995464	-1.0888078	-1.1055119	-1.0983531	-1.0830407	-1.049434	-1.0675297
6	-1.1733227	-1.1731241	-1.1462779	-1.1719306	-1.1339488	-1.1202276	-1.1552267	-1.1480677	-1.1786919
7	-1.143693	-1.122216	-1.1291761	-1.1341476	-1.1214206	-1.1100857	-1.1136649	-1.1230116	-1.1200287
8	-1.1098869	-1.1351421	-1.1434941	-1.138125	-1.1311648	-1.1244037	-1.1351421	-1.1424997	-1.1255968
9	-1.2325828	-1.2301965	-1.2166743	-1.2117026	-1.2146852	-1.2325828	-1.2542582	-1.2465031	-1.2486904
10	-1.0689218	-1.0713081	-1.0782685	-1.0955689	-1.103126	-1.121222	-1.1253979	-1.150454	-1.1627827
11	-1.1562209	-1.1385224	-1.1142619	-1.0788648	-1.0858247	-1.1110799	-1.0909951	-1.1080971	-1.1494596
12	-1.2399402	-1.2162762	-1.2443154	-1.2105093	-1.2005663	-1.2142875	-1.2321851	-1.2580366	-1.2343724
13	-1.3249664	-1.3161888	-1.3035185	-1.2697692	-1.2919564	-1.2864451	-1.2880642	-1.2879505	-1.3182626
14	-1.1323581	-1.0639503	-1.0756829	-1.0593767	-1.0544052	-1.0414796	-1.0331273	-1.0420761	-1.0251732
15	-1.1846578	-1.1693454	-1.165766	-1.1784928	-1.1675556	-1.1605957	-1.1578116	-1.1397159	-1.1516473
16	-1.1832657	-1.1969869	-1.225225	-1.2124982	-1.2087197	-1.1965892	-1.1922145	-1.1894302	-1.2003675
17	-1.2361622	-1.2443154	-1.2224412	-1.2385483	-1.2455084	-1.2220433	-1.2299974	-1.2103105	-1.2329805
18	-1.092586	-1.089802	-1.1250002	-1.0762796	-1.1411078	-1.1637776	-1.1737204	-1.1637776	-1.1625843
19	-1.1703398	-1.1667602	-1.1446874	-1.1482668	-1.1484656	-1.1790895	-1.1938052	-1.1822712	-1.1902258
20	-1.1478691	-1.1582093	-1.1434941	-1.1554255	-1.1369317	-1.167357	-1.1498575	-1.1514482	-1.1514482
21	-1.1703398	-1.1528404	-1.1494596	-1.1898279	-1.1804817	-1.1550276	-1.1635785	-1.1804817	-1.1576128
22	-1.2258213	-1.1892314	-1.1759079	-1.208322	-1.1415057	-1.1552267	-1.1594026	-1.1592038	-1.1496589
23	-1.2341735	-1.2250261	-1.2119014	-1.2130945	-1.2023561	-1.1898279	-1.2168729	-1.2005663	-1.2190602
24	-1.1961915	-1.152045	-1.1482668	-1.1498575	-1.1596014	-1.1842601	-1.1810782	-1.1977823	-1.2439177
25	-1.1578116	-1.1689477	-1.133352	-1.1156535	-1.150454	-1.1246023	-1.1154547	-1.1438918	-1.1186366
26	-1.1594026	-1.1466758	-1.164175	-1.1355398	-1.1623855	-1.1339488	-1.1572151	-1.1623855	-1.1514482
27	-1.0971599	-1.0772738	-1.0911939	-1.0955689	-1.0816488	-1.0792627	-1.0830407	-1.0436671	-1.0675299
28	-1.0655415	-1.0846319	-1.0951715	-1.0850294	-1.1154547	-1.1090913	-1.13375	-1.1242046	-1.1327558



	A
1	Greek Labels
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	0
10	1
11	1
12	0
13	0
14	1
15	1
16	0
17	1
18	0
19	0
20	1
21	1
22	0
23	0
24	0
25	1
26	1
27	2
28	2

B. Create a truncated model

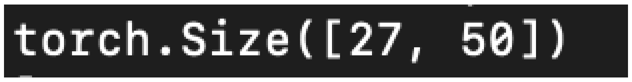
We create a truncated model from the old network that terminates at the Dense layer with 50 outputs. On applying this truncated model to our example data set. we can see that it gives 50 numbers as its output:



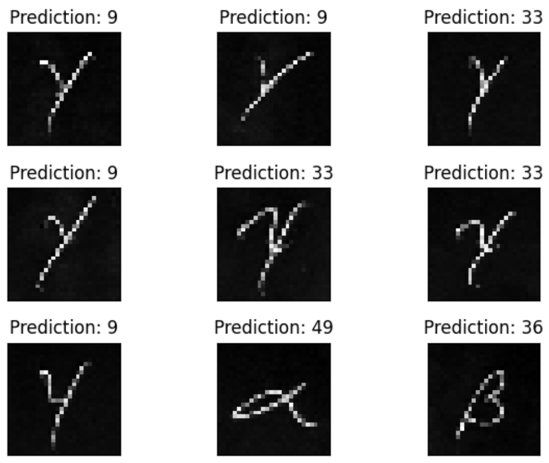
```
torch.Size([1000, 50])
```

C.Project the greek symbols into the embedding space

We then apply this truncated model to out greek symbol data set and get a set of 27 50 element vectors. The following image shows some of the predictions of the model applied to the greek symbol dataset.

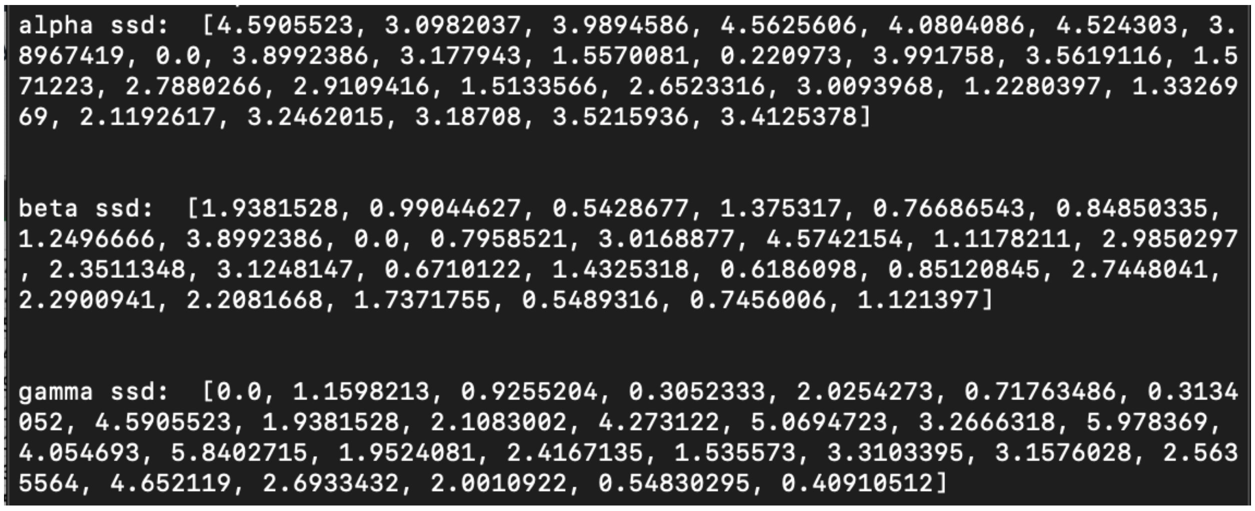


```
torch.Size([27, 50])
```



D. Compute distances in the embedding space

The following image shows the results of computing the sum squared distance between 3 examples of alpha, beta, gamma and all 27 examples:

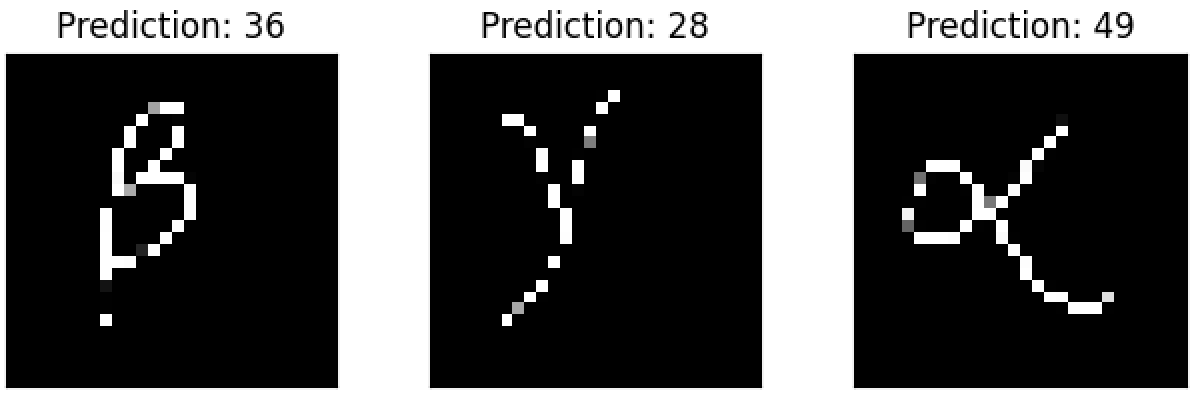


```
alpha ssd:  [4.5905523, 3.0982037, 3.9894586, 4.5625606, 4.0804086, 4.524303, 3.
8967419, 0.0, 3.8992386, 3.177943, 1.5570081, 0.220973, 3.991758, 3.5619116, 1.5
71223, 2.7880266, 2.9109416, 1.5133566, 2.6523316, 3.0093968, 1.2280397, 1.33269
69, 2.1192617, 3.2462015, 3.18708, 3.5215936, 3.4125378]


beta ssd:  [1.9381528, 0.99044627, 0.5428677, 1.375317, 0.76686543, 0.84850335,
1.2496666, 3.8992386, 0.0, 0.7958521, 3.0168877, 4.5742154, 1.1178211, 2.9850297
, 2.3511348, 3.1248147, 0.6710122, 1.4325318, 0.6186098, 0.85120845, 2.7448041,
2.2900941, 2.2081668, 1.7371755, 0.5489316, 0.7456006, 1.121397]


gamma ssd:  [0.0, 1.1598213, 0.9255204, 0.3052333, 2.0254273, 0.71763486, 0.3134
052, 4.5905523, 1.9381528, 2.1083002, 4.273122, 5.0694723, 3.2666318, 5.978369,
4.054693, 5.8402715, 1.9524081, 2.4167135, 1.535573, 3.3103395, 3.1576028, 2.563
5564, 4.652119, 2.6933432, 2.0010922, 0.54830295, 0.40910512]
```

We can see that the distance between the same greek symbols is less as compared to the distance with other symbols. A nearest neighbour or KNN classifier would work well in the embedding space for this task as again, the sum of k similar symbols would be the least and we'd be able to accurately classify the symbols.

E. Create your own greek symbol data

The model was applied to handwritten images of alpha, beta and gamma which were transformed to resemble the training dataset. The following image shows the results of the same:



We can see that alpha and beta are correctly recognised (they have the same values as when the model was applied to the training data set of greek symbols). From the training data set we can see that gamma has various different values and a correct prediction could not be made for it. We would need to add many more examples to our training data set to get the correct results.

Design your own experiment

To gain a deeper understanding of how deep networks work and the impact of varying various dimensions on the performance of the network, we designed an experiment that would vary 3 dimensions one by one. The dimensions could be any of the following:

- The number of convolution layers
- The size of the convolution filters
- The number of convolution filters in a layer
- The number of hidden nodes in the Dense layer
- The dropout rates of the Dropout layer
- Whether to add another dropout layer after the fully connected layer
- The size of the pooling layer filters
- The number or location of pooling layers
- The activation function for each layer
- The number of epochs of training
- The batch size while training

A. Develop a plan

The dimensions I chose to vary were the following:

- The **batch size** while training: [32,64,128]
- The **dropout rates** of the Dropout layer: [0.25,0.5,0.75]
- The number of **epochs** of training: [2-7]

The dimensions are changed in the same order as they are mentioned, by keeping 2 dimensions constant and looping through all values of the third and so on. This gives a total of 54 network variations.

B. Predict the results

I believe that as the number of epochs increase, the performance of the network should get better to a certain extent, as the batch size for training increases the performance will get worse, and as the dropout rate increases the performance will get worse as we might get rid of a lot of useful data that would have helped the network perform better.

C. Execute your plan

The following table shows the accuracy and losses observed for the different combinations of the three dimensions. The accuracy always increased by increasing the number of epochs, hence I have only shown the accuracy of the model after the maximum epoch value.

Batch Size	Dropout rates	Epochs	Loss (Test set)	Accuracy(%)
32	0.25	7	0.0389	99
32	0.50	7	0.040	99
32	0.75	7	0.0594	99
64	0.25	7	0.0487	99
64	0.50	7	0.0532	99
64	0.75	7	0.0864	98

128	0.25	7	0.0658	98
128	0.50	7	0.0772	98
128	0.75	7	0.01181	98

We can see that, as predicted, the models performance got worse as we increased the batch size and the dropout rate.

Reflections

This project gave me an insight into how deep networks work and helped put the lectures into perspective. I got a better understanding of convolutional layers, the various dimensions and how varying them impacts the performance of the network. Additionally, it also familiarised me with Pytorch and its various functions. The project seemed daunting at first, just because I am new to the world of neural networks but the lectures and various online resources helped me pursue the project and clarify many doubts I had about deep networks.

Acknowledgements

Heavily relied on Pytorch documentation and the tutorials linked in the project description. https://nextjournal.com/gkoehler/pytorch-mnist