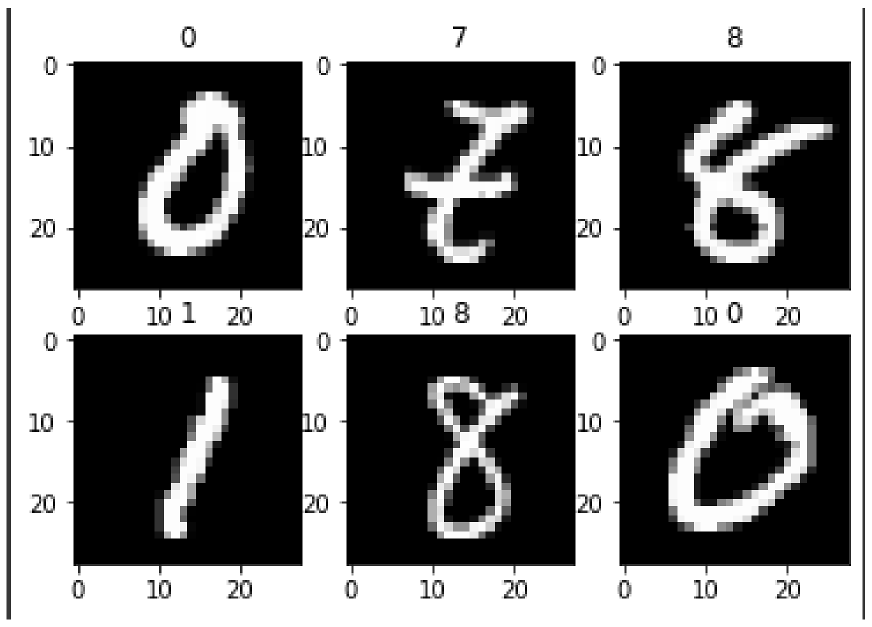
The project involved constructing a Convolutional Neural Network that classifies dataset according to layers containing randomized filter weights. The project focused on training a network with MNIST dataset and testing out the network on test data of the MNIST dataset and handwritten digits. We further analyzed how the network analyzes data by printing out the filters in each layer. The filters showed individually what the network looks for in an image to classify it. The project further explored training a similar but small dataset of Greek symbols, create an embedding space for the same and create truncated models of the network.

Task 1

Build and train a network to recognize digits

Part A: Get the MNIST dataset

Used Torch vision library to import the MNIST dataset and create a loader by specifying batch size for both training and testing dataset and printed out the first 6 digits using matplotlib:



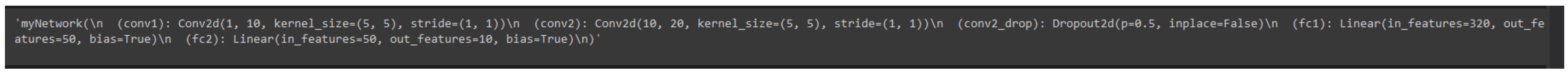
Part B: Make your network Code Repeatable:

Used torch.manual_seed(69)and disabled cuda in order to make the network code repeatable

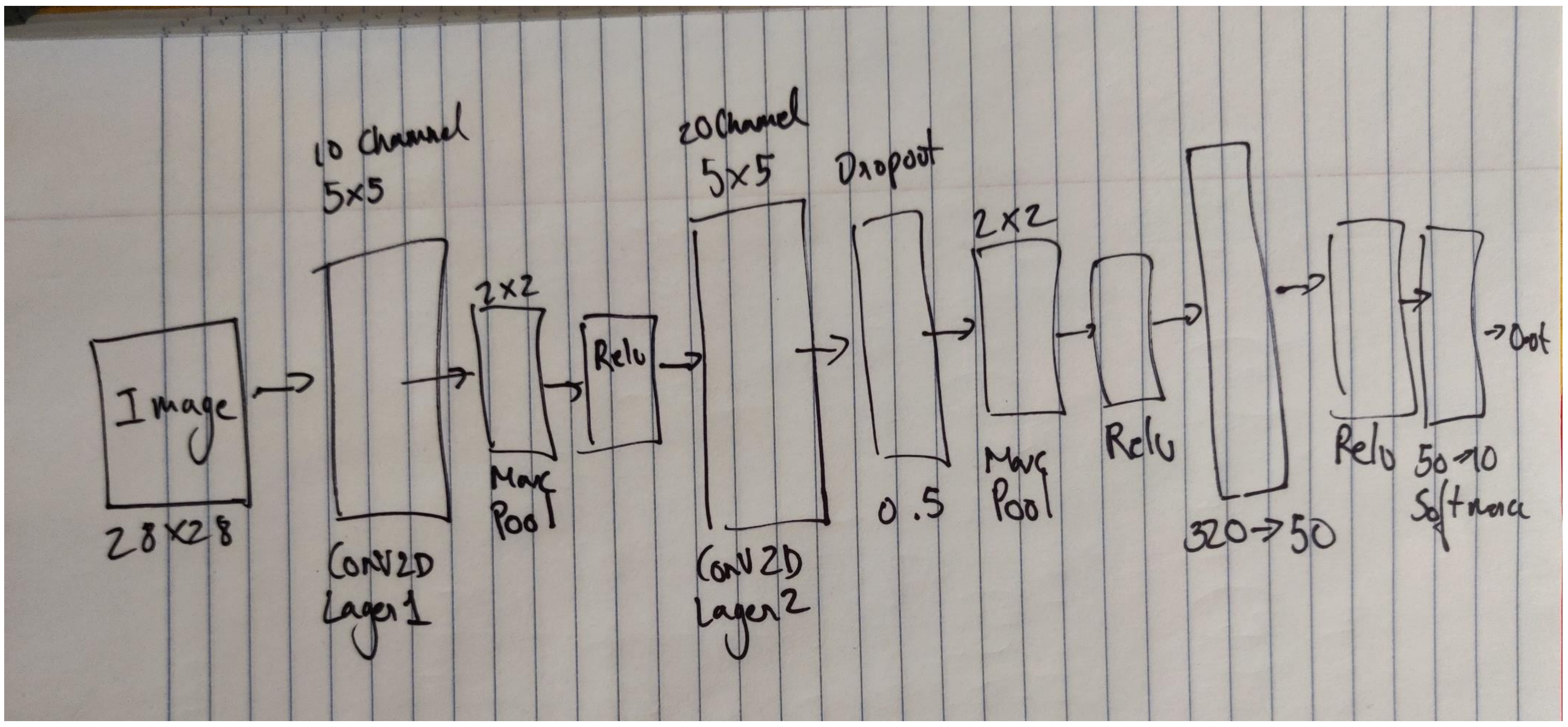
Part C: Build a Network Model:

Created a network model as stated in the instruction

Image -> Conv2D (10 channel 5*5 kernel) -> max pool(2*2) -> rectified linear activation function(Relu) -> Conv2D (20 filters, 10 input channels, 5*5 kernel) -> drop out function(0.5) ->Max pooling -> ReLu -> tensor.view(320) -> linear transformation(320 -> 50) -> linear transformation (50 -> 10)



```
'myNetwork(\n  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))\n  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))\n  (conv2_drop): Dropout2d(p=0.5, inplace=False)\n  (fc1): Linear(in_features=320, out_fe
atures=50, bias=True)\n  (fc2): Linear(in_features=50, out_features=10, bias=True)\n)'
```



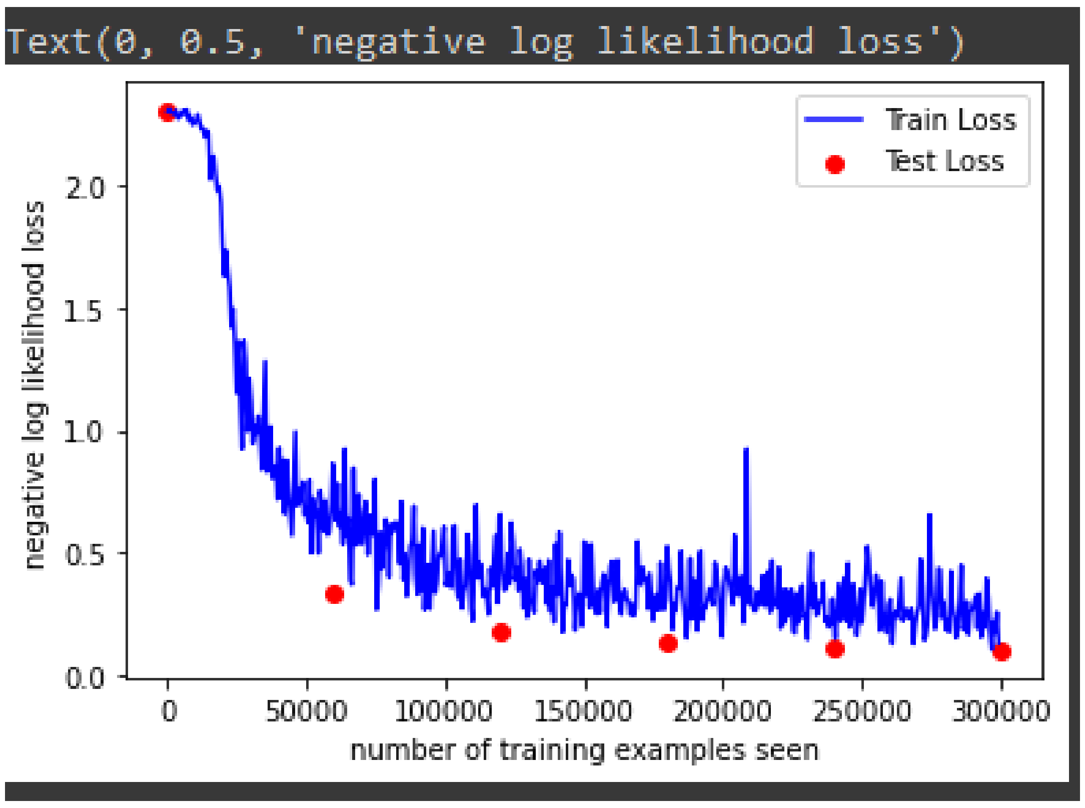
Part D: Train the Model:

Number of Epochs = 5

Number of Batch Size for training – 640

Number of Batch Size for testing – 2000

I tried implementing different epoch and batch size, but the model went through overfitting and learnt the data more than it needed to, this configuration worked out the best. I didn't want the model to be more than 97% accurate.



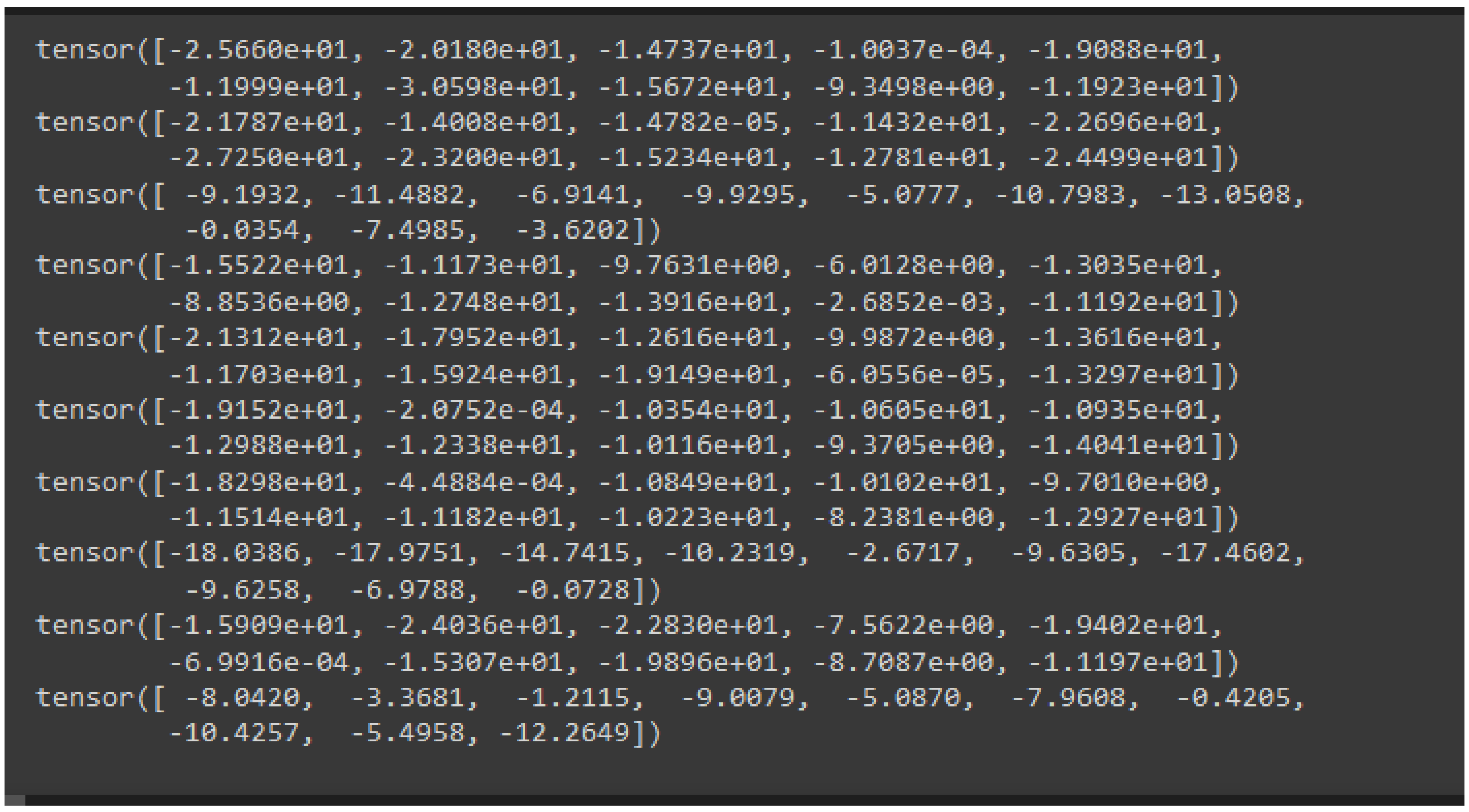
```
Text(0, 0.5, 'negative log likelihood loss')
```

Part E) Save the network to a file

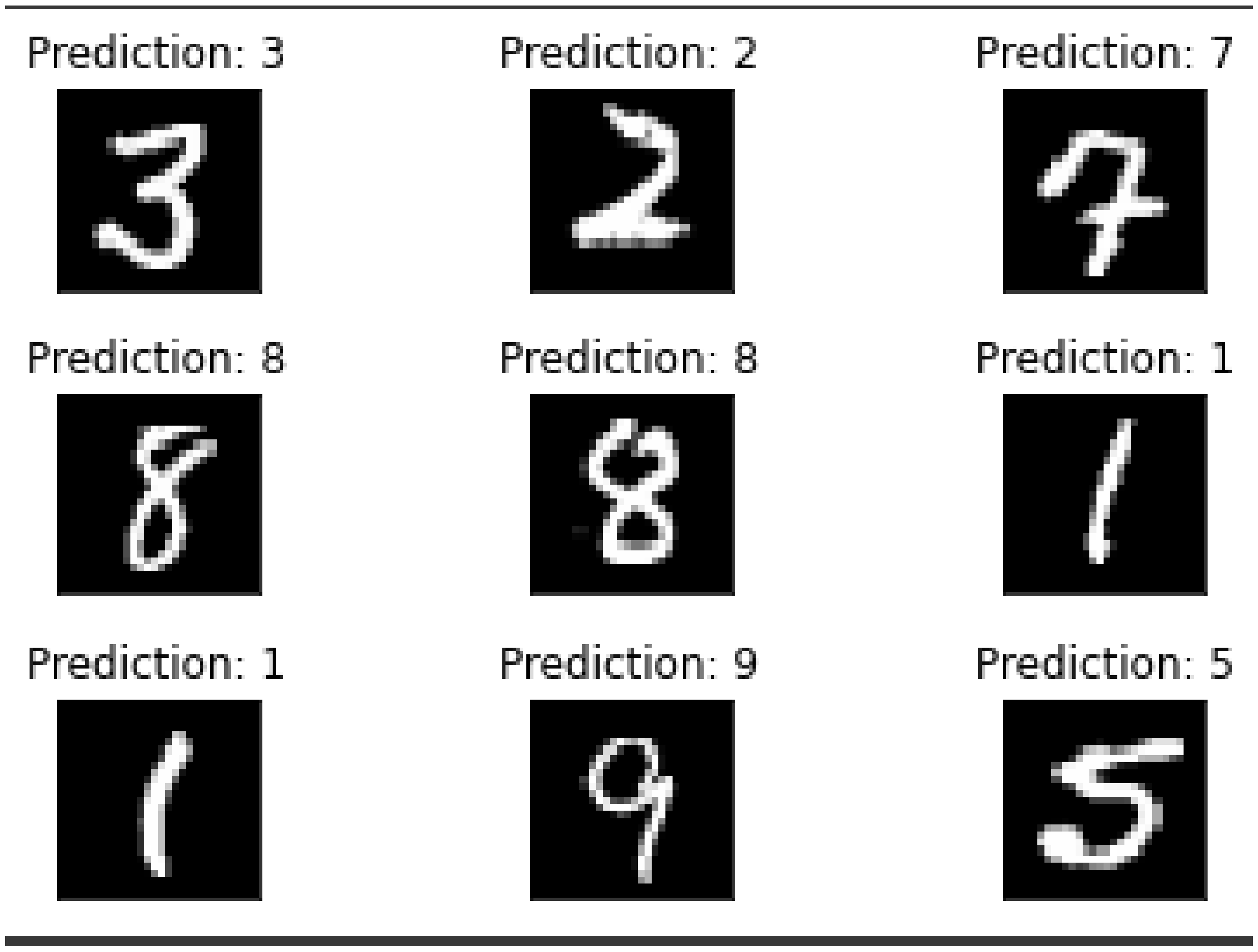
Saved the model and optimizer file using network.state_dict() in google drive and enabled the user to be able to download them directly

Part F) Read the network and run it on a test

In a separate file, I loaded the network using load_state_dict(torch.load) and ran it through the same MNIST dataset while setting the network to eval() mode

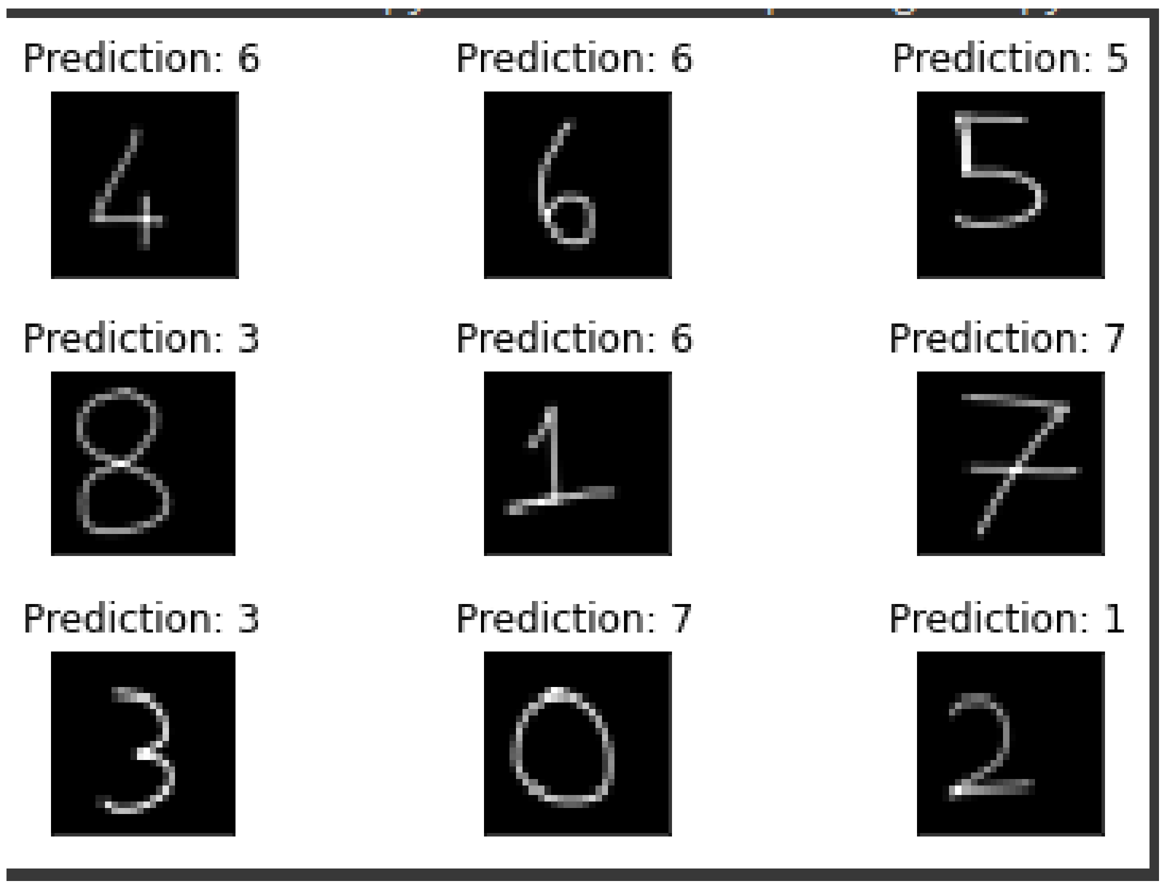


```
tensor([-2.5660e+01, -2.0180e+01, -1.4737e+01, -1.0037e-04, -1.9088e+01,
        -1.1999e+01, -3.0598e+01, -1.5672e+01, -9.3498e+00, -1.1923e+01])
tensor([-2.1787e+01, -1.4008e+01, -1.4782e-05, -1.1432e+01, -2.2696e+01,
        -2.7250e+01, -2.3200e+01, -1.5234e+01, -1.2781e+01, -2.4499e+01])
tensor([ -9.1932, -11.4882,  -6.9141,  -9.9295,  -5.0777, -10.7983, -13.0508,
         -0.0354,  -7.4985,  -3.6202])
tensor([-1.5522e+01, -1.1173e+01, -9.7631e+00, -6.0128e+00, -1.3035e+01,
        -8.8536e+00, -1.2748e+01, -1.3916e+01, -2.6852e-03, -1.1192e+01])
tensor([-2.1312e+01, -1.7952e+01, -1.2616e+01, -9.9872e+00, -1.3616e+01,
        -1.1703e+01, -1.5924e+01, -1.9149e+01, -6.0556e-05, -1.3297e+01])
tensor([-1.9152e+01, -2.0752e-04, -1.0354e+01, -1.0605e+01, -1.0935e+01,
        -1.2988e+01, -1.2338e+01, -1.0116e+01, -9.3705e+00, -1.4041e+01])
tensor([-1.8298e+01, -4.4884e-04, -1.0849e+01, -1.0102e+01, -9.7010e+00,
        -1.1514e+01, -1.1182e+01, -1.0223e+01, -8.2381e+00, -1.2927e+01])
tensor([-18.0386, -17.9751, -14.7415, -10.2319,  -2.6717,  -9.6305, -17.4602,
         -9.6258,  -6.9788,  -0.0728])
tensor([-1.5909e+01, -2.4036e+01, -2.2830e+01, -7.5622e+00, -1.9402e+01,
        -6.9916e-04, -1.5307e+01, -1.9896e+01, -8.7087e+00, -1.1197e+01])
tensor([ -8.0420,  -3.3681,  -1.2115,  -9.0079,  -5.0870,  -7.9608,  -0.4205,
        -10.4257,  -5.4958, -12.2649])
```



Task G: Test the network on new inputs

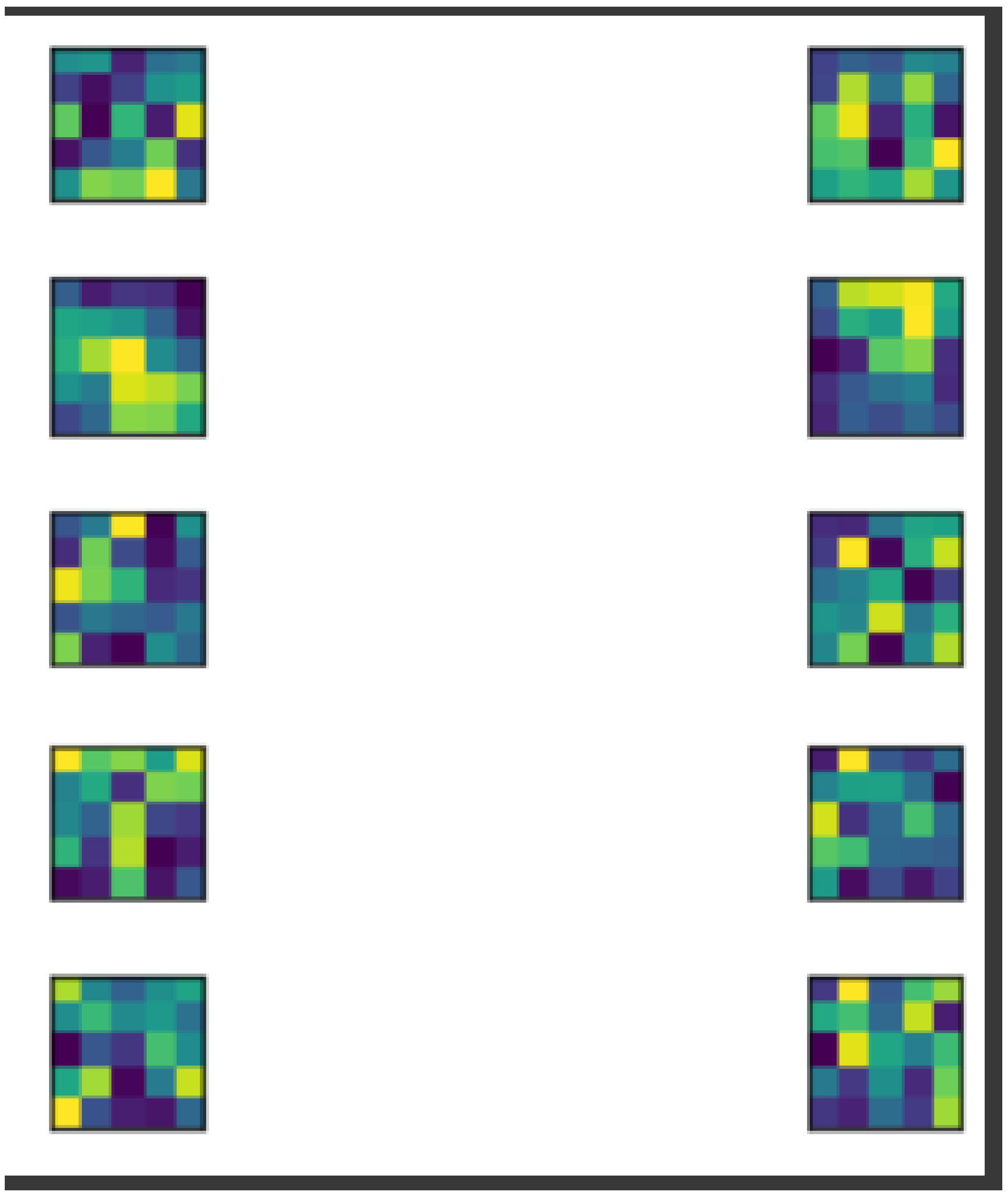
For this task I uploaded the folder of handwritten digits on Google Drive and used torchvision.datasets.ImageFolder to make a testing dataset, I tried to make the new images as sharp as possible for the system to have difficulty predicting them. The trained network managed to recognize 4/10 images.



Task 2: Examine your network:

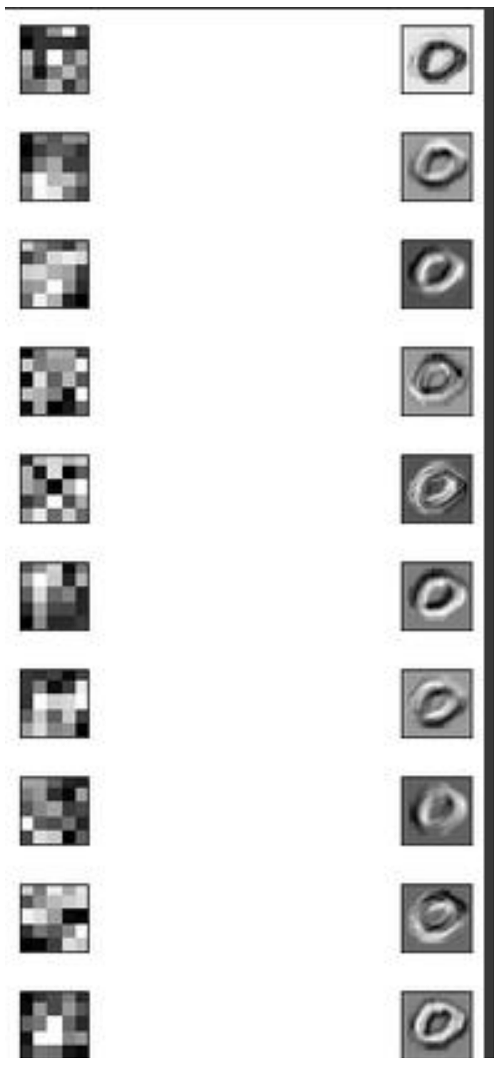
A) Analyze the first layer

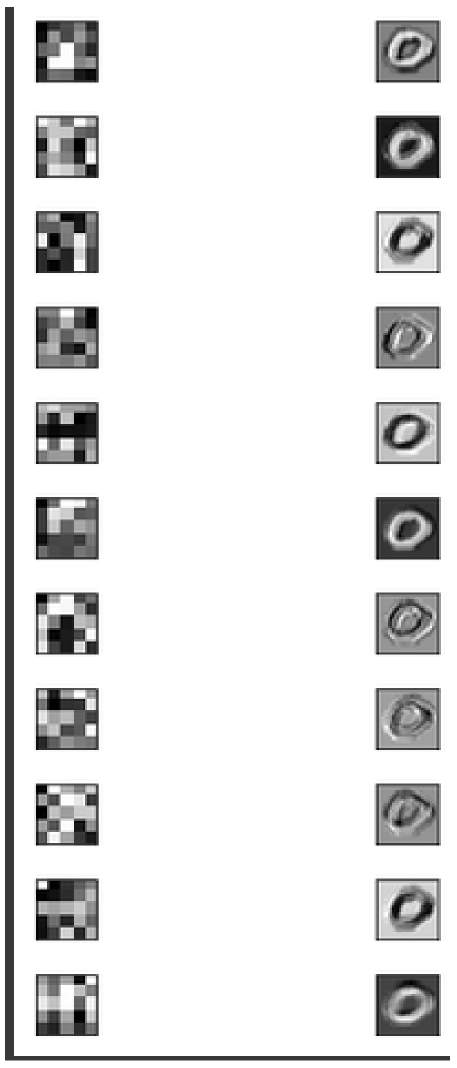
Got the weights of the first layer and printed out the filters(5*5). The filters seem to be detecting edges and shapes. Most of them are Sobel filters to print out the intensities. The filters are random, so they are also trying to predict the orientation and rotation of the digits.



B) Show the effect of filters:

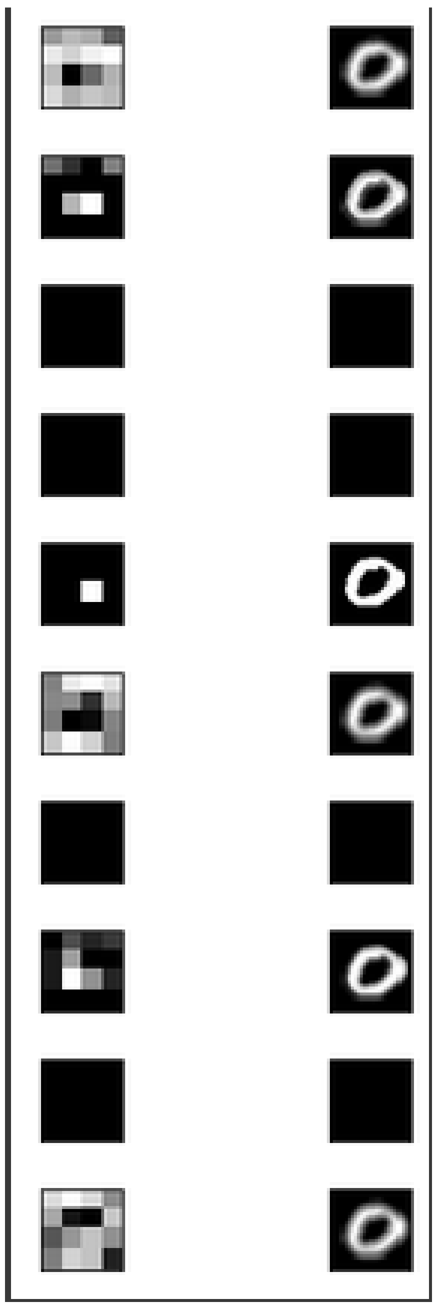
With using torch.no_grad(), I applied 20 2d filters using opencv function





C) Build a truncated Model:

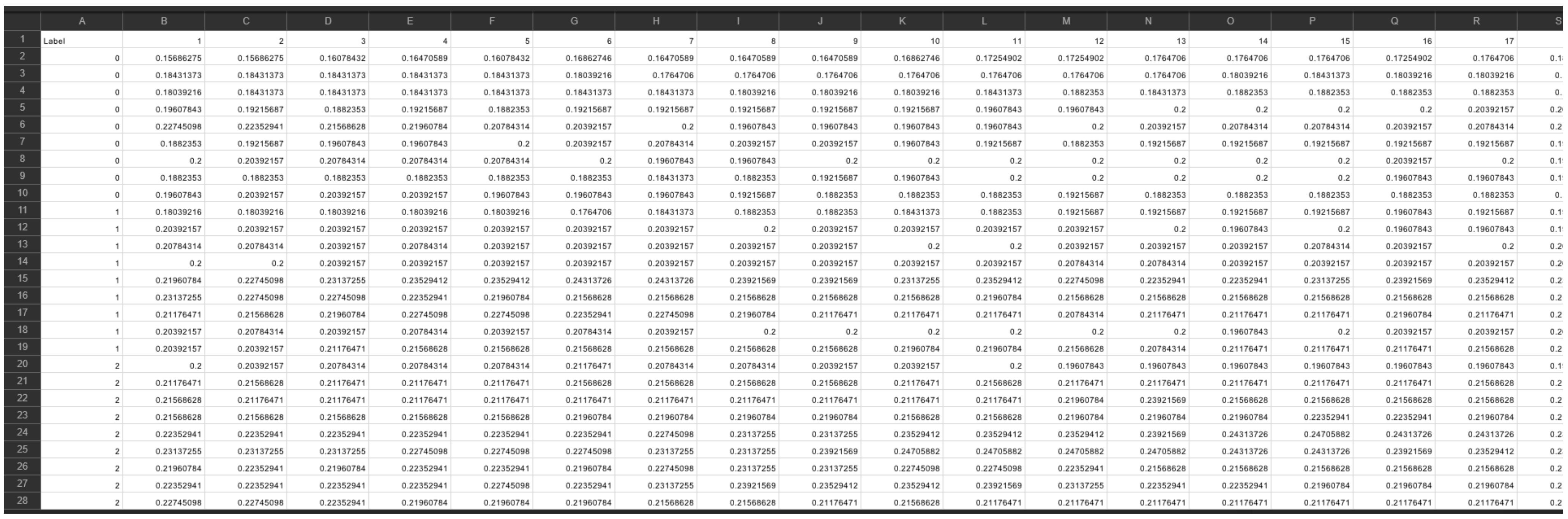
Built a truncated layer which takes only the first two convolutional layers, created a submodel set to eval and created an object for the same. Applied the truncated model to first image and printed out the 10 4*4 filters as well as its impact on the input image.



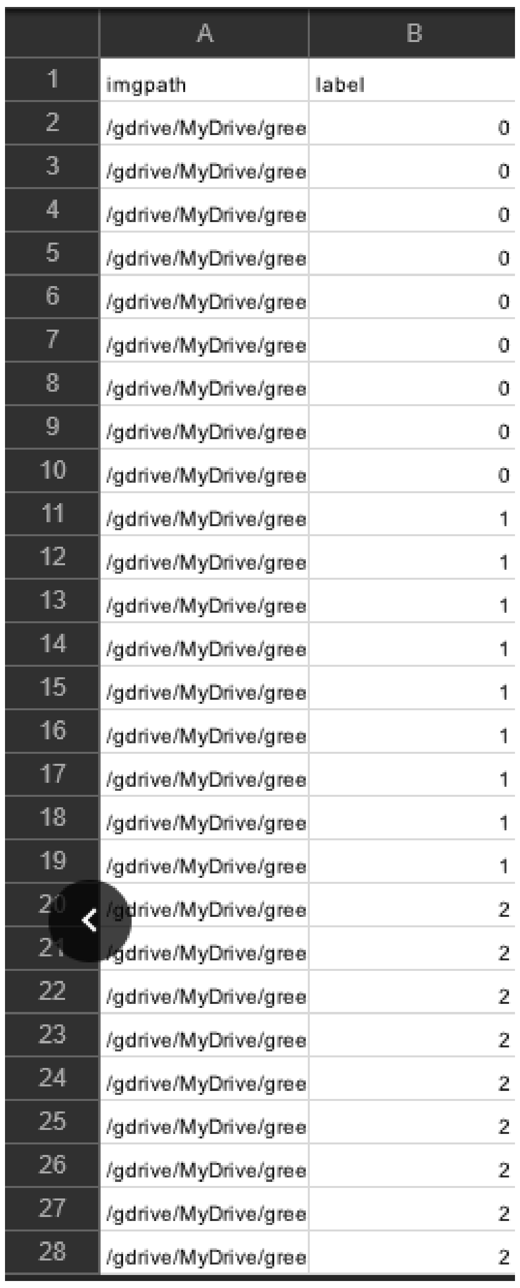
Task 3: Create a digit embedding space:

Task A: Create a greek symbol dataset

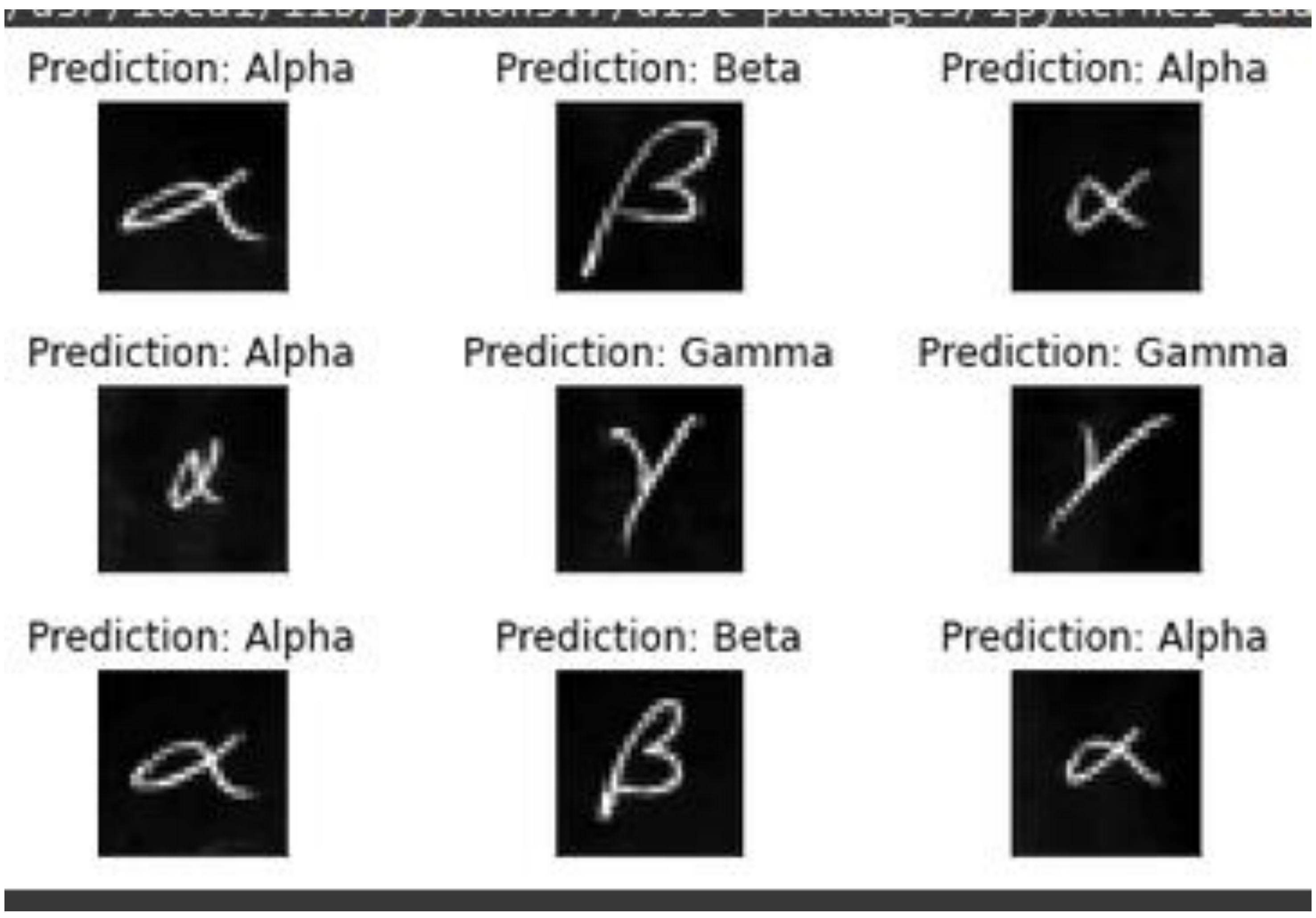
Created a dataset of greek symbols using torchvision.dataset.ImageFolder, and used tensor trasnformations of resizing, inverting colors, and grayscale. Created a fucntion to Convert the tensor data to numpy, added image label and flattened the numpy to save 784 values for each image into a csv file. Created another function to save the image path and image label into another csv. Trained the model and saved the model into google drive. The function can read any number of new images and print out corresponding csv file.



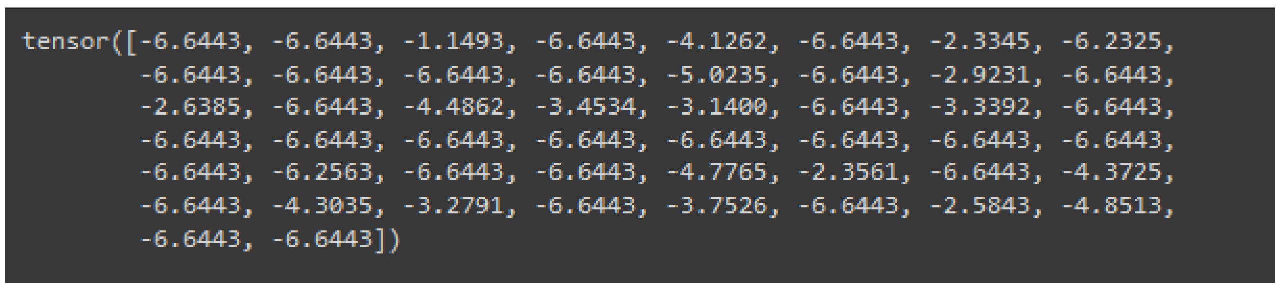
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Label	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
2	0	0.15686275	0.15686275	0.16078432	0.16470589	0.16078432	0.16862746	0.16470589	0.16470589	0.16470589	0.16862746	0.17254902	0.17254902	0.1764706	0.1764706	0.1764706	0.17254902	0.1764706	0.1
3	0	0.18431373	0.18431373	0.18431373	0.18431373	0.18431373	0.18039216	0.1764706	0.1764706	0.1764706	0.1764706	0.1764706	0.1764706	0.1764706	0.18039216	0.18431373	0.18039216	0.18039216	0.
4	0	0.18039216	0.18431373	0.18431373	0.18431373	0.18431373	0.18431373	0.18431373	0.18039216	0.18039216	0.18039216	0.18431373	0.1882353	0.18431373	0.1882353	0.1882353	0.1882353	0.1882353	0.
5	0	0.19607843	0.19215687	0.1882353	0.19215687	0.1882353	0.19215687	0.19215687	0.19215687	0.19215687	0.19215687	0.19607843	0.19607843	0.2	0.2	0.2	0.2	0.20392157	0.2
6	0	0.22745098	0.22352941	0.21568628	0.21960784	0.20784314	0.20392157	0.2	0.19607843	0.19607843	0.19607843	0.19607843	0.2	0.20392157	0.20784314	0.20784314	0.20392157	0.20784314	0.2
7	0	0.1882353	0.19215687	0.19607843	0.19607843	0.2	0.20392157	0.20784314	0.20392157	0.20392157	0.19607843	0.19215687	0.1882353	0.19215687	0.19215687	0.19215687	0.19215687	0.19215687	0.1
8	0	0.2	0.20392157	0.20784314	0.20784314	0.20784314	0.2	0.19607843	0.19607843	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.20392157	0.2	0.1
9	0	0.1882353	0.1882353	0.1882353	0.1882353	0.1882353	0.1882353	0.18431373	0.1882353	0.19215687	0.19607843	0.2	0.2	0.2	0.2	0.2	0.19607843	0.19607843	0.1
10	0	0.19607843	0.20392157	0.20392157	0.20392157	0.19607843	0.19607843	0.19607843	0.19215687	0.1882353	0.1882353	0.1882353	0.19215687	0.1882353	0.1882353	0.1882353	0.1882353	0.1882353	0.
11	1	0.18039216	0.18039216	0.18039216	0.18039216	0.18039216	0.1764706	0.18431373	0.1882353	0.1882353	0.18431373	0.1882353	0.19215687	0.19215687	0.19215687	0.19215687	0.19607843	0.19215687	0.1
12	1	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.2	0.20392157	0.20392157	0.20392157	0.20392157	0.2	0.19607843	0.2	0.19607843	0.19607843	0.1
13	1	0.20784314	0.20784314	0.20392157	0.20784314	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.2	0.2	0.20392157	0.20392157	0.20392157	0.20784314	0.20392157	0.2	0.2
14	1	0.2	0.2	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20392157	0.20784314	0.20784314	0.20392157	0.20392157	0.20392157	0.20392157	0.2
15	1	0.21960784	0.22745098	0.23137255	0.23529412	0.23529412	0.24313726	0.24313726	0.23921569	0.23921569	0.23137255	0.23529412	0.22745098	0.22352941	0.22352941	0.23137255	0.23921569	0.23529412	0.2
16	1	0.23137255	0.22745098	0.22745098	0.22352941	0.21960784	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.21960784	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.2
17	1	0.21176471	0.21568628	0.21960784	0.22745098	0.22745098	0.22352941	0.22745098	0.21960784	0.21176471	0.21176471	0.21176471	0.20784314	0.21176471	0.21176471	0.21176471	0.21960784	0.21176471	0.2
18	1	0.20392157	0.20784314	0.20392157	0.20784314	0.20392157	0.20784314	0.20392157	0.2	0.2	0.2	0.2	0.2	0.2	0.19607843	0.2	0.20392157	0.20392157	0.2
19	1	0.20392157	0.20392157	0.21176471	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.21960784	0.21960784	0.21568628	0.20784314	0.21176471	0.21176471	0.21176471	0.21568628	0.2
20	2	0.2	0.20392157	0.20784314	0.20784314	0.20784314	0.21176471	0.20784314	0.20784314	0.20392157	0.20392157	0.2	0.19607843	0.19607843	0.19607843	0.19607843	0.19607843	0.19607843	0.1
21	2	0.21176471	0.21568628	0.21176471	0.21176471	0.21176471	0.21568628	0.21568628	0.21568628	0.21568628	0.21176471	0.21568628	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21568628	0.2
22	2	0.21568628	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21960784	0.23921569	0.21568628	0.21568628	0.21568628	0.21568628	0.2
23	2	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.21960784	0.21960784	0.21960784	0.21960784	0.21568628	0.21568628	0.21960784	0.21960784	0.21960784	0.22352941	0.22352941	0.21960784	0.2
24	2	0.22352941	0.22352941	0.22352941	0.22352941	0.22352941	0.22352941	0.22745098	0.23137255	0.23137255	0.23529412	0.23529412	0.23529412	0.23921569	0.24313726	0.24705882	0.24313726	0.24313726	0.2
25	2	0.23137255	0.23137255	0.23137255	0.22745098	0.22745098	0.22745098	0.23137255	0.23137255	0.23921569	0.24705882	0.24705882	0.24705882	0.24705882	0.24313726	0.24313726	0.23921569	0.23529412	0.2
26	2	0.21960784	0.22352941	0.21960784	0.22352941	0.22352941	0.21960784	0.22745098	0.23137255	0.23137255	0.22745098	0.22745098	0.22352941	0.21568628	0.21568628	0.21568628	0.21568628	0.21568628	0.2
27	2	0.22352941	0.22352941	0.22352941	0.22352941	0.22745098	0.22352941	0.23137255	0.23921569	0.23529412	0.23529412	0.23921569	0.23137255	0.22352941	0.22352941	0.21960784	0.21960784	0.21960784	0.2
28	2	0.22745098	0.22745098	0.22352941	0.21960784	0.21960784	0.21960784	0.21568628	0.21568628	0.21176471	0.21568628	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.21176471	0.2



	A	B
1	imgpath	label
2	/gdrive/MyDrive/gree	0
3	/gdrive/MyDrive/gree	0
4	/gdrive/MyDrive/gree	0
5	/gdrive/MyDrive/gree	0
6	/gdrive/MyDrive/gree	0
7	/gdrive/MyDrive/gree	0
8	/gdrive/MyDrive/gree	0
9	/gdrive/MyDrive/gree	0
10	/gdrive/MyDrive/gree	0
11	/gdrive/MyDrive/gree	1
12	/gdrive/MyDrive/gree	1
13	/gdrive/MyDrive/gree	1
14	/gdrive/MyDrive/gree	1
15	/gdrive/MyDrive/gree	1
16	/gdrive/MyDrive/gree	1
17	/gdrive/MyDrive/gree	1
18	/gdrive/MyDrive/gree	1
19	/gdrive/MyDrive/gree	1
20	gdrive/MyDrive/gree	2
21	gdrive/MyDrive/gree	2
22	/gdrive/MyDrive/gree	2
23	/gdrive/MyDrive/gree	2
24	/gdrive/MyDrive/gree	2
25	/gdrive/MyDrive/gree	2
26	/gdrive/MyDrive/gree	2
27	/gdrive/MyDrive/gree	2
28	/gdrive/MyDrive/gree	2



Task B) Create a truncated Model: Tensor of 50 numbers as output



```
tensor([-6.6443, -6.6443, -1.1493, -6.6443, -4.1262, -6.6443, -2.3345, -6.2325,
        -6.6443, -6.6443, -6.6443, -6.6443, -5.0235, -6.6443, -2.9231, -6.6443,
        -2.6385, -6.6443, -4.4862, -3.4534, -3.1400, -6.6443, -3.3392, -6.6443,
        -6.6443, -6.6443, -6.6443, -6.6443, -6.6443, -6.6443, -6.6443, -6.6443,
        -6.6443, -6.2563, -6.6443, -6.6443, -4.7765, -2.3561, -6.6443, -4.3725,
        -6.6443, -4.3035, -3.2791, -6.6443, -3.7526, -6.6443, -2.5843, -4.8513,
        -6.6443, -6.6443])
```

Task C) Project the Greek Symbol into the truncated space:

Got 27 output of 50 numbers when the model was applied to the csv file originally stored

Task D) Compute the SSD values

Computed the SSD values of all 27 images with each other. The distance for images of each type(alpha, beta, gamma) was close to their own type.

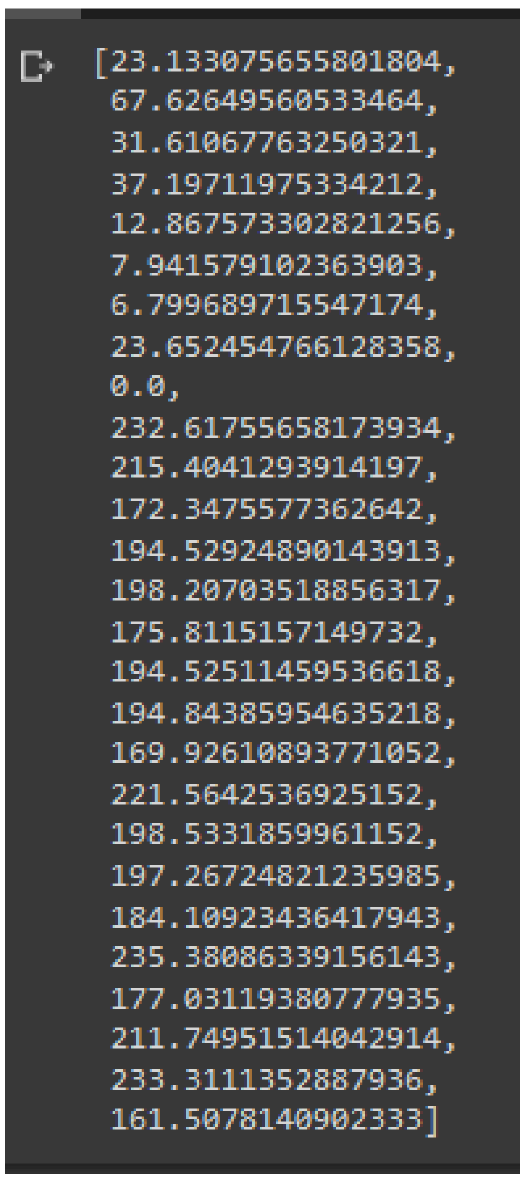
The SSD distance of image to itself was 0, whereas its distance to other alpha image was 0-50. Similar case was observed with beta and gamma

I found that the model did a better job than KNN would have as I am able to categorize each image

Task E) Create your own symbol

Alpha SSD

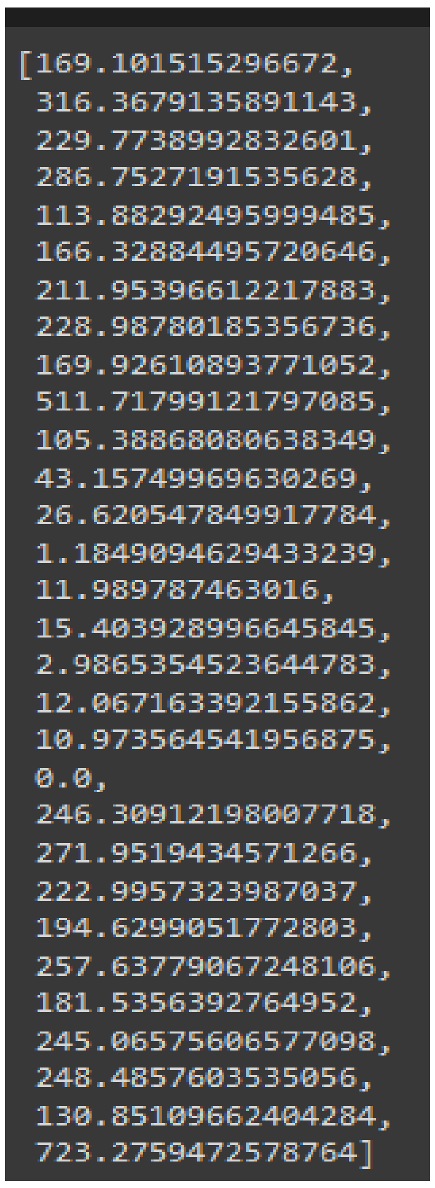
The first 10 values are all gamma



```
[23.133075655801804,
 67.62649560533464,
 31.61067763250321,
 37.19711975334212,
 12.867573302821256,
 7.941579102363903,
 6.799689715547174,
 23.652454766128358,
 0.0,
 232.61755658173934,
 215.4041293914197,
 172.3475577362642,
 194.52924890143913,
 198.20703518856317,
 175.8115157149732,
 194.52511459536618,
 194.84385954635218,
 169.92610893771052,
 221.5642536925152,
 198.5331859961152,
 197.26724821235985,
 184.10923436417943,
 235.38086339156143,
 177.03119380777935,
 211.74951514042914,
 233.3111352887936,
 161.5078140902333]
```

Beta SSD

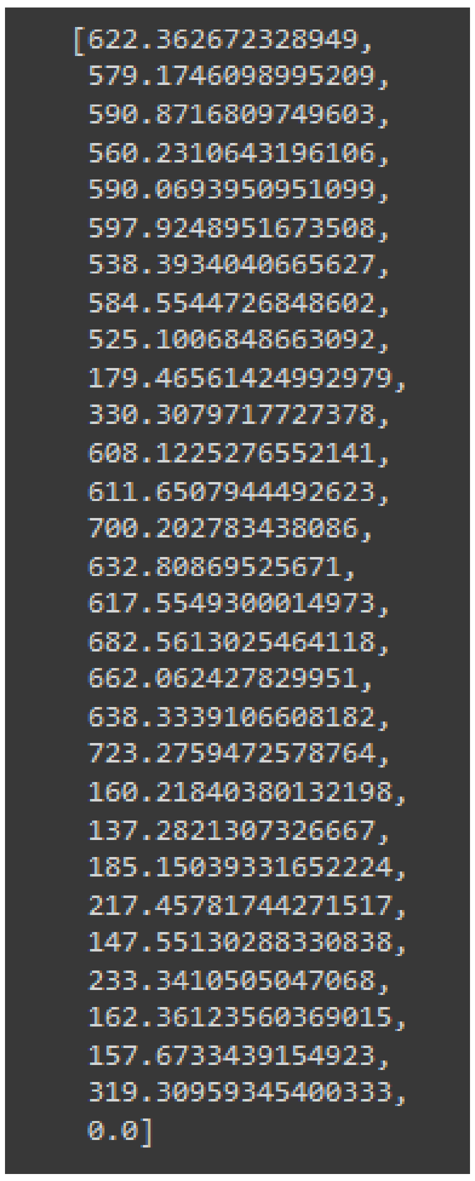
The first 10 values are all alpha images



```
[169.101515296672,
 316.3679135891143,
 229.7738992832601,
 286.7527191535628,
 113.88292495999485,
 166.32884495720646,
 211.95396612217883,
 228.98780185356736,
 169.92610893771052,
 511.71799121797085,
 105.38868080638349,
 43.15749969630269,
 26.620547849917784,
 1.1849094629433239,
 11.989787463016,
 15.403928996645845,
 2.9865354523644783,
 12.067163392155862,
 10.973564541956875,
 0.0,
 246.30912198007718,
 271.9519434571266,
 222.9957323987037,
 194.6299051772803,
 257.63779067248106,
 181.5356392764952,
 245.06575606577098,
 248.4857603535056,
 130.85109662404284,
 723.2759472578764]
```

gamma SSD

The last 10 values are all beta images



```
[622.362672328949,
 579.1746098995209,
 590.8716809749603,
 560.2310643196106,
 590.0693950951099,
 597.9248951673508,
 538.393404066562 7,
 584.5544726848602,
 525.1006848663092,
 179.46561424992979,
 330.3079717727378,
 608.1225276552141,
 611.6507944492623,
 700.202783438086,
 632.80869525671,
 617.5549300014973,
 682.5613025464118,
 662.062427829951,
 638.3339106608182,
 723.2759472578764,
 160.21840380132198,
 137.2821307326667,
 185.15039331652224,
 217.45781744271517,
 147.55130288330838,
 233.3410505047068,
 162.36123560369015,
 157.6733439154923,
 319.30959345400333,
 0.0]
```

The new Greek symbols all passed the test of SSD, The distance between each type of image is less as observed(gamma values were a bit off), by this we can confirm that the new symbols do match the existing ones

Task 4: Design your own experiment:

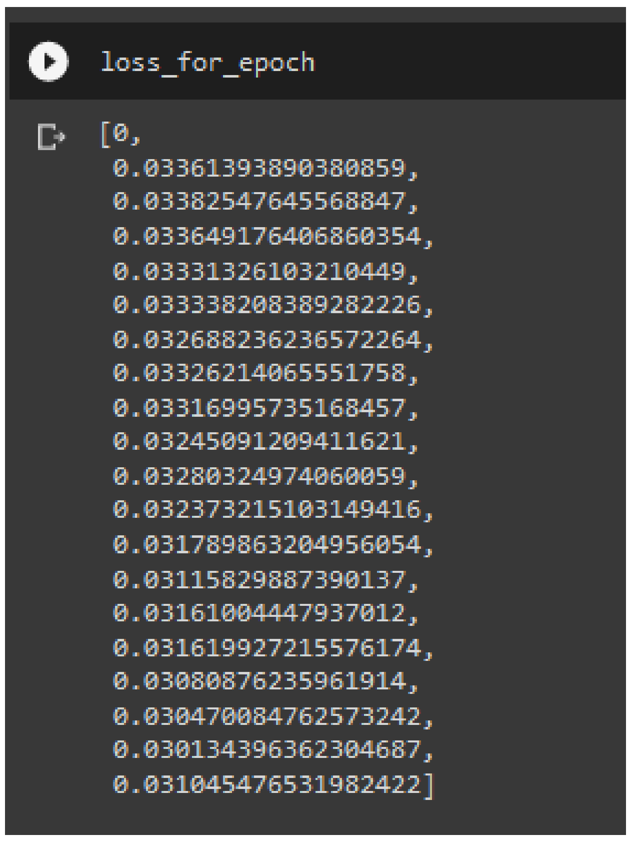
A) The metric I used for these experiments were:

1) Number of epochs: range(1-20)
2) Batch Size = 500 to 10000 with increments of 500 batches (total 20)

B and C) Predict and evaluate the result

Epochs: more epochs will result in less loss but will cause overfitting:

Loss table for increasing number of epochs: As predicted we got less loss for increasing the number of epochs

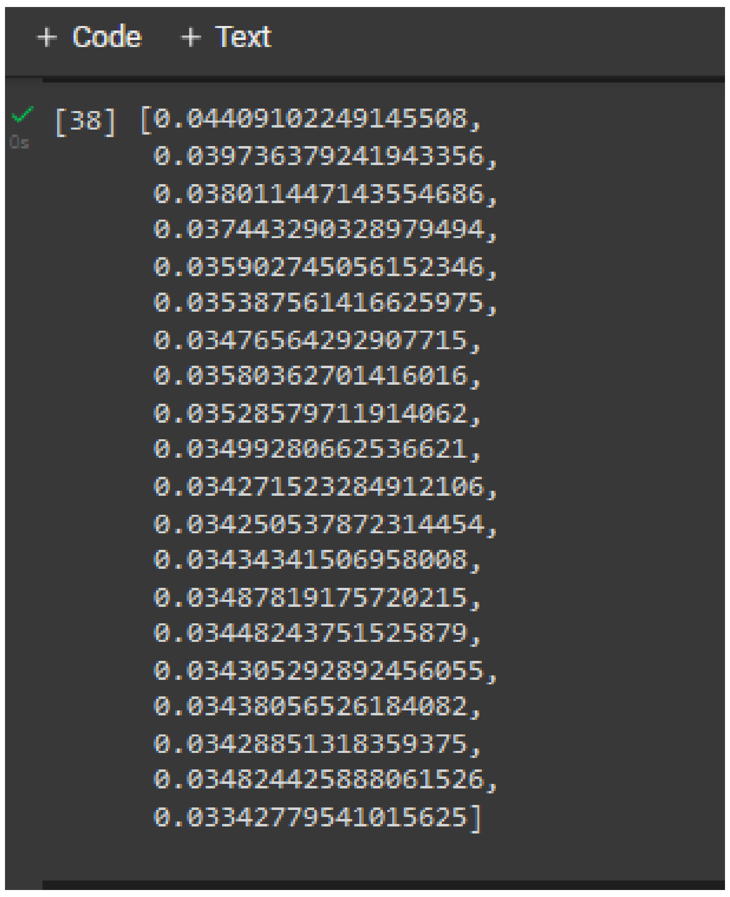


```
loss_for_epoch

[0,
 0.03361393890380859,
 0.03382547645568847,
 0.033649176406860354,
 0.03331326103210449,
 0.033338208389282226,
 0.032688236236572264,
 0.03326214065551758,
 0.03316995735168457,
 0.03245091209411621,
 0.03280324974060059,
 0.032373215103149416,
 0.031789863204956054,
 0.03115829887390137,
 0.03161004447937012,
 0.031619927215576174,
 0.03080876235961914,
 0.030470084762573242,
 0.030134396362304687,
 0.031045476531982422]
```

Batch Size: Similar to epochs

I kept the batch size high initially cause it was taking forever to train the network with low batch size as I incremented the batch size 20 times for each epoch

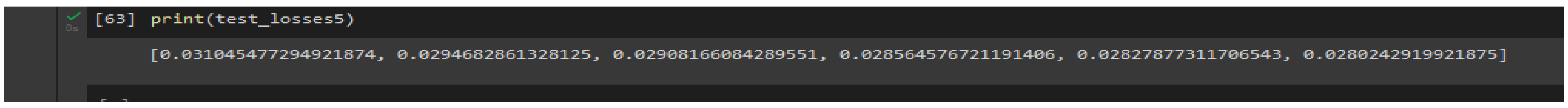


```
+ Code   + Text

[38] [0.04409102249145508,
      0.039736379241943356,
      0.038011447143554686,
      0.037443290328979494,
      0.035902745056152346,
      0.035387561416625975,
      0.03476564292907715,
      0.03580362701416016,
      0.03528579711914062,
      0.03499280662536621,
      0.034271523284912106,
      0.034250537872314454,
      0.03434341506958008,
      0.03487819175720215,
      0.03448243751525879,
      0.034305292892456055,
      0.03438056526184082,
      0.03428851318359375,
      0.034824425888061526,
      0.03342779541015625]
```

Second Dropout Layer after fc2:

We can place dropout layer anywhere but its better to place it after fully connected layer as it has more number of nodes



```
[63] print(test_losses5)

     [0.031045477294921874, 0.0294682861328125, 0.02908166084289551, 0.028564576721191406, 0.02827877311706543, 0.0280242919921875]
```

Conclusion: The project involved diving deep into the CNN architecture, how it makes filters, used different kernels, reduces filter size by max pooling and classifies each image and improves itself overtime. It also introduced us to how the filters are created and their purpose in classification. How tensors work and how we can compute distances to finally check what the classifiers are meant to do.

Acknowledgements:

https://pytorch.org/tutorials/beginner/basics/intro.html

https://pytorch.org/

#lot of stackoverflow for creating custom datasets