Project 5: Recognition using Deep Networks - Report

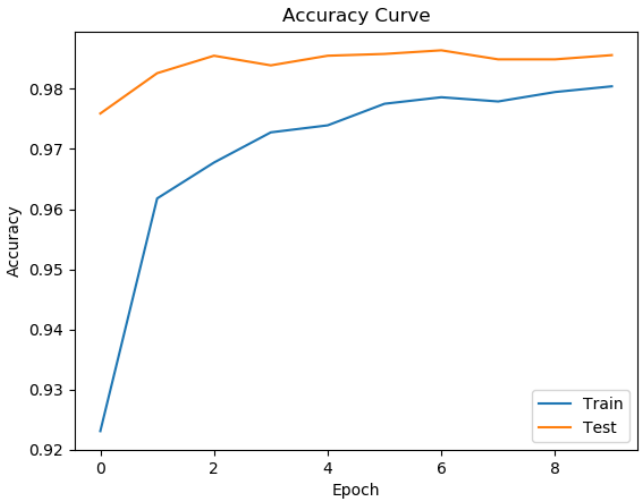
Short Description

In this project, we built a convolutional neural network model for Digit recognition. Then learned to use the built embedding space learned by a model on set of English digits to classify Greek letters. Also visualizing the activations and weights of hidden layers helped us understand how a network learns the model. Then we experimented with tuning different parameters of the models like, batch size, number of convolutional layers, filter size, activation functions etc. to do the empirical observations of each method and experiment.

Task 1

- A - Image of the first two example digits

- C - The output of the python program is in the file out.txt in the task1 folder.
- D - Training and testing accuracy graph



- E - The model is saved in the file my_model.h5 in the task1 folder. This is the summary of the model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 4608)              0
_________________________________________________________________
dense (Dense)                (None, 128)               589952
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 600,810
Trainable params: 600,810
Non-trainable params: 0
_________________________________________________________________
```

- E - The output for the first 10 test images (can be found at the end of out.txt file)

```
The predicted digit is 7 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0] and the correct digit 7
The predicted digit is 2 with the class probabilities [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 2
The predicted digit is 1 with the class probabilities [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 1
The predicted digit is 0 with the class probabilities [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 0
The predicted digit is 4 with the class probabilities [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 4
The predicted digit is 1 with the class probabilities [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 1
The predicted digit is 4 with the class probabilities [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 4
The predicted digit is 9 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0] and the correct digit 9
The predicted digit is 5 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.87, 0.13, 0.0, 0.0, 0.0] and the correct digit 5
The predicted digit is 9 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0] and the correct digit 9
```

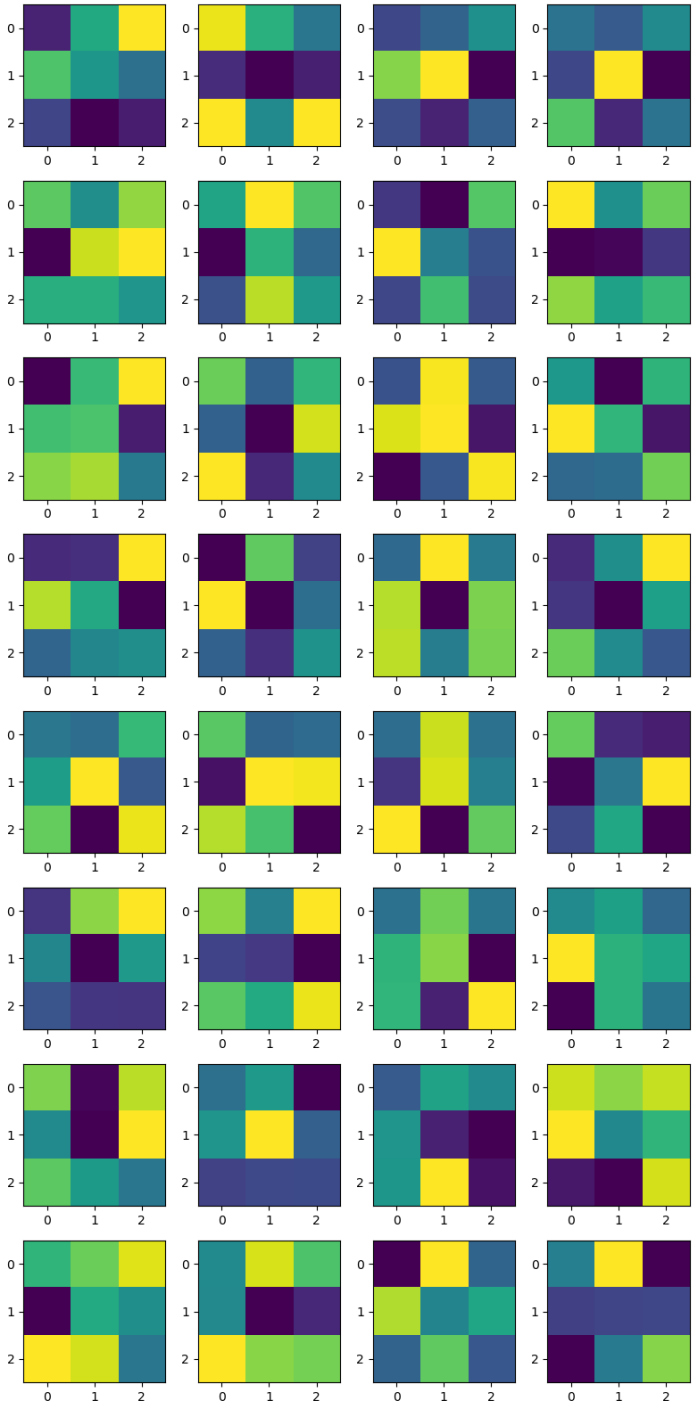
- F - The output for custom test files (can be found at the end of out.txt file)

```
The predicted digit is 0 with the class probabilities [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 0
The predicted digit is 1 with the class probabilities [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 1
The predicted digit is 2 with the class probabilities [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 2
The predicted digit is 3 with the class probabilities [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 3
The predicted digit is 4 with the class probabilities [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 4
The predicted digit is 5 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.96, 0.0, 0.0, 0.0, 0.04] and the correct digit 5
The predicted digit is 6 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0] and the correct digit 6
The predicted digit is 7 with the class probabilities [0.0, 0.0, 0.12, 0.1, 0.0, 0.0, 0.0, 0.78, 0.0, 0.0] and the correct digit 7
The predicted digit is 8 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0] and the correct digit 8
The predicted digit is 9 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0] and the correct digit 9
```

- F - These are the custom files used

Task 2

- B - The weights of the first layer when plotted



- B - The values of the weights of the first layer (compare to the plots in a row major order).

```
[[-0.09646592  0.04388453  0.15326902]
 [ 0.0765382   0.02255454 -0.0220204 ]
 [-0.0662495  -0.12421463 -0.10294966]]

[[ 0.06694997  0.02106617 -0.01238082]
 [-0.04875608 -0.06645638 -0.05442373]
 [ 0.07118979 -0.00064594  0.07121831]]

[[-0.03179614 -0.01371307  0.01871933]
 [ 0.0751766   0.1070129  -0.06988493]
 [-0.02794624 -0.05330068 -0.01591975]]
```

```
[[-0.00336326 -0.01542271  0.00887306]
 [-0.02419522  0.0750742  -0.05191009]
 [ 0.04108505 -0.03736936 -0.00373846]]

[[ 0.06942571  0.01673607  0.08762891]
 [-0.08547939  0.10484221  0.12150645]
 [ 0.04471939  0.04531214  0.02243523]]

[[ 0.00790377  0.14710456  0.05590374]
 [-0.18805197  0.02881619 -0.07646387]
 [-0.10503182  0.11252618 -0.00626615]]

[[-0.03733929 -0.06246521  0.05184721]
 [ 0.09292512  0.00371196 -0.02318851]
 [-0.02957212  0.04573593 -0.02689629]]

[[ 0.08347263  0.00710922  0.04888682]
 [-0.06928213 -0.06712811 -0.04458132]
 [ 0.05804966  0.0179424   0.03436492]]

[[-0.14164783  0.01772071  0.09429454]
 [ 0.02270898  0.0273172  -0.12388757]
 [ 0.0519858   0.063219   -0.04626216]]

[[ 0.11960521 -0.0979156   0.06538083]
 [-0.09859669 -0.24353638  0.19415544]
 [ 0.22560768 -0.19195244 -0.01826039]]

[[-0.03270433  0.05751356 -0.02899747]
 [ 0.05249178  0.05878052 -0.05648655]
 [-0.06378205 -0.02992551  0.05741747]]

[[ 0.01055213 -0.08032571  0.03054798]
 [ 0.08992142  0.03171091 -0.07000522]
 [-0.02302068 -0.02007564  0.05382502]]

[[-0.09877044 -0.09376534  0.15926546]
 [ 0.12622873  0.0426769  -0.13386603]
 [-0.03830044  0.0005558   0.01044799]]

[[-0.09935918  0.09256721 -0.05004956]
 [ 0.15412566 -0.09922141 -0.00762783]
 [-0.02028011 -0.06491474  0.03086017]]

[[-0.0374957   0.05133113 -0.02829271]
 [ 0.03627348 -0.08342738  0.02461814]
 [ 0.03776312 -0.02705222  0.02357564]]

[[-0.06577425  0.00817384  0.10726909]
 [-0.05912755 -0.0900782   0.02161041]
 [ 0.06319723  0.00501799 -0.03620623]]

[[-0.00957911 -0.01171827  0.00522501]
```

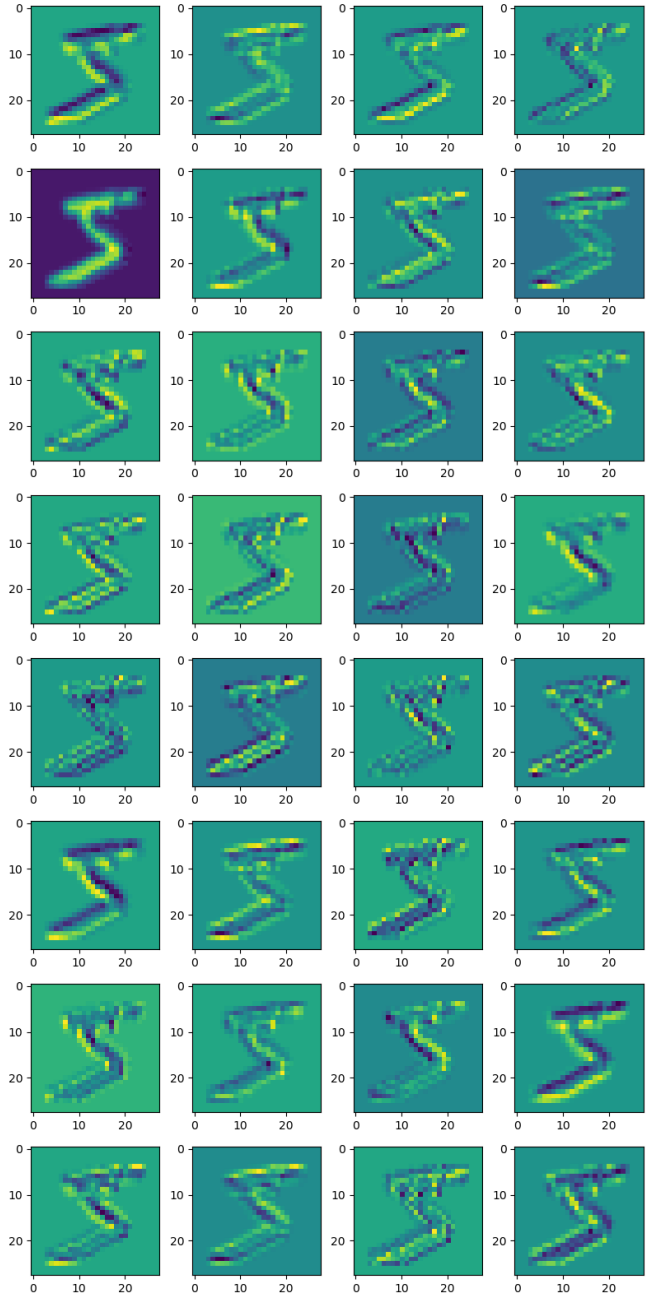
```
 [-0.00063575  0.02340971 -0.01617041]
 [ 0.01051049 -0.03139581  0.02173098]]

[[ 0.04304597 -0.05139913 -0.04517453]
 [-0.11304355  0.1001932   0.0961825 ]
 [ 0.07525034  0.03500593 -0.12374169]]

[[-0.05772569  0.10223331 -0.05292481]
 [-0.1158317   0.10758803 -0.03627886]
 [ 0.12484387 -0.15922669  0.05654709]]

[[ 0.07712019 -0.0456727  -0.05322045]
 [-0.06753602  0.00676985  0.12294096]
 [-0.0273067   0.04569406 -0.06974021]]

[[-0.08482764  0.11095347  0.16193596]
 [ 0.00466504 -0.12918708  0.02686302]
 [-0.05240298 -0.08274443 -0.08391188]]

[[ 0.05563399 -0.02077902  0.0887783 ]
 [-0.06546367 -0.07275677 -0.10471006]
 [ 0.03919862  0.01323379  0.08321404]]

[[-0.029216    0.03728519 -0.02692805]
 [ 0.01538299  0.04294206 -0.08998539]
 [ 0.01631729 -0.07533216  0.07197212]]

[[-0.00447867  0.0041965  -0.01814256]
 [ 0.04414186  0.01093728  0.00576189]
 [-0.04877922  0.0108035  -0.01282634]]

[[ 0.0394453  -0.09388102  0.05484745]
 [-0.01664339 -0.09709735  0.07203701]
 [ 0.02923425 -0.00480235 -0.03045583]]

[[-0.00315184  0.01324891 -0.03827322]
 [ 0.01157081  0.057239   -0.00890754]
 [-0.01990899 -0.01715852 -0.01670283]]

[[-0.01207178  0.02522201  0.01175946]
 [ 0.01815844 -0.03748189 -0.04913823]
 [ 0.01919696  0.07966689 -0.04274692]]

[[ 0.06260698  0.04022792  0.05969131]
 [ 0.08051697 -0.04276967  0.00042766]
 [-0.13635209 -0.1515193   0.06544473]]

[[-0.00123935  0.01510054  0.03933048]
 [-0.09087242 -0.00730406 -0.022687  ]
 [ 0.04579875  0.03619427 -0.03718005]]

[[-0.01896643  0.07036304  0.02806736]
 [-0.0206453  -0.11174268 -0.09047304]
```

[0.08236261 0.04759531 0.04187178]]

[[-0.09846691 0.08906594 -0.03791817]
 [0.06689656 -0.01355527 0.0117174]
 [-0.03901952 0.04356601 -0.04781342]]

[[0.0056793   0.06581161 -0.03965715]
 [-0.02002333 -0.01793327 -0.01748182]
 [-0.04000673 0.0035089   0.04662318]]

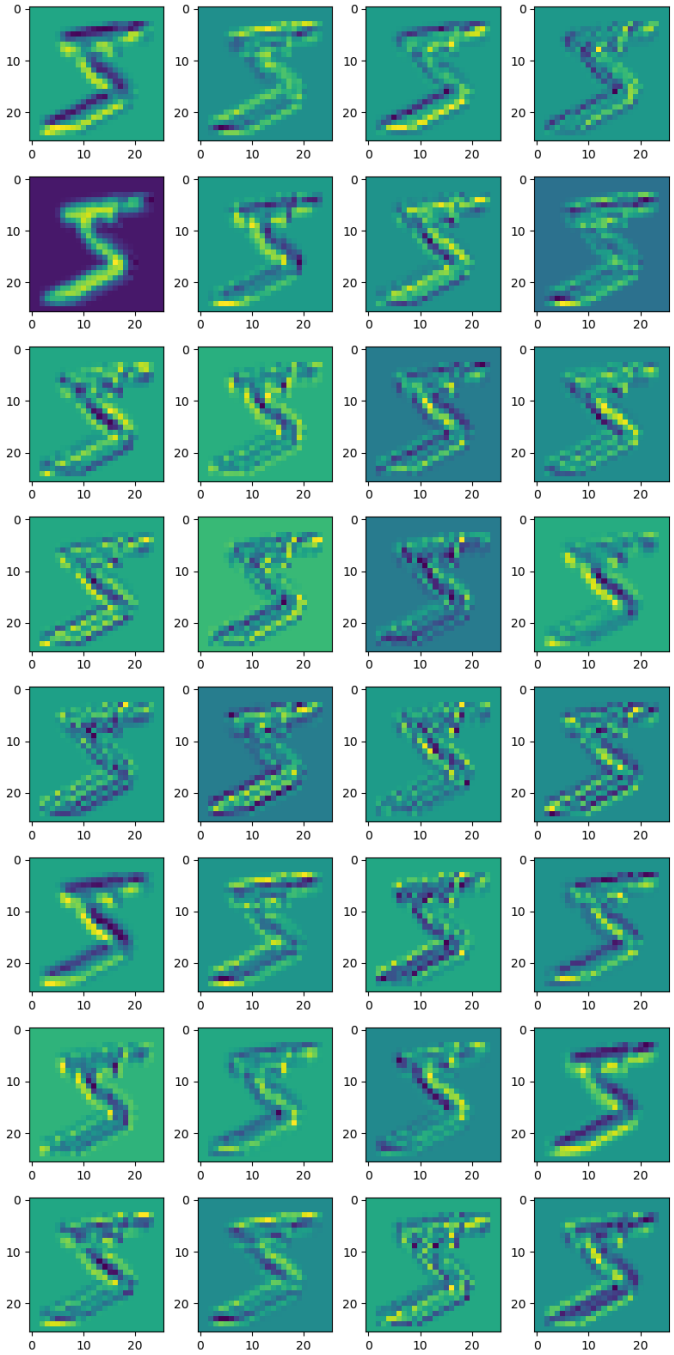
- C - The effect of the filter to the first example



The bluer the color the more negative the value is, the yellower it is the more positive it is, and green is in the middle. This can be confirmed from the filter values.
Keeping the above convention in mind these results do make sense. Why? Because we can see that most of these filters are trying to detect some kind of edge like features. Take the third kernel in the first row for example, it has smaller values on the bottom right corner and higher values in the middle, so it can be roughly thought of as a filter which would detect edges at a 45 degree angle with the x-axis, and if we look at the corresponding filtered output, that is exactly what we see.

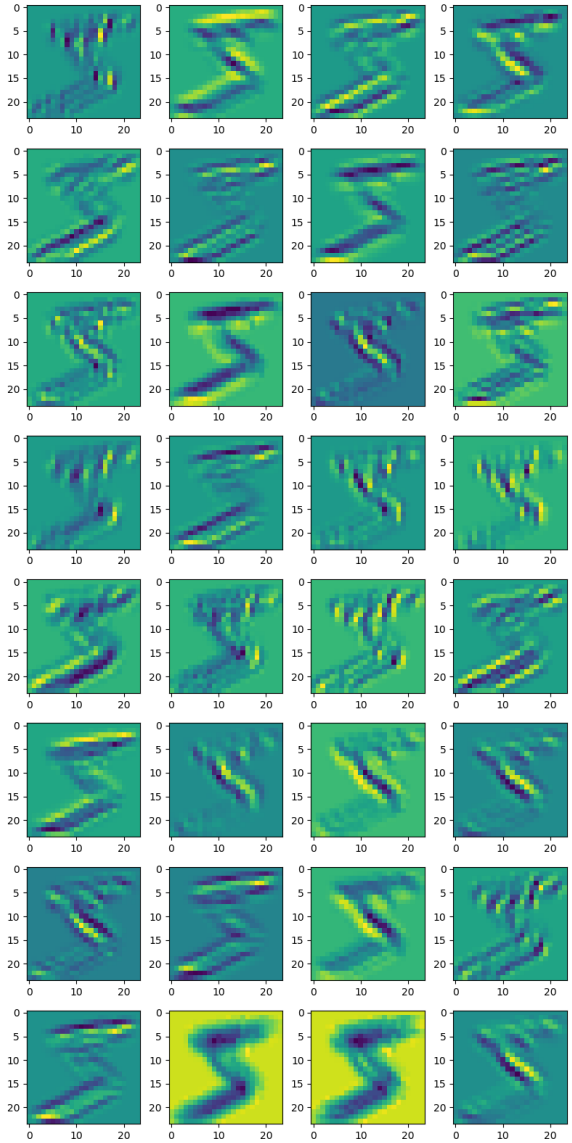
A lot of the kernels can be explained in a similar fashion, though not all, as some of the kernels just seem to have random patterns which don't look anything like edge detectors, but still the outputs seem to make sense for most of these cases too.

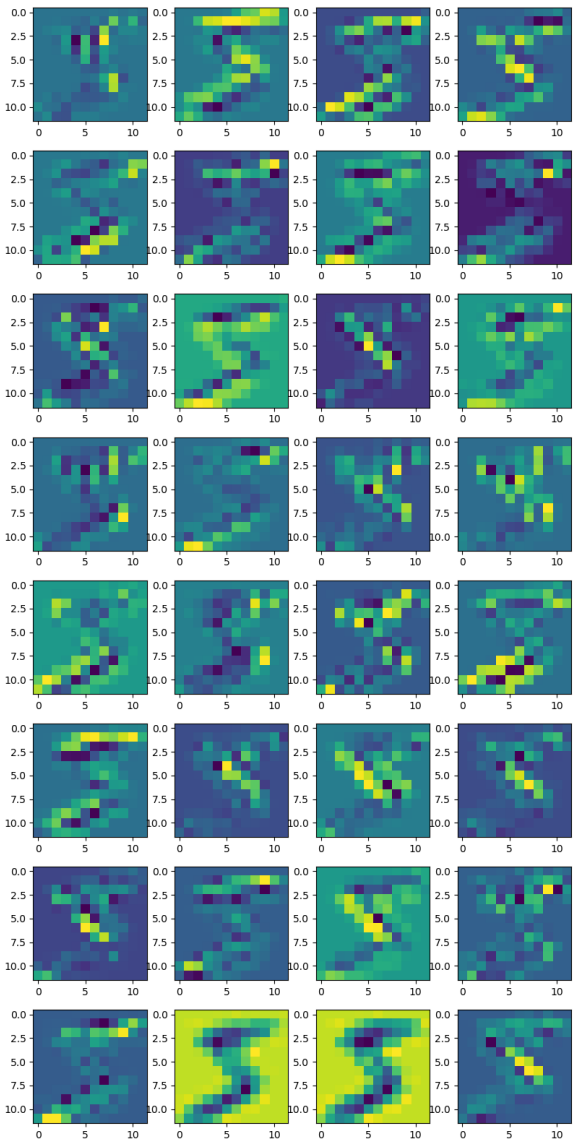
- D - The output of the truncated model



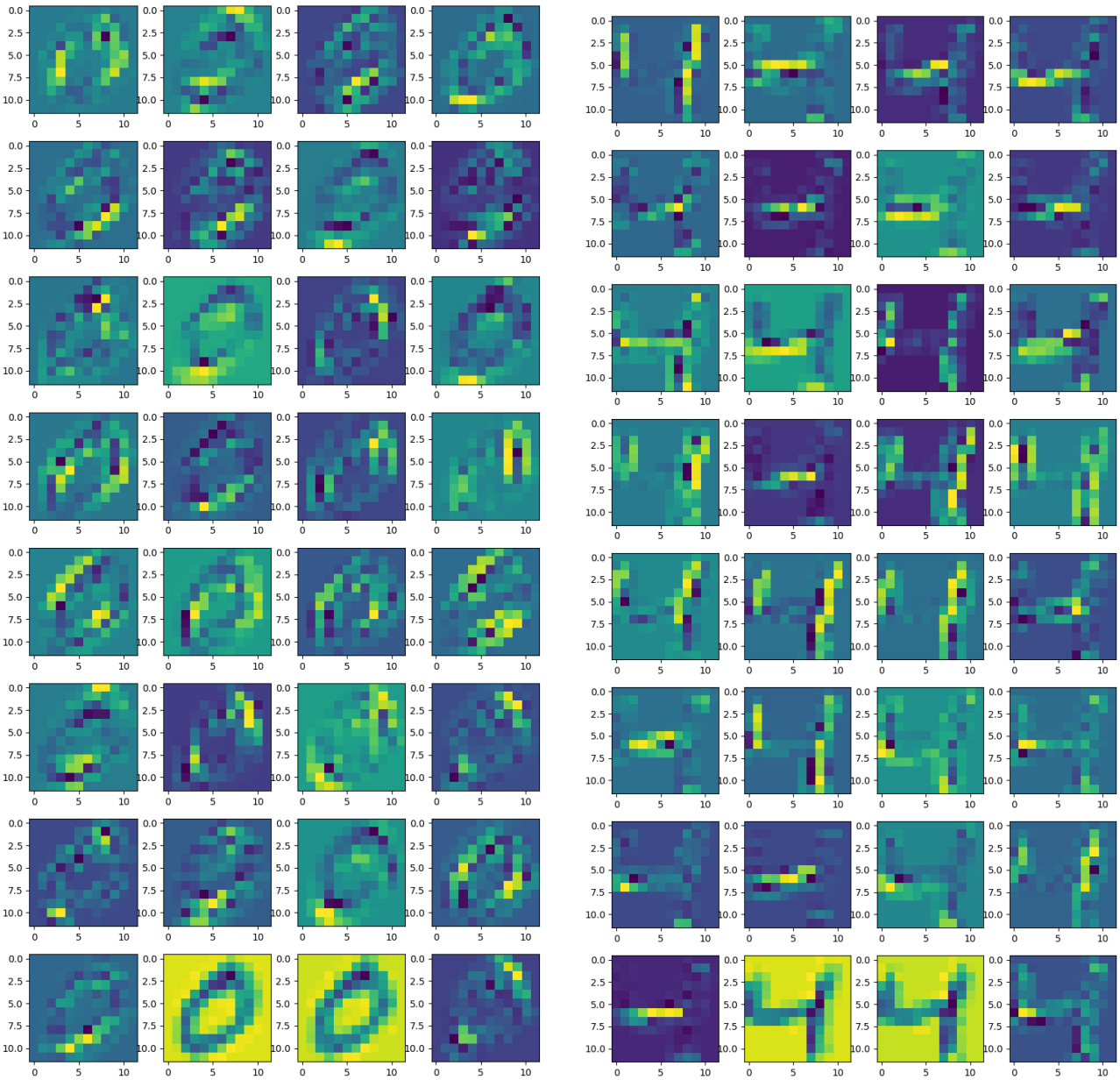
The results of applying the truncated model of just the first layer is the exact same as the output by applying the filters using the open cv functions. This is because the model is essentially doing the same thing, that is applying those filters over the image i.e. convolving the filters with the image. The only difference between both the outputs is that the output of the model is a little roomed in towards the center of the image. This is because we're not doing any padding while applying the convolution layer, which results in the edges of the images being chopped off a little.

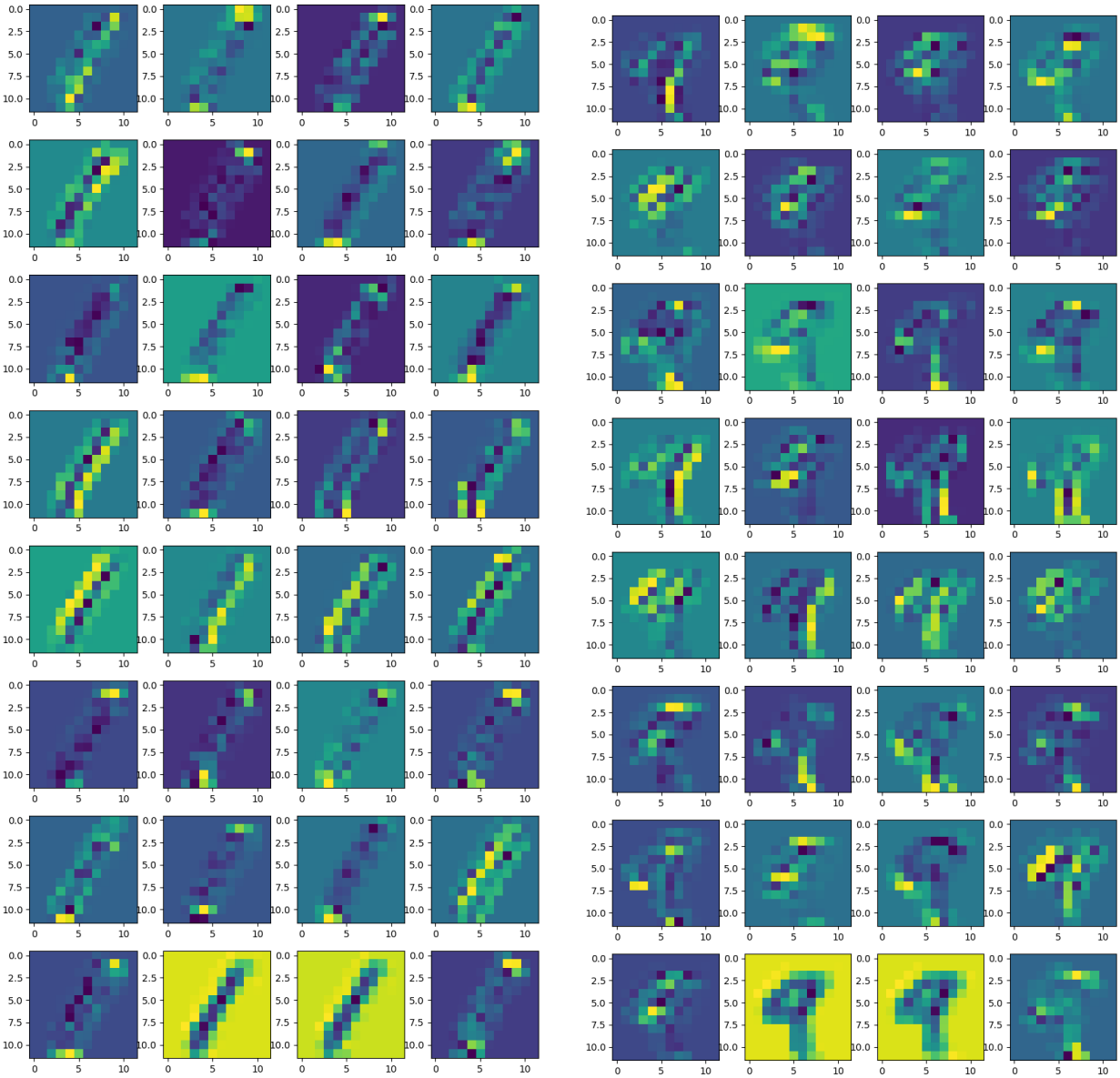
- E - The output after the second conv layer and the pool layer respectively

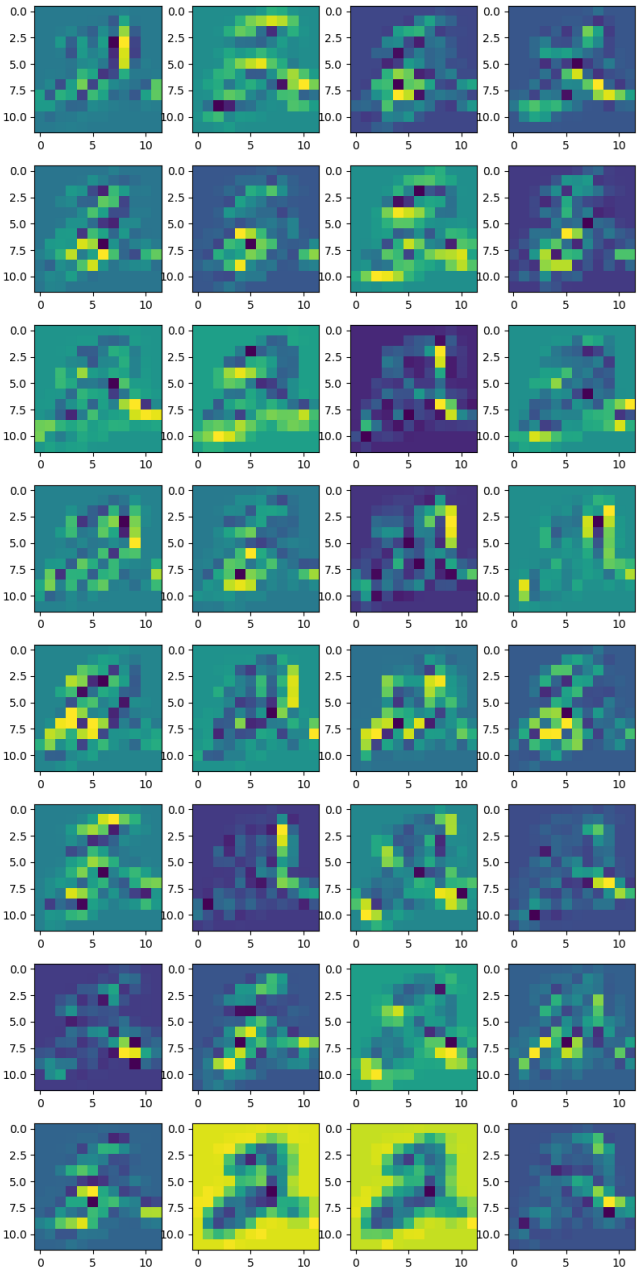




- E – More pool layer outputs







Looking at these outputs it seems like the output of most of the filters correspond to detecting a localized feature like detecting a vertical edge towards the right side of the image or detecting a round curve in the bottom right side of the image or horizontal edges at the top side of the image. For example – look at the outputs in the last column of the fifth row, it only lights up for the digits 5 and 0 (and very little in 2 as well) and that too only for the curvature in the bottom right quarter of the images, so it may be concluded that it's detecting the curvature in the bottom right quarter of the inputs. Similar conclusions can be made for most other outputs as well.

Task 3

- A – The file DataMaker.py contains the code for this part. The way this code is written it'll work with any number of categories and any number of examples. Provided that the files have are following the naming convention category name followed by an underscore, the rest could be anything. All you have to do is add in those files in the folder greek inside the task3 folder.
- B – to prove that the output is of size 128 we can do two things, one is to look at the model summary and verify that the last layer is a dense layer with 128 nodes or we can pass one example through the model and look at the shape of the output. Both are pasted below and can also be found in the out.txt file for task3

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_input (InputLayer)    [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 4608)              0
_________________________________________________________________
dense (Dense)                (None, 128)               589952
=================================================================
Total params: 599,520
Trainable params: 599,520
Non-trainable params: 0
_________________________________________________________________
```

The model summary:

The output of the program for the following line of code

```
# prints (1, 128) and ensures that the output is a vector of size 128
print(truncated_model.predict(data[0:1]).shape)
```

```
(1, 128)
```

- D – The following are the output when we calculate the ssd of the embedding of the first image with the rest of the embeddings. They are intentionally shown in the groups of 9. The first group of 9 corresponds to the embeddings for the letter alpha and the second and third correspond to beta and gamma respectively.

0.0	206.12534	257.08633
44.969795	141.85529	165.8956
98.630844	138.67554	183.57059
115.42185	178.18578	243.93913
135.18132	170.52208	222.12512
83.23733	172.91315	190.62814
99.29543	143.63086	175.11551
100.8486	168.37672	205.18463
135.71243	209.78075	218.69359

The pattern we find here is that the values in the first column are generally smaller than the values in the second and third columns. This was expected because an embedding for the letter alpha should be closer to other embeddings for the letter alpha when compared to the other letters if the embeddings are calculated are to be any useful. Though some of the values in the first column are still a bit big, this could be explained by the vastly different ways different people write the same letter.

- E – writing for this part before D because it's used in the explanation for D. The examples used

- E – The output of the program for these examples
  1. The predicted class for this test example is alpha with the least ssd being 760.92596
     And the sorted order of the ssds being [(760.92596, 'alpha'), (764.24567, 'alpha'), (767.6239, 'alpha'), (777.9092, 'alpha'), (778.0696, 'beta'), (795.68274, 'alpha'), (797.49554, 'beta'), (797.58044, 'beta'), (803.1222, 'alpha'), (803.3125, 'alpha'), (805.5499, 'gamma'), (809.4066, 'beta'), (812.85394, 'beta'),

(814.4122, 'alpha'), (823.8227, 'beta'), (831.39246, 'beta'), (832.56335, 'alpha'), (850.4369, 'beta'), (854.56573, 'gamma'), (855.3436, 'gamma'), (863.0204, 'gamma'), (867.57587, 'gamma'), (884.4514, 'gamma'), (892.9863, 'gamma'), (900.6026, 'gamma'), (900.66016, 'beta'), (915.2759, 'gamma')]

2. The predicted class for this test example is alpha with the least ssd being 1218.0718
   And the sorted order of the ssds being [(1218.0718, 'alpha'), (1225.8298, 'alpha'), (1297.9861, 'alpha'), (1318.3754, 'beta'), (1320.9689, 'beta'), (1325.7136, 'alpha'), (1326.834, 'alpha'), (1327.7542, 'beta'), (1329.9807, 'gamma'), (1342.2771, 'beta'), (1344.871, 'alpha'), (1346.3511, 'alpha'), (1351.2736, 'beta'), (1352.1132, 'beta'), (1355.4512, 'gamma'), (1358.2692, 'alpha'), (1376.996, 'gamma'), (1378.7351, 'beta'), (1385.5261, 'gamma'), (1388.5593, 'gamma'), (1388.727, 'beta'), (1397.1155, 'alpha'), (1404.7112, 'beta'), (1404.8633, 'gamma'), (1404.8818, 'gamma'), (1448.8042, 'gamma'), (1449.3851, 'gamma')]
3. The predicted class for this test example is beta with the least ssd being 272.33878
   And the sorted order of the ssds being [(272.33878, 'beta'), (272.67413, 'beta'), (281.6898, 'beta'), (313.5133, 'beta'), (316.18677, 'beta'), (330.94016, 'beta'), (345.341, 'beta'), (346.43585, 'beta'), (348.09937, 'alpha'), (350.1725, 'beta'), (357.75873, 'alpha'), (367.60162, 'alpha'), (369.27765, 'alpha'), (370.76117, 'alpha'), (375.26306, 'alpha'), (375.7703, 'alpha'), (385.52258, 'alpha'), (396.57697, 'gamma'), (405.57764, 'gamma'), (413.77908, 'gamma'), (436.289, 'alpha'), (437.33548, 'gamma'), (442.81848, 'gamma'), (445.11383, 'gamma'), (461.91, 'gamma'), (462.3432, 'gamma'), (473.48553, 'gamma')]
4. The predicted class for this test example is beta with the least ssd being 1032.418
   And the sorted order of the ssds being [(1032.418, 'beta'), (1059.4359, 'gamma'), (1062.1442, 'beta'), (1065.81, 'beta'), (1066.6863, 'beta'), (1069.636, 'beta'), (1074.1743, 'gamma'), (1074.847, 'beta'), (1076.4106, 'beta'), (1088.9434, 'alpha'), (1091.1721, 'alpha'), (1095.7346, 'beta'), (1099.2477, 'beta'), (1101.9634, 'alpha'), (1112.0942, 'gamma'), (1116.1715, 'alpha'), (1116.1895, 'gamma'), (1121.4702, 'gamma'), (1124.0642, 'gamma'), (1126.6866, 'gamma'), (1133.6013, 'alpha'), (1136.6729, 'gamma'), (1136.7668, 'alpha'), (1140.7612, 'alpha'), (1159.3481, 'alpha'), (1163.9783, 'gamma'), (1202.3268, 'alpha')]
5. The predicted class for this test example is gamma with the least ssd being 220.42996
   And the sorted order of the ssds being [(220.42996, 'gamma'), (225.51855, 'gamma'), (243.57407, 'alpha'), (249.32504, 'alpha'), (249.53116, 'gamma'), (258.696, 'beta'), (261.3225, 'beta'), (265.60406, 'gamma'), (266.0514, 'alpha'), (271.86835, 'beta'), (272.53302, 'gamma'), (275.41678, 'gamma'), (276.30127, 'alpha'), (281.3958, 'alpha'), (284.76147, 'beta'), (286.06674, 'beta'), (288.13477, 'alpha'), (289.25446, 'alpha'), (304.8319, 'gamma'), (306.31436, 'beta'), (308.18982, 'beta'), (308.5205, 'beta'), (308.7435, 'gamma'), (308.80328, 'alpha'), (313.65323, 'alpha'), (318.8186, 'gamma'), (339.6079, 'beta')]
6. The predicted class for this test example is gamma with the least ssd being 105.69834
   And the sorted order of the ssds being [(105.69834, 'gamma'), (119.36495, 'gamma'), (122.23416, 'gamma'), (137.2319, 'gamma'), (139.03111, 'gamma'), (139.29636, 'beta'), (141.3153, 'beta'), (149.72977, 'gamma'), (156.49487, 'beta'), (159.18347, 'beta'), (171.75308, 'alpha'), (172.75218, 'alpha'), (174.93536, 'alpha'), (182.9768, 'gamma'), (183.9507, 'gamma'), (185.98401, 'beta'), (186.14864, 'beta'), (191.73547, 'beta'), (192.89806, 'alpha'), (201.1836, 'beta'), (202.1629, 'alpha'), (209.10052, 'alpha'), (209.74527, 'alpha'), (233.25987, 'alpha'), (238.52965, 'gamma'), (247.6221, 'beta'), (301.98044, 'alpha')]

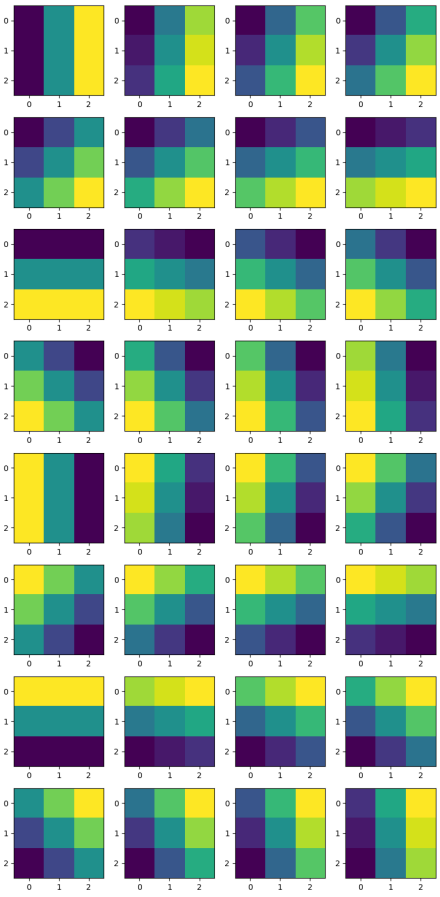
- D – a knn classifier should work pretty well for this task. As we can see from the ssds between the embedding of the test images (from part E) and the 27 embeddings the best match is not always a very small value (like see examples 1 and 2), so it's might also be the case that the best might is not the correct match, but it'd still be very likely that the second best, third best and so on might be the correct matches, or at least majority of the top few matches would still be correct. So, overall our guess is that using knn would just make our system more robust. This robustness would be especially nice to have if we increase the number of letters in our database.

Task 4

We've done this task in a Jupyter Notebook, which includes the report as well as the code for this task. Please go to the following link
https://colab.research.google.com/drive/1BV2w2xwR96jUTWYWftYOJ3dYMbXQZUwG?usp=sharing

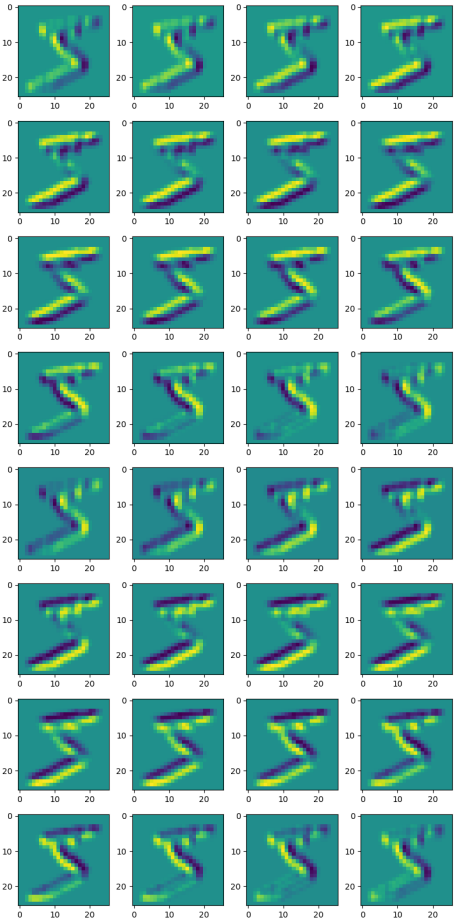
Extensions

1. A generalized way to fix the first layer of the model (i.e. make it untrainable) and initialize it with any kind of filter. We build this mainly to initialize the first layer with a set of Gabor filters, but just to show the capability of our system to work with any type of filters we also implement Gaussian filters. This system along with being capable of being trained and tested, also has the ability to perform all the sub-tasks of task, i.e. we can easily analyze the layers of the system or the outputs for the layers, etc.
    - The plot of the filters

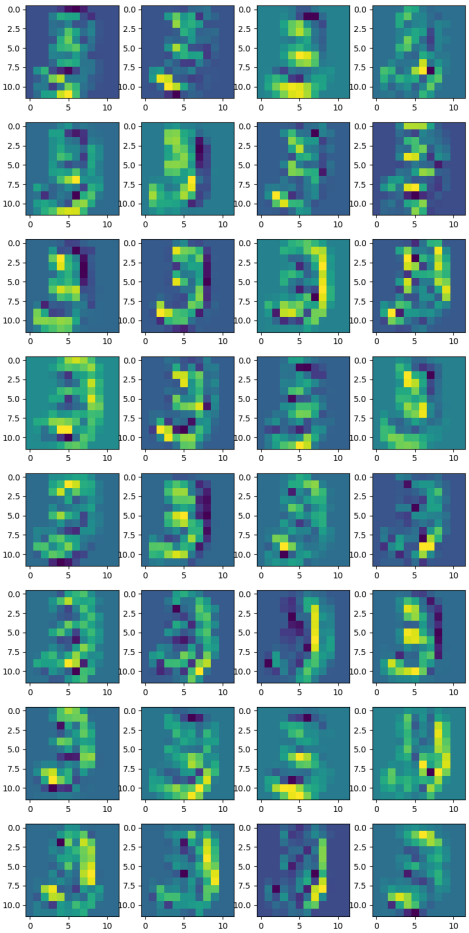


    - While making the Gabor filters we only vary the orientation of the filter but keep the rest of the parameters fixed. These can easily be changed in the code though. We make the filters by dividing a whole 360 degree angle into c number of parts (where c in the number of filters required).

- The output of the first layer of Gabor filters. And we can see that the output shows that each filter detects a certain kind of change in the pixel intensity or a certain orientation of edge.



- The output of the pooling layer for the 10th example (just to showcase that we can plot the output of any layer for any example we wish)



- Values of the filters (only showing the first few, the complete list can be found in the out.txt file of the extension)
  tf.Tensor(
  [[-5.6526756e-01 6.0755827e-17 5.6526756e-01]
  [-5.6970102e-01 6.1232343e-17 5.6970102e-01]
  [-5.6526756e-01 6.0755827e-17 5.6526756e-01]], shape=(3, 3), dtype=float32)
  tf.Tensor(
  [[-6.4183843e-01 -1.2121242e-01 4.5981041e-01]
  [-5.6069291e-01 6.1232343e-17 5.6069291e-01]
  [-4.5981041e-01 1.2121242e-01 6.4183843e-01]], shape=(3, 3), dtype=float32)
  tf.Tensor(
  [[-6.9218397e-01 -2.3547406e-01 3.2611090e-01]
  [-5.3338838e-01 6.1232343e-17 5.3338838e-01]
  [-3.2611090e-01 2.3547406e-01 6.9218397e-01]], shape=(3, 3), dtype=float32)
  tf.Tensor(
  [[-7.2021389e-01 -3.3692095e-01 1.6950892e-01]
  [-4.8714474e-01 6.1232343e-17 4.8714474e-01]
  [-1.6950892e-01 3.3692095e-01 7.2021389e-01]], shape=(3, 3), dtype=float32)
  …
- The accuracy of this model on the custom hand written digits is 100%. The output of the program:

```
The predicted digit is 0 with the class probabilities [0.99, 0.0, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 0
The predicted digit is 1 with the class probabilities [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 1
The predicted digit is 2 with the class probabilities [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 2
The predicted digit is 3 with the class probabilities [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 3
The predicted digit is 4 with the class probabilities [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 4
The predicted digit is 5 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0] and the correct digit 5
The predicted digit is 6 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.01, 0.98, 0.0, 0.02, 0.0] and the correct digit 6
The predicted digit is 7 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0] and the correct digit 7
The predicted digit is 8 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0] and the correct digit 8
The predicted digit is 9 with the class probabilities [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.99] and the correct digit 9
```

- The same outputs for the model with gaussian filters can be seen by running the FirstLayerFixed.py file, but ensure that the folder contains gaussian_model.h5 or that the load_model parameter is set to False.

2. Realtime digit recognition system:
   Using learned and saved model, we built a real time digit recognition system. Though not very precise prediction for noisy data, model is able to recognize digits correctly from captured frames in realtime.

   Demo link: https://youtu.be/gk-jt9IZQL8

3. The code for loading the images for the greek alphabets and creating a database out of it is generic enough that we can just add images for new alphabets and run the code, and it would just work without making any changes to the code. The only thing we need to ensure is that we're putting in the alphabets in the greek folder inside task3, and that the files follow the same naming convention as before which is the class name followed by an underscore followed by a number.

Reflection, Learnings

We learned to build a convolutional neural network and classify digits and fashion images. We used fashion dataset for performing experiments. Most important learning is to visualize the activations and filters of hidden layer, which really provides insights into how the network learns per epoch. Another learning was to use embedding space built by training the model using available dataset and use it for building the model for classification of other domain for which enough training samples are not available.

References

1. StackOverflow
2. Python documentation

3. Lectures and notes
4. Keras Tutorials
5. OpenCV tutorials
6. We also mention the specific links in the code if that part of the code was heavily inspired off that source.