EE6123 Deep Learning

Fall 2018

Assignment 1

Report

Author: Varun Sundar

EE16B068

16th September 2018

1 Implementation and Technical Notes

Python 3.6 was used to code, with modularity of components being the main focus. Questions may be run as:

```
python3 main --question q
```

with questions numbered from 1 to 7. The tarball also contains a dockerfile, which maybe used to replicate results.

1.1 Modularity of Code

Following modules are used:

- *Activations*
- *Confusion Matrix*
- *MNIST Downloader*

1.2 Directories

The following directories are meant for specific purposes:

- *data*: stores MNIST zip files
- *logs*: stores plots and outputs.
- *models*: stores .npz files, which are weights of the trained model (if saved).

2 Question 1 and 2

Implemented a feedforward network (Multilayer Perceptron) in numpy.

Used the MNIST Dataset with the following folds:

- Training Dataset, original 60k images
- k-Fold Dataset of 2k each, total 5 folds.
- Epochs ranging from 6 to 8, each with a minibatch of 64.
- We use this since it becomes more intuitive to interpret observations.

We shuffle both the train and fold datasets, and the first four folds are used as validation sets.

Network initialisation is done with random normal distributions of small variations ($\sigma = 0.08$ or lesser).

We obtain the following training loss and accuracy curves.

Notice that no regularisation was used, accounting for the overfitting nature of the MLP.

For part 2, we run metric evaluations on all 5 folds of the test-validation set.

We report the following: (for each fold)

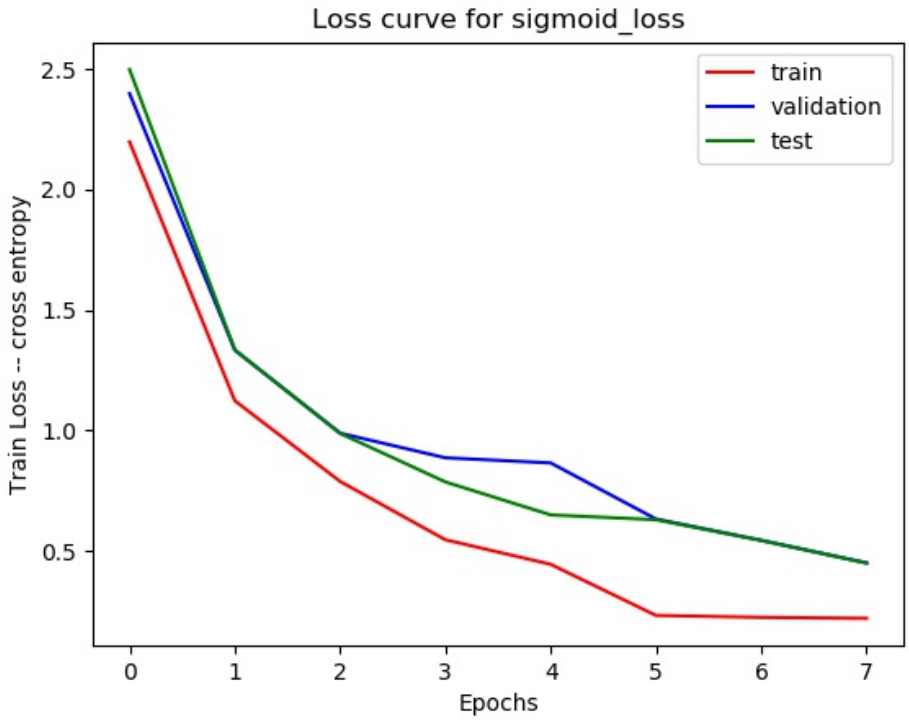

Figure 1: Loss curves for **MLP with sigmoid activations**

- Confusion Matrix
- Accuracy
- F1 Score (per class)
- Precision (per class)
- Recall (per class)

For brevity, metrics are not shown for all folds. They may be recovered by running

```
2     python3 main --question 1
```

```
cm [[ 773.    0.    7.    1.    0.    4.    9.    1.    7.    1.]
 [   0.  845.    1.   23.    0.    0.    3.    0.   48.    4.]
 [   8.    0.  728.   27.    8.    1.    9.    1.   37.    6.]
 [   0.    0.    2.  728.    1.   22.    0.    2.   27.   10.]
 [   3.    0.    5.    6.  667.    3.   12.    0.   30.   48.]
 [  11.    0.    5.   30.    2.  630.    7.    1.   30.   12.]
 [   9.    1.    2.    0.    2.   10.  724.    0.   10.    2.]
 [   5.    0.   96.   10.    6.    2.    0.  629.    8.   62.]
 [   2.    0.    3.   12.    5.    8.    5.    3.  714.   13.]
 [   6.    2.    5.   15.   20.   15.    0.    3.   29.  716.]]
 accuracy 0.9225
```

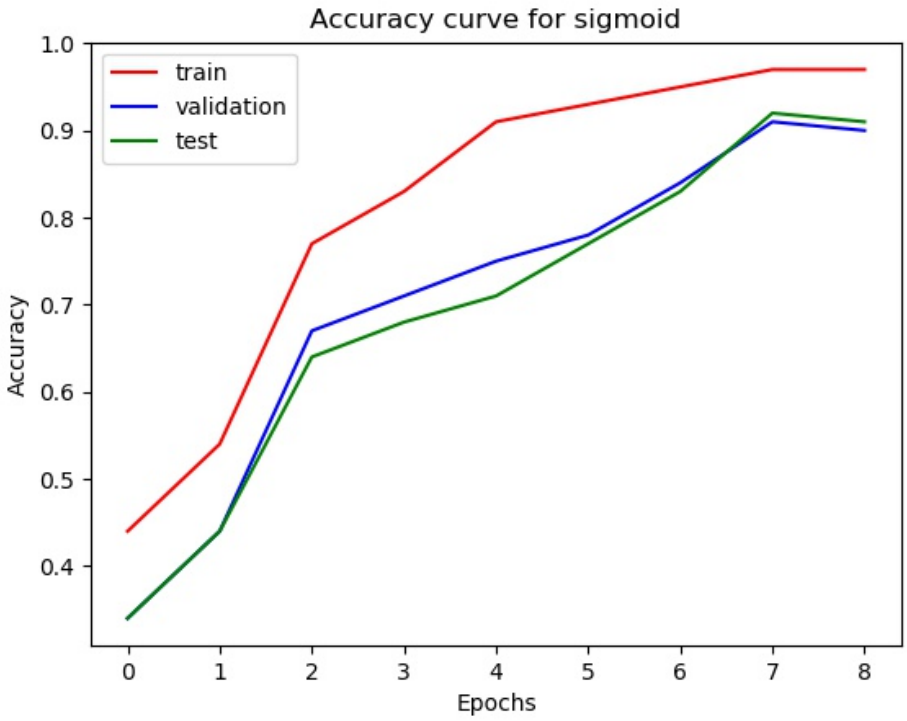

Figure 2: Loss curves for MLP with sigmoid activations

```
precision [ 0.94614443  0.99646226  0.85245902  0.85446009  0.93811533
    0.90647482
 0.94148244  0.9828125   0.75957447  0.81922197]
recall [ 0.9626401   0.91450216  0.88242424  0.91919192  0.86175711
   0.86538462
 0.95263158  0.76894866  0.93333333  0.88286067]
F1_score [ 0.95432099  0.9537246   0.86718285  0.88564477  0.8983165
   0.88545327
 0.9470242   0.86282579  0.83753666  0.84985163]
```

2.1 Code Blocks (Forward and Backward Pass)

```
20 def _forward_prop(self, x):
21         '''
22         RUn forward prop.
23         '''
24         a = np.array(x).reshape((len(x),1))
25         for count, b, w in zip(range(self.num_layers-1),self.biases,
               self.weights):
26             if count==self.num_layers-2:
27                 a = activations.softmax(np.dot(w, a)+b)
28             else:
29                 a = self.activation(np.dot(w, a)+b)
```

```
30          return a
31
32      def _back_prop(self, x, y):
33          """
34          Compute gradients of Cost
35
36          Returns:
37          * (nabla_b, nabla_w) representing the
38          gradient for the cost function C_x.
39
40          nabla_b and nabla_w are similar
41          to self.biases and self.weights.
42          """
43          nabla_b = [np.zeros(b.shape) for b in self.biases]
44          nabla_w = [np.zeros(w.shape) for w in self.weights]
45
46          # backward pass
47          delta = self.cost_derivative(a_ss[-1], y) * activations.softmax_
                prime(zs[-1])
48
49          nabla_b[-1] = delta
50          nabla_w[-1] = np.dot(delta, a_ss[-2].transpose())
51
52          for l in range(2, self.num_layers):
53              delta = np.dot(self.weights[-l+1].transpose(), delta) * self
                    .activation_prime(zs[-l])
54              #print(delta)
55              nabla_b[-l] = delta
56              nabla_w[-l] = np.dot(delta, a_ss[-l-1].transpose())
57
58          return (nabla_b, nabla_w)
```

We run the list of properties for all possible GPU's present in the given machine.

2.2 Running a Test Image

This is originally a part of question 5, but has been integrated into questions(1-2) , question(3), and question(4).

The full outputs may be recovered by running

```
59    python3 main --question 1
```

For brevity, a sample output is shown here:

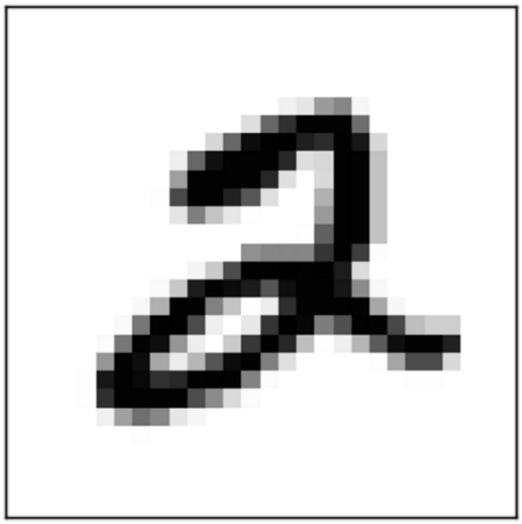

Figure 3: Sample MNIST digit. Y true =2 , Y pred =2 , probability (softmax) =0.91

3 Question 3

We use relu activation layers instead of sigmoid for training the MLP.

Question 3 involves using smaller variances to ensure no exploding gradients while using relu. We however observe faster and better convergence with relu. This could be attributed to the nearly linear nature of the activation and the fact that it does not die for large values.

The relevant metrics:

```
test loss 0.49272746608690554
cm [[ 170.    0.    3.    1.    0.    0.    3.    0.    0.    0.]
 [   0.  197.    0.    5.    0.    0.    1.    0.    7.    1.]
 [   0.    0.  184.    7.    1.    0.    3.    0.    9.    3.]
 [   0.    0.    2.  199.    0.    8.    0.    0.    4.    5.]
 [   0.    0.    1.    2.  180.    1.    3.    0.   10.   11.]
 [   0.    0.    0.    7.    0.  145.    2.    0.    9.    1.]
 [   3.    1.    1.    0.    2.    2.  185.    0.    3.    1.]
 [   0.    0.   18.    4.    1.    4.    0.  161.    4.   18.]
 [   0.    0.    2.    3.    5.    1.    0.    1.  196.    1.]
 [   0.    0.    1.    6.    9.    4.    2.    0.    6.  170.]]
```

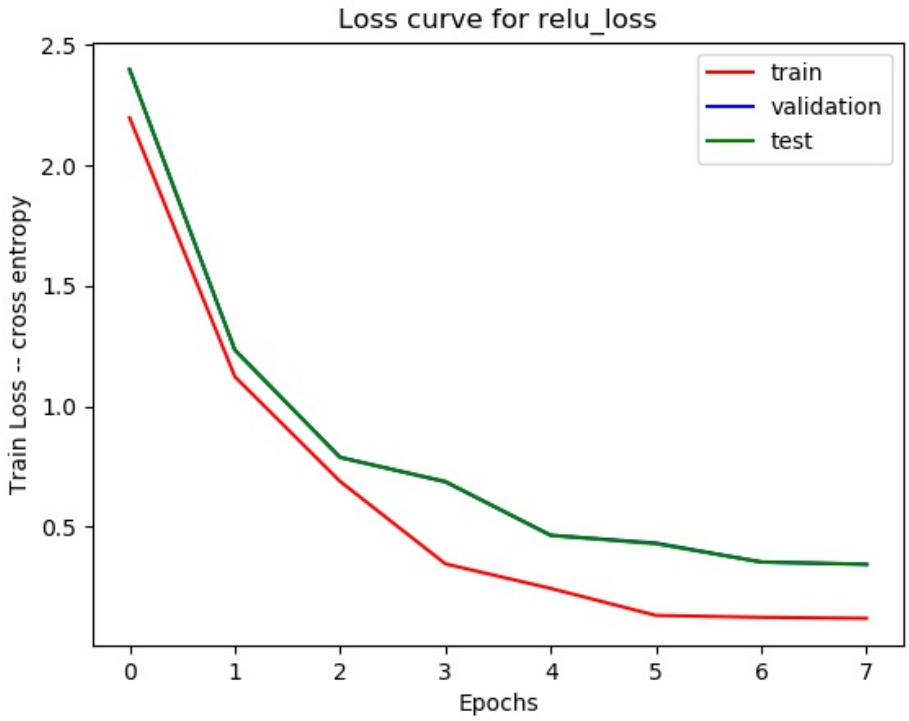

Figure 4: ReLu Loss curves

```
accuracy 0.9635
precision [ 0.98265896  0.99494949  0.86792453  0.85042735  0.90909091
    0.87878788
 0.92964824  0.99382716  0.89032258  0.8056872 ]
recall [ 0.96045198  0.93364929  0.88888889  0.91284404  0.86538462
   0.88414634
 0.93434343  0.76666667  0.93779904  0.85858586]
F1_score [ 0.97142857  0.96332518  0.87828162  0.88053097  0.88669951
   0.88145897
 0.93198992  0.8655914   0.85776805  0.83129584]
```

4 Question 4

Here, we use regularisation to decrease the variance of the MLP model. We use both L2 and L1 regularisation, as well as data augmentation via noise pertubation.

The regularisation lets us achieve higher test accuracies and lowers the gap between the test and train curves.

The relevant training plots are shown below:

The relevant metrics: (on 1 fold)

```
Test Stats:
 test loss 0.6416773891908012
cm [[ 172.    0.    0.    2.    0.    2.    1.    0.    0.    0.]
```

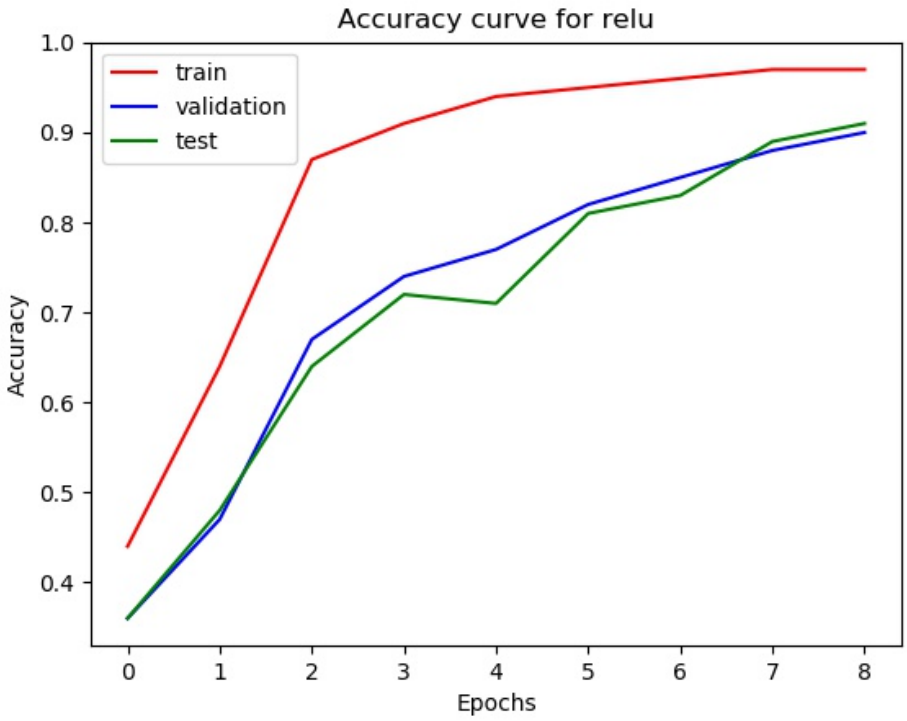

Figure 5: ReLu Accuracy curves

```
[   0.  208.    0.    2.    0.    0.    0.    0.    1.    0.]
[   1.    0.  192.    2.    3.    0.    1.    0.    6.    2.]
[   1.    0.    5.  202.    0.    1.    0.    5.    1.    3.]
[   6.    1.    2.    0.  184.    0.    0.    1.    4.   10.]
[   2.    0.    0.   16.    0.  140.    1.    0.    4.    1.]
[   3.    2.    1.    8.    2.    3.  178.    0.    1.    0.]
[   0.    2.    5.    2.    1.    1.    0.  196.    0.    3.]
[   4.    0.    1.   10.    3.    6.    0.    4.  179.    2.]
[   2.    1.    0.    2.    9.    2.    2.    0    1.  179.]]
accuracy 0.9115
precision [ 0.90052356  0.97196262  0.93203883  0.82113821  0.91089109
    0.90322581
 0.9726776   0.92018779  0.90862944  0.89119171]
recall [ 0.97175141  0.98578199  0.92753623  0.9266055   0.88461538
   0.85365854
 0.8989899   0.93333333  0.85645933  0.86868687]
F1_score [ 0.93478261  0.97882353  0.92978208  0.87068966  0.89756098
   0.87774295
 0.9343832   0.92671395  0.8817734   0.8797954 ]
```

4.1 Code Changes

```
97          self.v = [
```

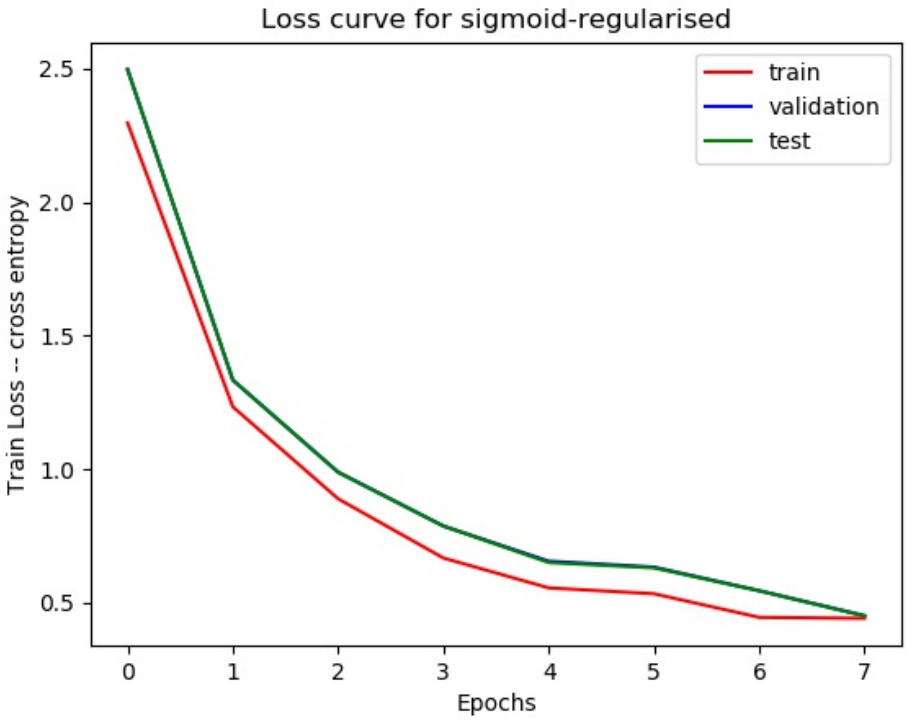

Figure 6: Sigmoid Regularised curves

```
98                      v*self.mu
99                      - self.eta/self.mini_batch_size * dw \
100                     - self.eta * self.l2/self.mini_batch_size *w \
101                     - self.eta * self.l1 * np.sign(w)/self.mini_batch_
                           size \
102                     for v,w, dw in zip(self.v,self.weights, nabla_w)]
103
104                 #print(self.v)
105                 self.weights = [
106                     w + v
107                     for w, v in zip(self.weights, self.v)]
108
109                 self.vb = [
110                     vb*self.mu \
111                     - (self.eta / self.mini_batch_size) * db \
112                     for vb, db in zip(self.vb, nabla_b)]
113
114                 self.biases = [
115                     b + vb \
116                     for b, vb in zip(self.biases, self.vb)]
```

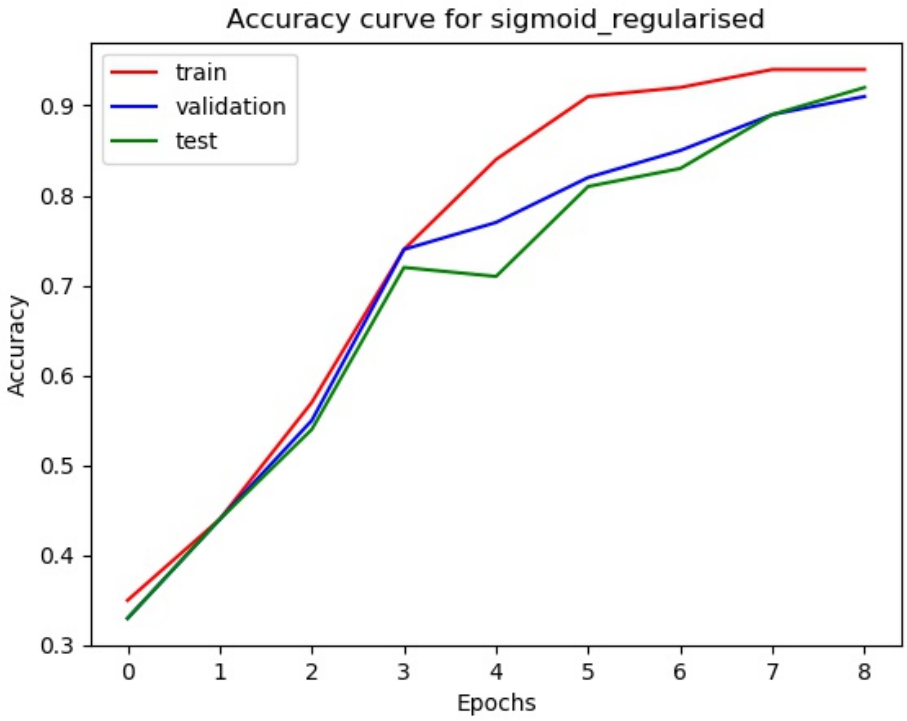

Figure 7: Sigmoid Regularised Loss curves

5 Question 6 and 7

Here, we use the **Histogram of Oriented Gradients** as the feature extractor before using a classifier. In **Question 6**, we try a MLP based classifier having one hidden layer of 32 units. The reasoning behing this is that a HOG feature descriptor itself is a simplified extractor which reduces 784 values to just 64.

Hence, a smaller network can be feasibly trained.

We report the following accuracy and loss curves.

Now, instead of using a MLP, we choose to use *scikit's* SVM classifier. We try both linear and radial basis function kernels (RBF). We find the RBF one to be more optimal, although it takes significantly longer to train.

We also report the confusion matrix and other parameters over one fold:

```
sigmoid-cross_val-fold-0 loss 1.2664457767108068
cm [[ 164.    2.    0.    9.    8.    4.    4.    1.    4.    4.]
 [   0.  229.    0.    0.    1.    0.    1.    0.    0.    1.]
 [  34.    0.    0.   74.    5.   33.    7.   25.   26.   18.]
 [   9.    0.    0.  124.    0.   27.    2.    1.   28.    3.]
 [   9.    3.    0.    1.  179.    0.   18.    9.    3.    7.]
 [   4.    0.    0.   17.    0.  129.    0.    1.   13.    1.]
 [   6.    0.    0.    4.    7.    3.  154.    0.    6.    0.]
 [   1.    2.    0.    3.    0.    0.    0.  173.    2.   17.]
 [  35.    1.    0.   25.    2.    8.   12.    7.   91.    4.]
```

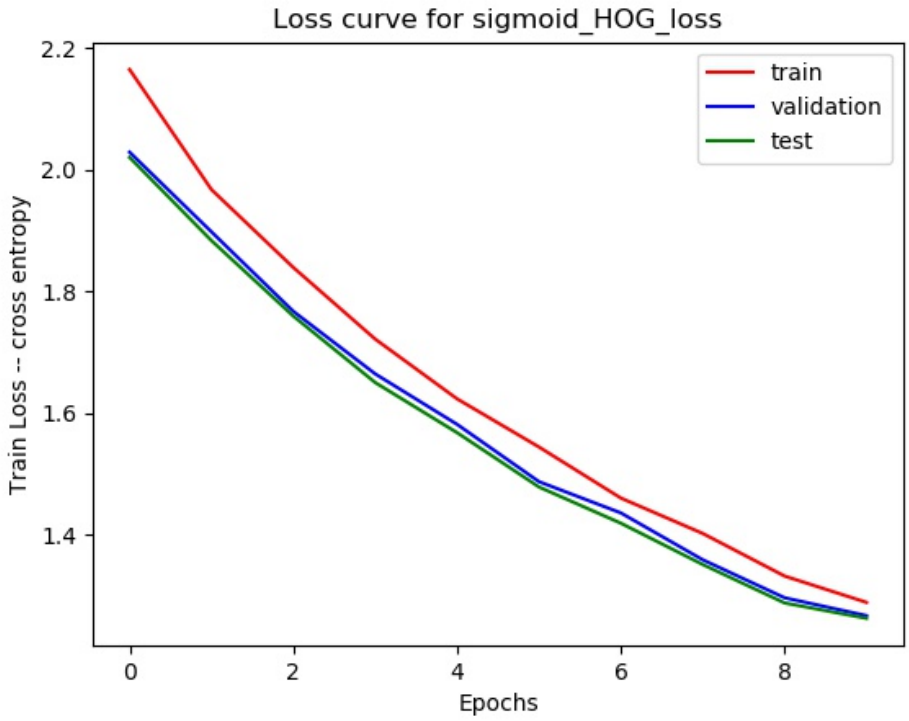

Figure 8: Sigmoid HOG Loss curves

```
 [  12.    0.    1.    4.    5.    0.    1.   50.    4.  118.]]
 accuracy 0.6805
 precision [ 0.59854015  0.96624473  0.          0.47509579  0.8647343
      0.63235294
  0.77386935  0.64794007  0.51412429  0.68208092]
 recall [ 0.82        0.98706897  0.          0.63917526  0.78165939
    0.78181818
  0.85555556  0.87373737  0.49189189  0.60512821]
 F1_score [ 0.69198312  0.97654584         nan  0.54505495  0.82110092
    0.69918699
  0.81266491  0.74408602  0.50276243  0.64130435]

sigmoid-cross_val-fold-1 Stats:
 sigmoid-cross_val-fold-1 loss 1.2676043570317577
cm [[ 151.    0.    0.    7.    4.    2.    3.    4.    2.    2.]
 [   0.  213.    0.    0.    4.    0.    0.    0.    2.    2.]
 [  36.    0.    0.   79.    2.   35.   11.   27.   15.    8.]
 [  13.    1.    0.  140.    1.   30.    2.    2.   19.    2.]
 [   4.    3.    0.    0.  163.    0.   12.    9.    2.    3.]
 [   2.    0.    0.   11.    0.  143.    2.    2.   15.    1.]
 [   9.    0.    0.    3.    5.    7.  176.    1.    1.    0.]
 [   2.    1.    0.    6.    2.    0.    0.  153.    2.   30.]
 [  41.    0.    0.   27.    2.    6.   17.   12.   94.    7.]
 [  14.    1.    0.    6.   10.    0.    0.   42.    6.  126.]]
```

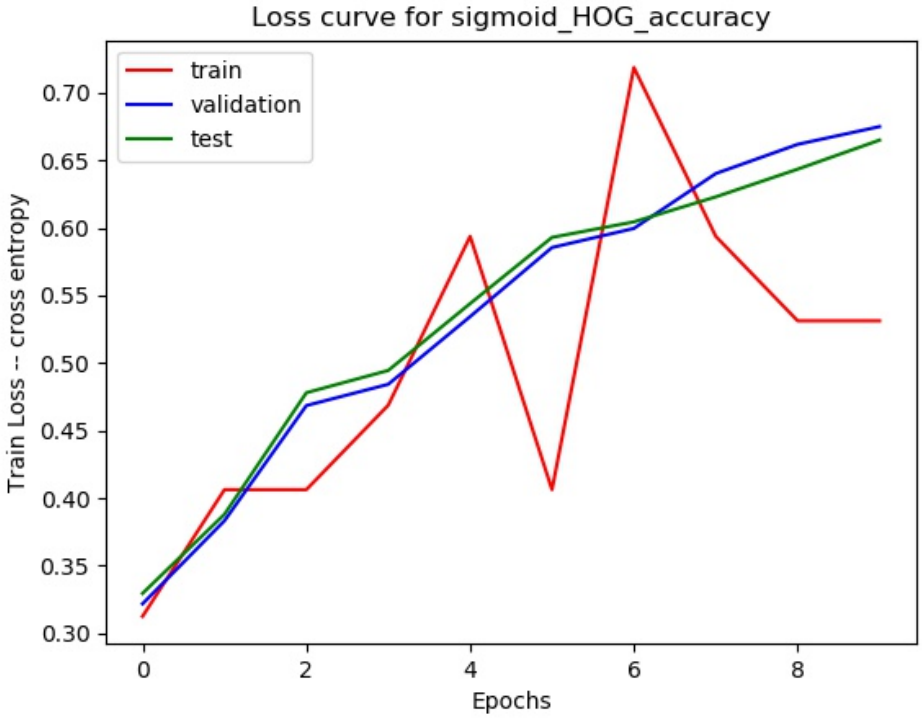

Figure 9: Sigmoid HOG Accuracy curves

```
accuracy 0.6795
precision [ 0.55514706  0.97260274         nan  0.50179211  0.84455959
    0.64125561
 0.78923767  0.60714286  0.59493671  0.6961326 ]
recall [ 0.86285714  0.9638009   0.          0.66666667  0.83163265
   0.8125
 0.87128713  0.78061224  0.45631068  0.61463415]
F1_score [ 0.67561521  0.96818182         nan  0.57259714  0.83804627
   0.71679198
 0.82823529  0.68303571  0.51648352  0.65284974]
```

We report the following accuracies for the SVM (RBF kernel):

- C=5, gamma=0.05
- accuracy = 0.9852

The confusion matrix and other metrics for the SVM case are not shown here for brevity. This may be recovered by running:

```
155     python3 main --question 7
```

5.1 Interpretation

MNIST although the hello-world dataset of Machine Learning and Deep Learning, is significantly smaller than modern datasets such as Imagenet, OpenImage, PASCAL VOC etc. This is

in both in terms of dataset size and number of features of each image. A feed-forward neural network is not the most optimal for this task, rather using the smaller number of features available. Another way to state this is that deep feed-forward networks are better suited for higher dimensional problems.

When using a carefully tailored and hand-engineered feature extractors, running standard ML classifiers such as SVMs and random (or boosted) forests turns out to be much more optimal. This again reflects the fact that as the dimensions of the features drop, neural nets are outperformed by more direct classifiers.

For the sake of this problem set, we considered HOG feature extractors for the following reasons:

- Given MNIST has high contrast images, using gradients seem intuitive.
- Binning the features given that the original images are just $28 * 28$ could be more efficient than attempting to use SIFT or SURF or even ORB features.
- Another possibility (not done here) could be to use a Haar Cascade trained for MNIST. (similar gradient approach).