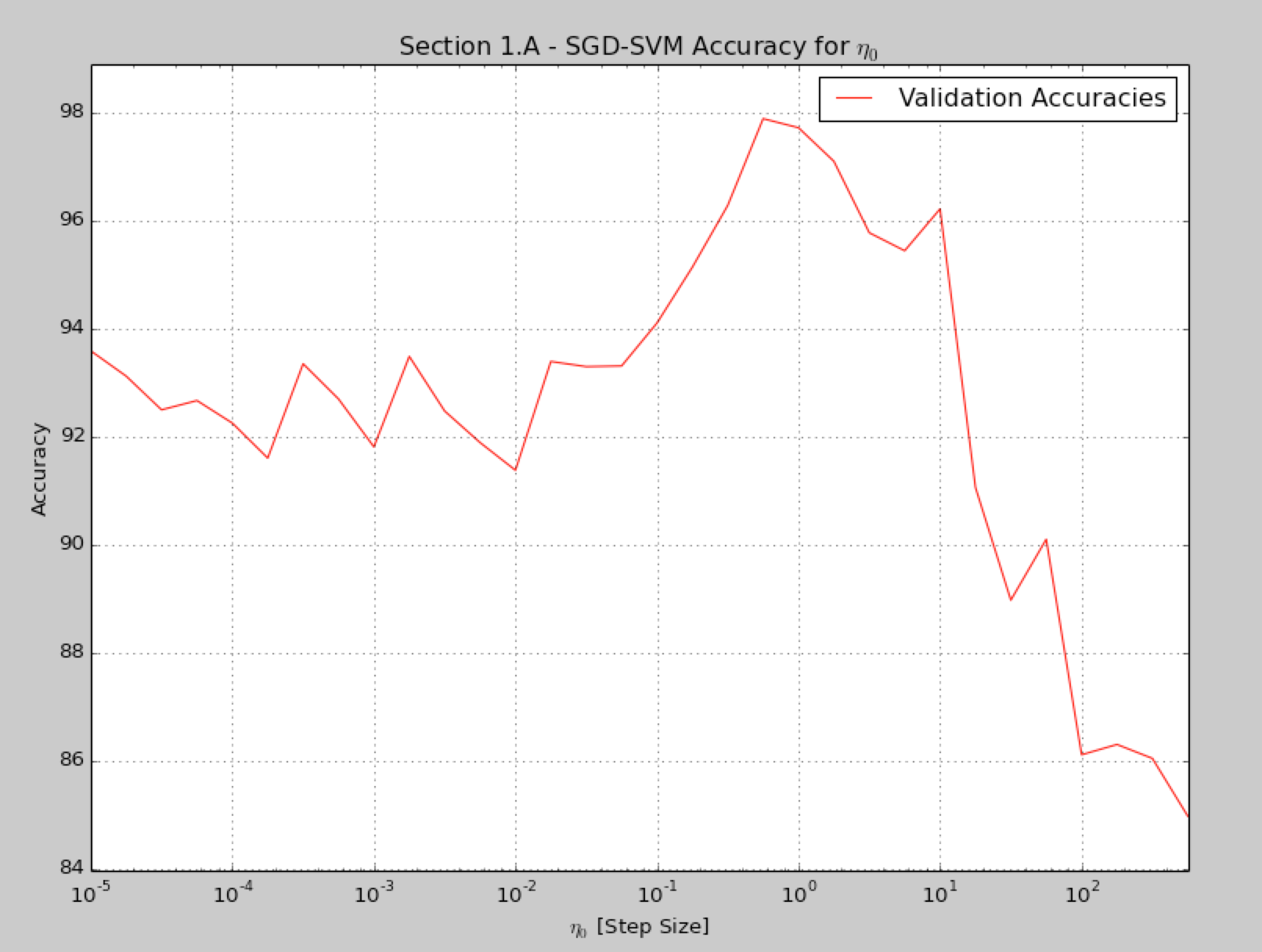
Assignment #4 – Programming Part

**Code Location:**

**Question 1**

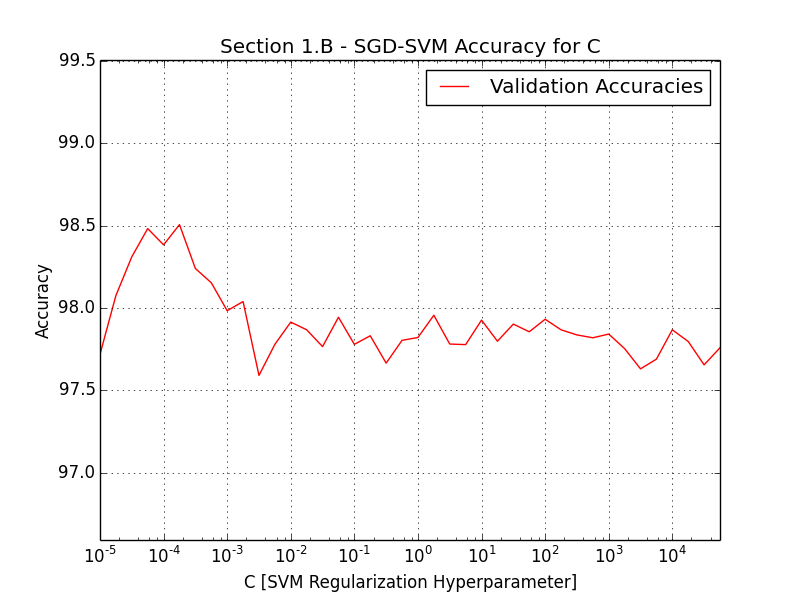
**a.** Searching for the step size that gives the highest validation accuracy over the range [10^-5, 10^3] with resolution of 0.25 yields the following graph:

****

The graph suggests that when the step size is too small for the amount of gradient steps SGD takes (T), the SVM’s weights don’t manage to update by the end of the training to fit the features MNIST model accurately. This phenomenon is mitigated as we approach the ideal step size (eta0=0.25, accuracy is almost 98%).

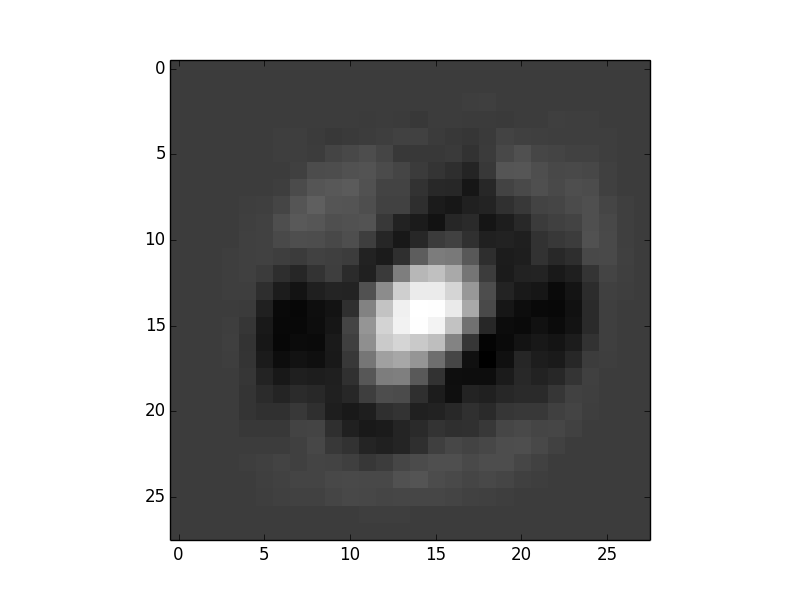
When the step size is too big, the SVM’s weights get big updates that may cause the gradient updates to skip the global minima of the loss function, yielding bad accuracy again (the algorithm fails to converge to the local minimum with such big step resolution). We’ve neglected results above 10^3 as they give extremely bad accuracies, and eventually cause numpy to overflow (even for float64 ndarrays).

**b.** Searching for the C regularization parameter that gives the highest validation accuracy over the range [10^-5, 10^5] with resolution of 0.25 yields the following graph:

****

The best C found is 5.7e-5, which yields an accuracy of almost 98.5% on the validation dataset (small value of C suggests the SVM is looking for a large-margin separating hyperplane).

**c.** With the optimal C, eta0 parameters found in Sections a, b, and T=20000, the classifier achieves an accuracy of 98.54% over the validation dataset.

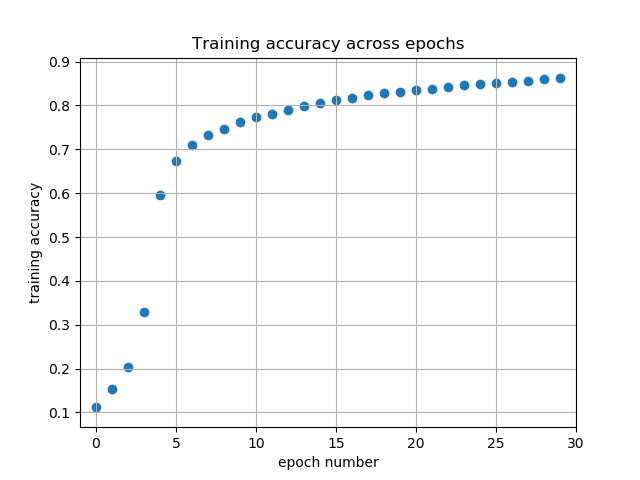
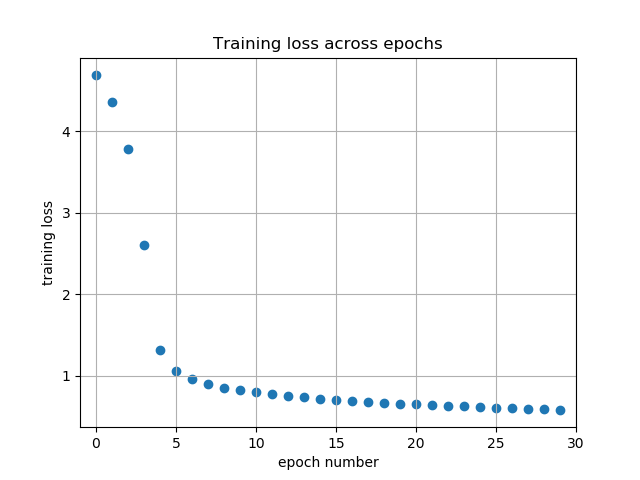
The SVM’s weights appear as follows: 

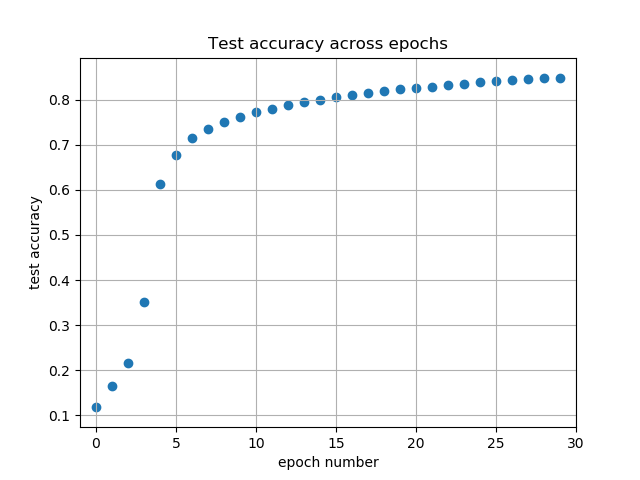
Compared to the Weight Matrix extracted in Ex3, this visualization looks smoother (lower frequencies), and the black / white areas that identify 0/8 digits appear to be more uniform.

**d.** The average accuracy for 10 iterations on the test set is: 99.20% (Best classifier, T=20,000).

**Question 2**

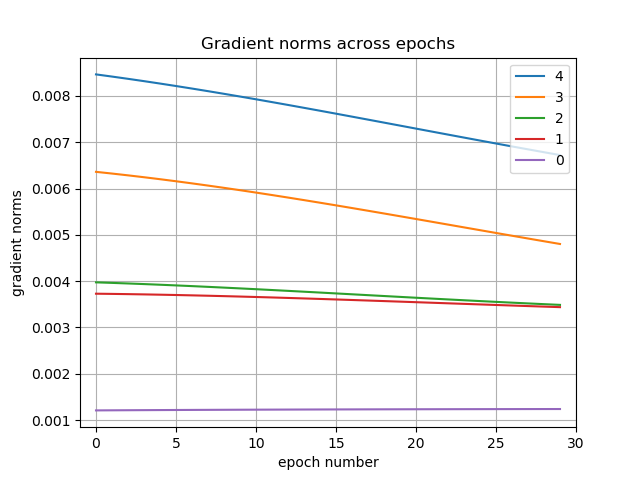
**b.** We got the following results:





As we can see from the graphs, as training progresses, the training and test accuracies increase and the training loss decreases.

**c.** In the last epoch we got a test accuracy of 91.59%

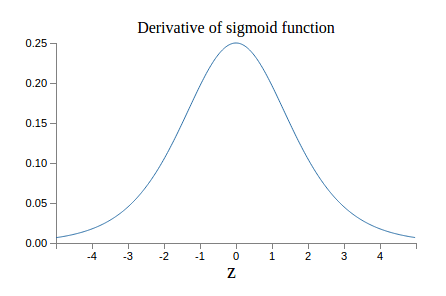
**d.** The plot for this section:

As we can see, gradients closer to the input (lower numbers) have lower norm.

The expression of the gradients is:

This implies that when we look at lower layer gradient, it will have more terms of (because unwrapping will yield more terms of ) .

The looks like this:



And the key feature here is that is always smaller than 0.25.

So if we take the product of many terms, because it is smaller than 1, the result will tend to exponentially decrease.

This can be a possible explanation for the vanishing gradient phenomenon.

**Question 3**

**a.** After 3000 steps, the test accuracy is 92.46%.

**b.** The 3-layered network achieves a test accuracy of 91.54%.

Compared with the Linear Classifier we now have tenfold more parameters (weights) to tune.

The network is now a bit deeper (albeit still quite shallow), so gradients have to backpropagate from the output of the network, all the way to the input. Since we initialize our weights randomly, some weights may contain values close to zero, which results in diminishing gradients (this phenomenon is actually attributed to deeper network more often than with shallow networks such as our 3-layered network).

Perhaps a more convincing suggestion would be the amount of steps taken to train the network: since we have more hyperparameters to tune, our loss function on the hidden layer is now very high dimensional. Since the loss function is not necessarily convex in this case, we have a high chance of getting stuck in a local minima, especially with a non-optimal learning rate (this is where smarter optimizers such as AdaGrad or Adam come into use).

**c.** Modifying the following lines (initializing W1, W2 with Xavier Initialization) yields a test accuracy of 0.975:

initializer = tf.contrib.layers.xavier\_initializer()

 W1 = tf.Variable(initializer(([784, n\_hidden\_size])))

 W2 = tf.Variable(initializer(([n\_hidden\_size, 10])))

**d.** Modifying the code to use ConvNets involves appending the x placeholder (input tensor) to the deepnn ConvNet. The y\_pred output of the ConvNet (following the last Fully Connected Layer) is followed by the original network’s SoftMax which gives probabilities of the image belonging to each of the 10 digit classes. The entire code is submitted as a modified version of tf\_mnist.py.

The test accuracy of the ConvNet classifier is 97.1% (700 steps, need to wait for 3000…).