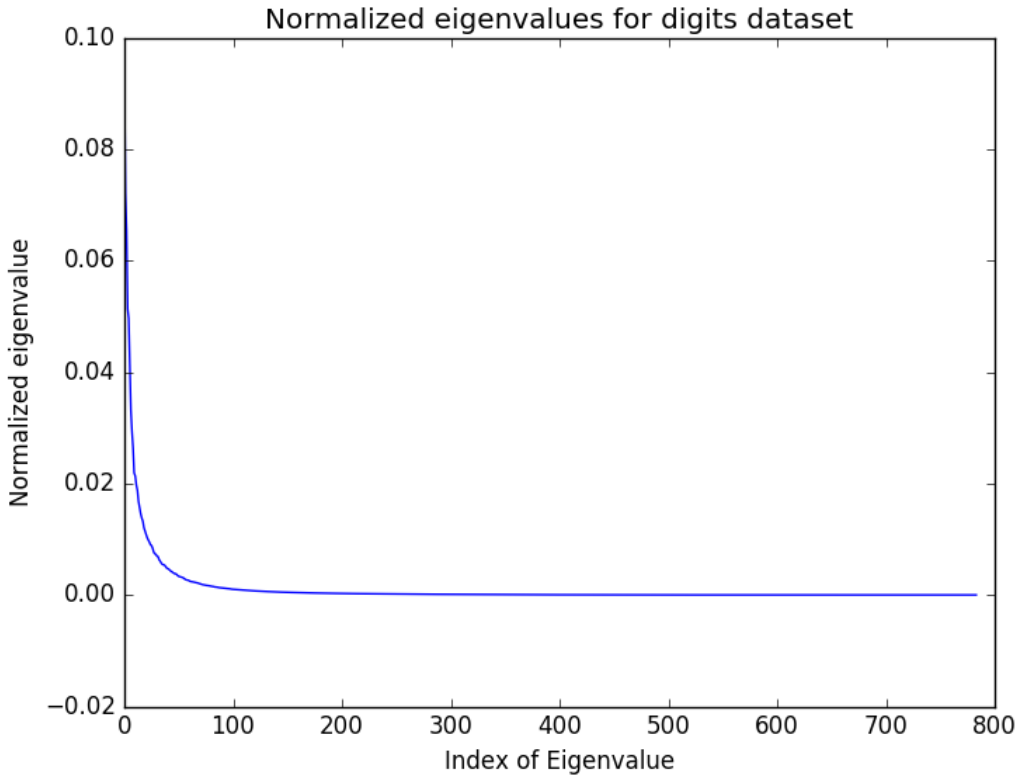
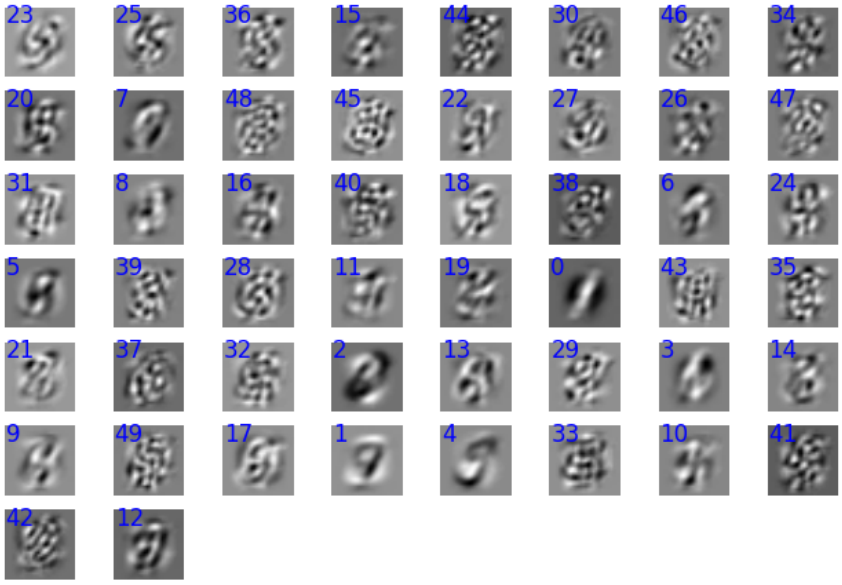
Qpca1 ***(15%)*** **Implement PCA**

Qpca2 (10%): Plot the normalized eigenvalues (include the plot in your writeup). How many eigenvectors do you have to include before you've accounted for 90% of the variance? 95%?



In order to determine the number of eigenvectors needed to account for a set % of variance, you need to find the total variance first, or the sum of the eigenvectors. Then, we use the cumsum function find the cumulative sum, and divide that value by the total variance. Finally, to find the number of eigenvectors that need to be included before you've accounted for 90% or 95% you just need to find the first time where the percent variance excceds .90 or .95. Using the plot above, we find that 82 eigenvectors are needed to account for 90% of the variance, and 136 eigenvectors are needed to account for 95% of the variance.

Qpca3 (5%): Do these look like digits? Should they? Why or why not? (Include the plot in your write-up.) (Make sure you have got rid of the imaginary part in pca.)



Some of the images plotted resemble digits. This result is expected because by plotting only the top 50 eigenvectors, we will get information that is a partial description of the data. Using assumptions from Qpca2, the plotted data here represents about 82% of the variance in the data. Thus, the image data is understandably unclear, since we are not showing all of the data.

Part II Softmax Regression

Qsr1

(1) Assume there are *K* classes, then the sum of the probabilities is:

$$\sum_{i=1}^{K} P[y=i] = \sum_{i=1}^{K} \frac{e^{\overrightarrow{w_i}\cdot\overrightarrow{x}}}{\sum_{j=1}^{K} e^{\overrightarrow{w_j}\cdot\overrightarrow{x}}} = \frac{\sum_{i=1}^{K} e^{\overrightarrow{w_i}\cdot\overrightarrow{x}}}{\sum_{j=1}^{K} e^{\overrightarrow{w_j}\cdot\overrightarrow{x}}} = 1$$

The last equality follows because both *i* and *j* range from 1 to K.

(2) Let *M* denote the number of examples, *D* denote the number of features of each example, and *K* denote the number of classes, then the dimensions of *W*, *X* and *WX* are $K \times D$, $D \times$ M and $K \times M$, respectively.

Qsr2

(1) Below is the code.

```
probabilities = np.exp(W_X)
probabilities = probabilities / probabilities.sum(axis=0)
cost = - np.average(np.log((indicator * probabilities).sum(axis=0)))
indicator = indicator - probabilities
gradient = - np.matmul(indicator, X.T) / len(Y)
```

(2) Below is the code.

```
predicted_classes = W_X.argmax(axis=0)
```

Note that at the prediction step there is no need to explicitly compute the probabilities. This is because within a column of W_X, the element of the largest numerical value will certainly be converted to the largest probability. Therefore, we can simply find the largest elements in each of the columns of W_X and return their row indices. This saves a lot of computation time since computing the probabilities are expansive.

If we have to implement the method the way instructed, then the code is:

```
probabilities = np.exp(W_X)
probabilities = probabilities / probabilities.sum(axis=0)
predicted_classes = probabilities.argmax(axis=0)
```

Qsr3

(1) Let $K \times M$ be the dimensions of W_X as discussed in Qsr1, and ’s be the elements of W_X, and b be the largest element in W_X.

The probability of the jth example being predicted the ith label is:

$$P[Y = i \,|\, x_j] = \frac{e^{a_{ij}}}{\sum_{k=1}^{K} e^{a_{kj}}}$$

If we subtract from every and do the computations in the same way, then the new probability is:

$$P'[Y = i \,|\, x_j] = \frac{e^{a_{ij}-b}}{\sum_{k=1}^{K} e^{a_{kj}-b}} = \frac{e^{b} \cdot e^{a_{ij}-b}}{e^{b} \cdot \sum_{k=1}^{K} e^{a_{kj}-b}} = \frac{e^{a_{ij}}}{\sum_{k=1}^{K} e^{a_{kj}}} = P[Y = i \,|\, x_j]$$

We have proved two probabilities are exactly the same.

(2) It’s possible that all of the entries in W_X are much larger or much smaller than 0.

If them all are much larger than 0, python will return infinity's when taking the exponentials of them. If they all are much smaller than o, python will return 0's. In both situations the computation of the probabilities will fail, because we will encounter 0/0 or inf/inf.

By subtracting the largest entry from every of the entries in W_X, we shift the data points closer to the origin point, thus reduces the probability of obtaining bad values (0's and infinity's) when taking exponentials. As the result, our program becomes more robust.

Because each element is only compared to the elements in the same column, there is a better way to do the optimization: instead of subtracting from every element the largest element in W_X, we can subtract from every element the largest element within the column of W_X.

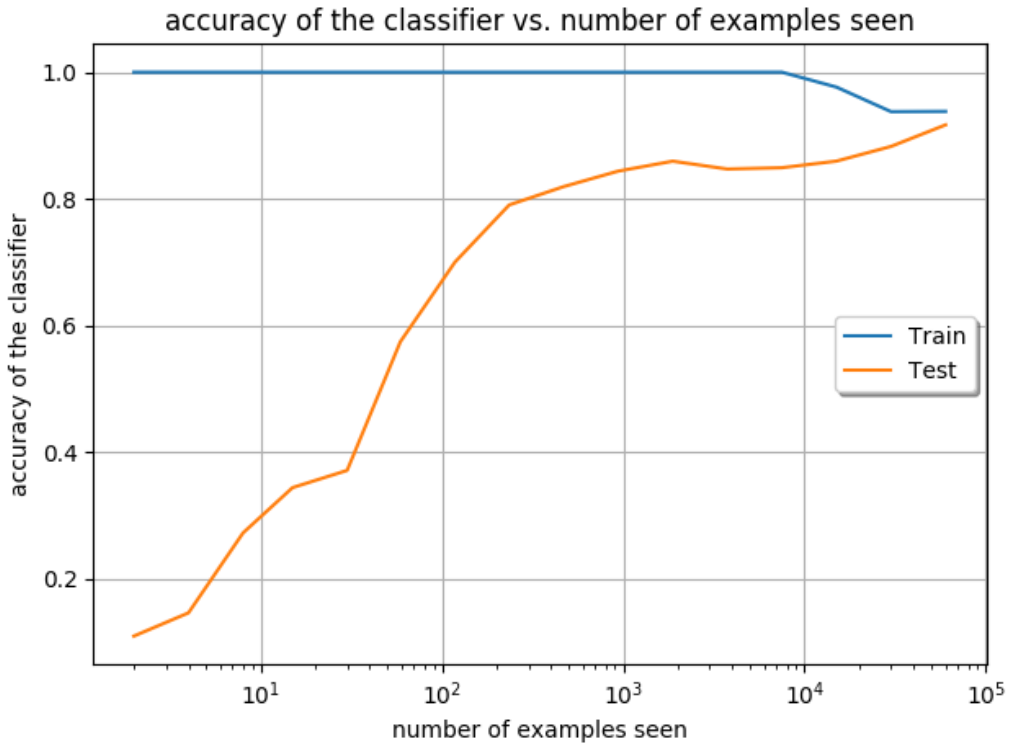
```
W_X = W_X - W_X.max(axis=0)
```

The probabilities of the predictions stay the same and the proof given in (1) applies here too.

Qsr4

There is no overfitting or underfitting at the end of the curves.

In the left half of the plot, we see that the training accuracy stays high while the test accuracy climbs from low to high rapidly as the classifier sees more and more training examples. This is what happens when the classifier is underfitting and it can learn more if given more training examples. At the end of the plot, the test accuracy has become high and is very close to the training accuracy. This is what happens when the classifier has learned just enough and generalize well to new data.



Qnn1

Qnn1.3 Run in epochs

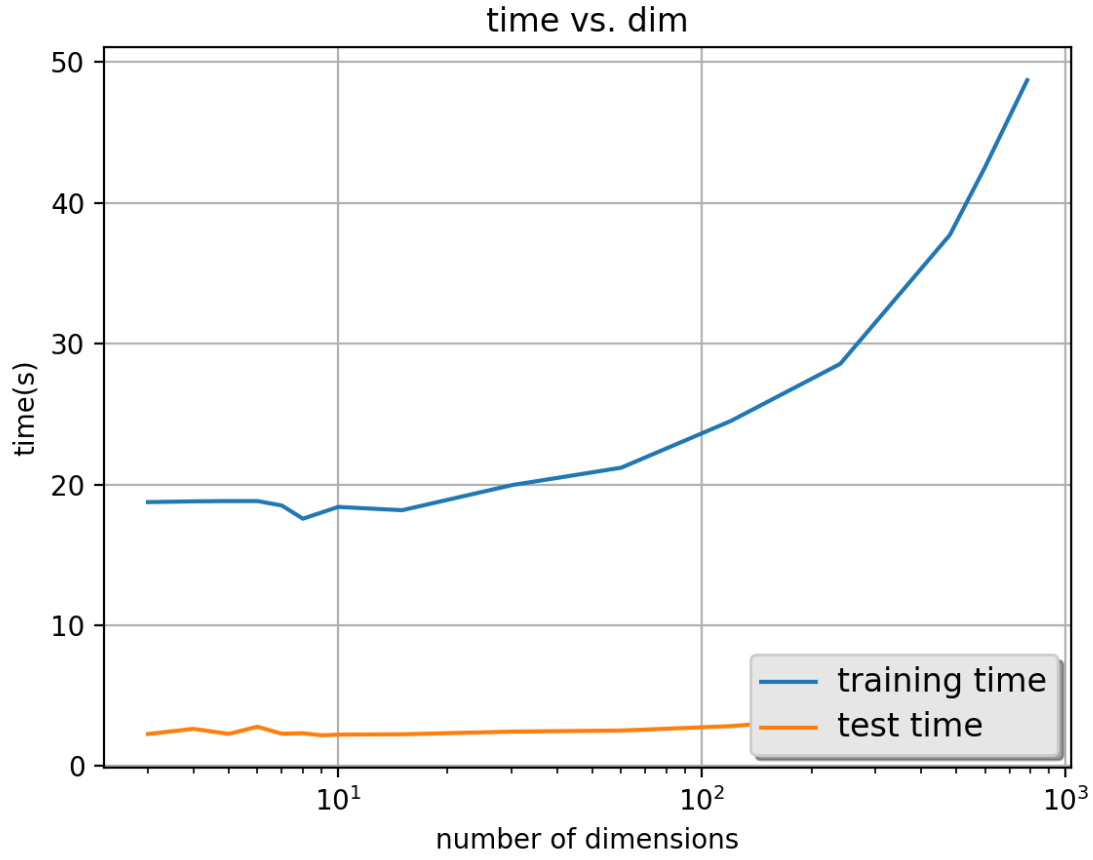
The accuracy of the dev test is 0.94630 in the end.

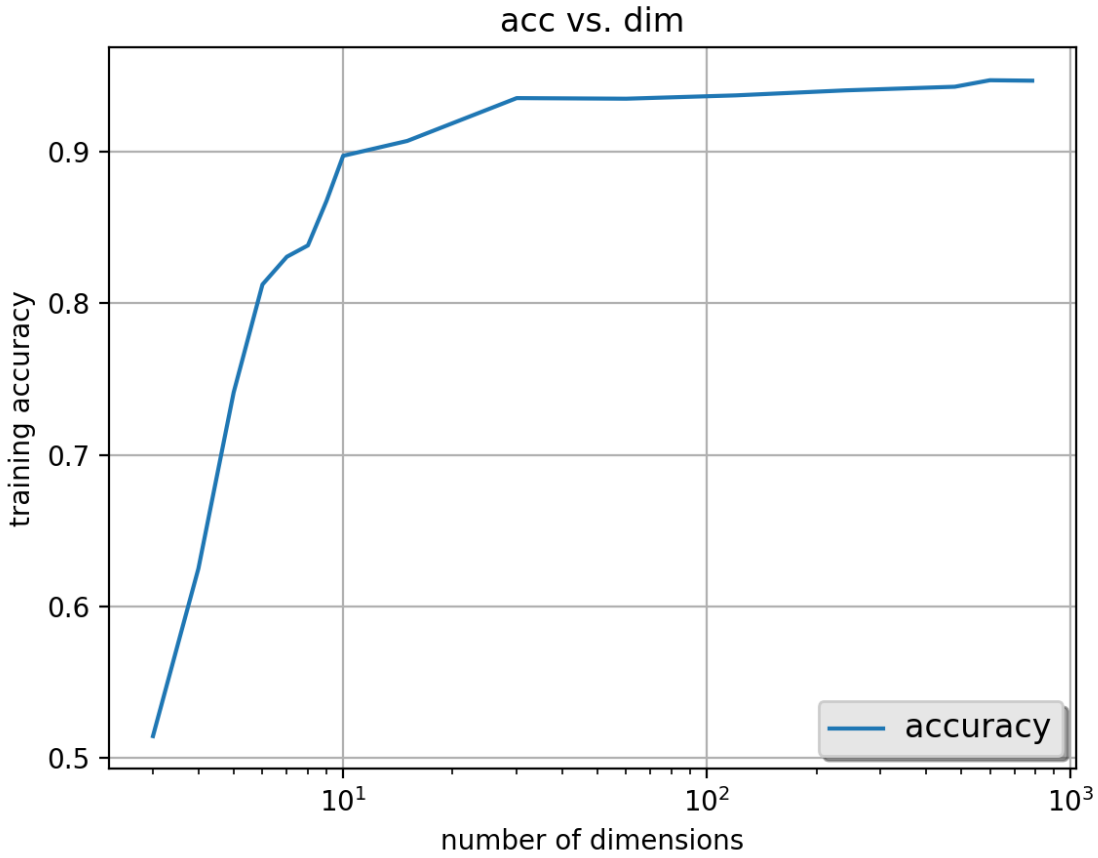
Qnn1.4 (No implementation needed for this question).

When initializing the weight, if all of the weights are zero, they will all have the same error and the model will not learn anything, however we need to break the symmetry. Randomization creates more entropy to the system to better find the global optimal solution to the problem.

Qnn2

(1) Do dimension reduction with PCA. Try with different dimensions. Can you observe the trade-off in time and acc? Plot training time v.s. dimension, testing time v.s dimension and acc v.s. dimension. Visualize the principal components.


time vs. dim
time(s)
number of dimensions
training time
test time
0
10
20
30
40
50
10^1
10^2
10^3


acc vs. dim
training accuracy
number of dimensions
accuracy
0.5
0.6
0.7
0.8
0.9
10^1
10^2
10^3

Qnn2.1 Explain what you did and what you found. Comment the code so that it is easy to follow. Support your results with plots and numbers. Provide the implementation so we can replicate your results.

I implemented the training process in Qnn2.py and modified run_nn.py as run_nn2.py. I set the dimensions for PCA to be in K=[3, 4, 5, 6, 7, 8, 9, 10, 15, 30, 60, 120, 240, 480, 600, 784]. With each dimension, I performed a NN training and get the training time, testing time and accuracy. I found that the training accuracy achieves 90% for dimensions greater than about 15. The training time consumption grows steadily (the x axis is log in the graph) as dimensions number grows. The training time remains the same.