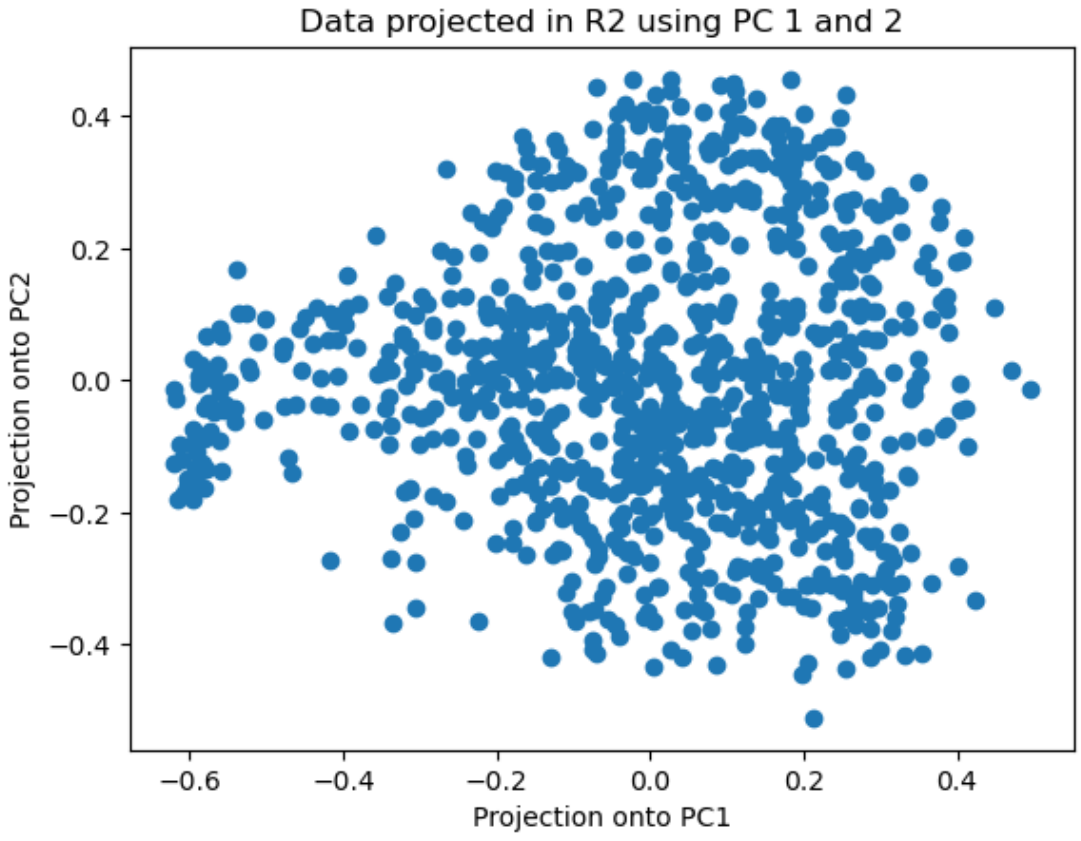
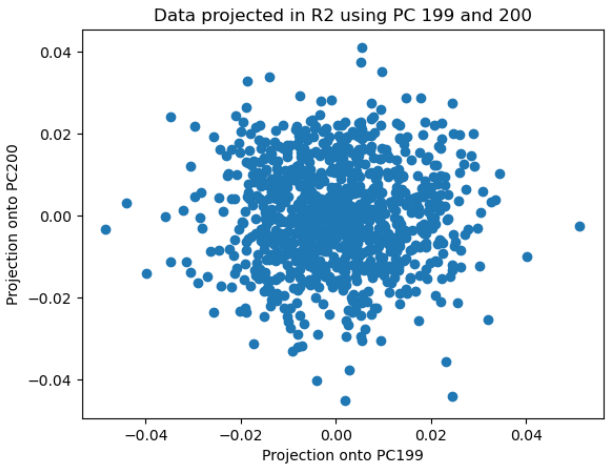
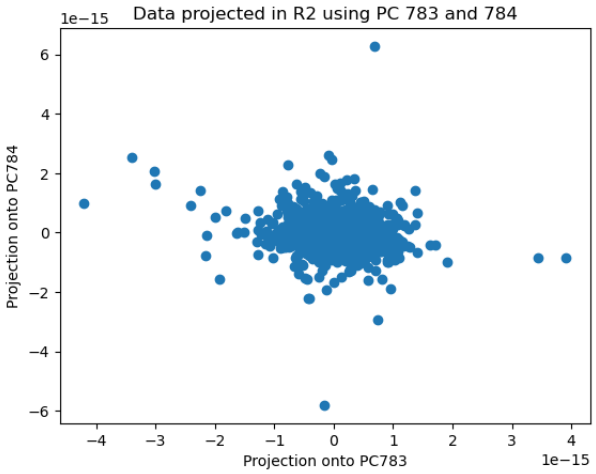
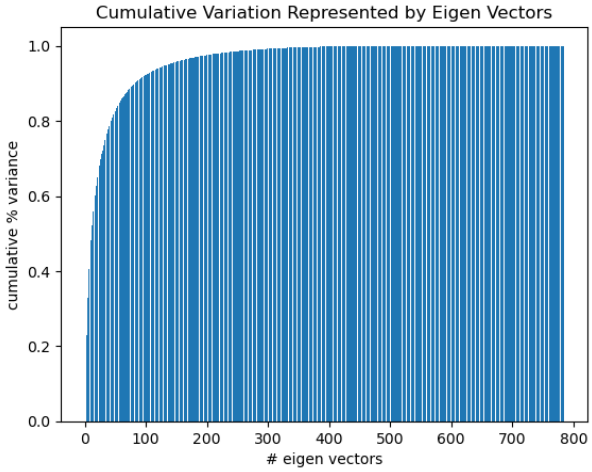
Qpca2 (10%): Plot the normalized eigenvalues (include the plot in your writeup). How many eigenvectors do you have to include before you've accounted for 90% of the variance? 95%? (Hint: see function cumsum.)

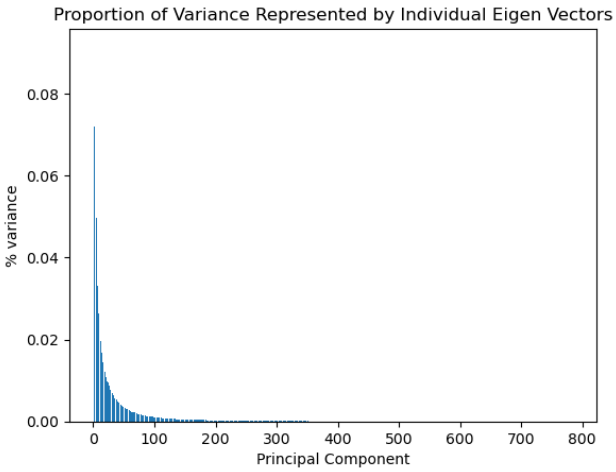
Since the eigen vectors are in dimension (784 x 1) we cannot plot them. What we can plot, however, is the embedding of the data which is the data projected into a lower dimensional space using the eigen vectors as basis vectors (i.e. the projections onto the principal components). Here we can show how the more important dimensions/eigen vectors correspond to more variation in the data. You need 82 eigen vectors (81th index) to account for 90% of the variance, and 136 eigen vectors (135th index) to account for 95% of the variance.



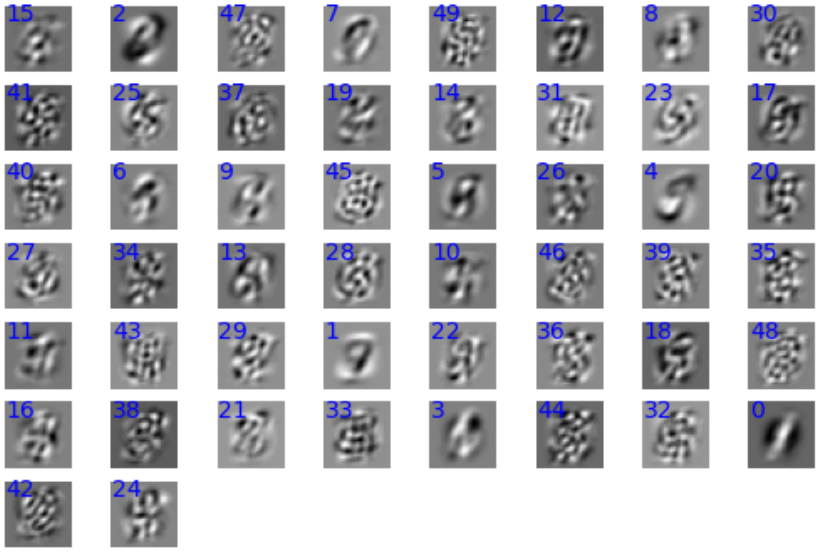




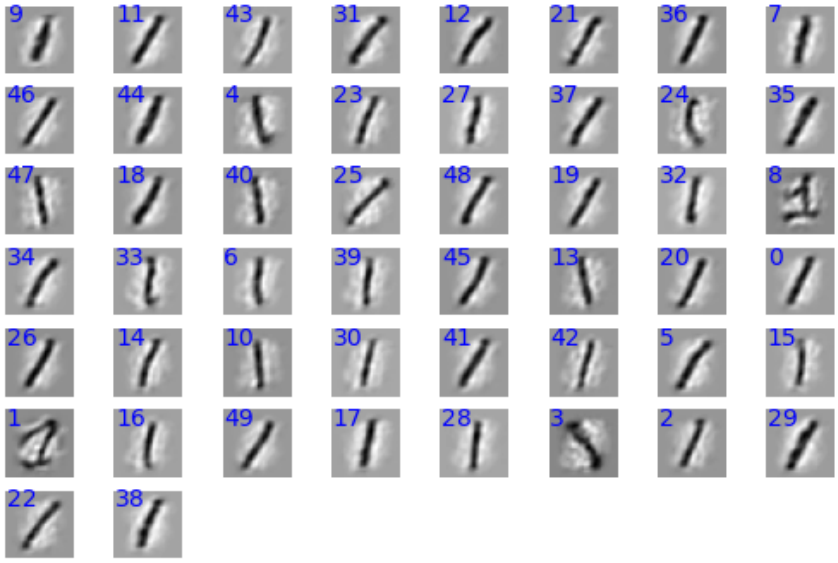




Qpca3 (5%): Do these look like digits? Should they? Why or why not? (Include the plot in your write-up.) (Make sure you have got rid of the imaginary part in pca.)



No these don't look like digits, and they shouldn’t, as these are just the eigen vectors corresponding to the top 50 dimensions with the most variance (i.e. basis vectors). In order to get an image that looks like a digit, you must have data somewhat similar to what you started with (i.e. data that represents pixels in an image, not basis vectors nor embedded data).
To reconstruct the original data you need to use the embedded data (P) and the transpose of the mapping/eigen vectors (Z.T will map the embedded data back to the original form like an inverse operation). If you only want to use the top 50 eigen vectors, call pca with 50, so P is of dimension (n x 50) and Z is of dimension (d x 50) so P@Z.T is of dimension (n x d) which is a "reconstruction" of the original image (it's an approximation, since you're storing less information in P and Z compared to the original X).
Below, showcases the reconstruction process outlined above using 50 eigen vectors for the first 50 digits.



Qsr1 (10%)

(1) Show that the probabilities sum to 1

(2) What are the dimensions of W? X? WX?

(1)

$$\sum_{k=1}^{K} P[y=k] = \sum_{k=1}^{K} \frac{e^{\vec{w_k}\cdot\vec{x}}}{\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}}}$$

$$= \frac{\sum_{k=1}^{K} e^{\vec{w_k}\cdot\vec{x}}}{\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}}}$$

$$= 1$$

(2) W is of dimension (K x D) where K is the number of classes and D is the number of features. This is because each row i of W is w_i and w_i is of dimension (1 x D) when it's represented as a row since it provides weights to the D features of an x vector. X i s of dimension (D x N) where N is the number of examples. This is because each column of X is a single example x, and x is of dimension (D x 1) when represented as a column vector. Hence, the dimension of WX is (K x N) since (K x D * D x N) = (K x N). This makes sense as their should be K probabilities returned for each of the N observations since we are using softmax.

Qsr3 (10%)

In the cost function, we see the line

W_X = W_X - np.max(W_X)

This means that each entry is reduced by the largest entry in the matrix.

(1) Show that this does not affect the predicted probabilities.

(2) Why might this be an optimization over using W_X? Justify your answer.

(1)

$$\frac{e^{\vec{w_k}\cdot\vec{x}-C}}{\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}-C}} = \frac{e^{\vec{w_k}\cdot\vec{x}}e^{-C}}{\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}}e^{-C}}$$

$$= \frac{e^{-C}e^{\vec{w_k}\cdot\vec{x}}}{e^{-C}\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}}}$$

$$= \frac{e^{\vec{w_k}\cdot\vec{x}}}{\sum_{j=1}^{K} e^{\vec{w_j}\cdot\vec{x}}}$$

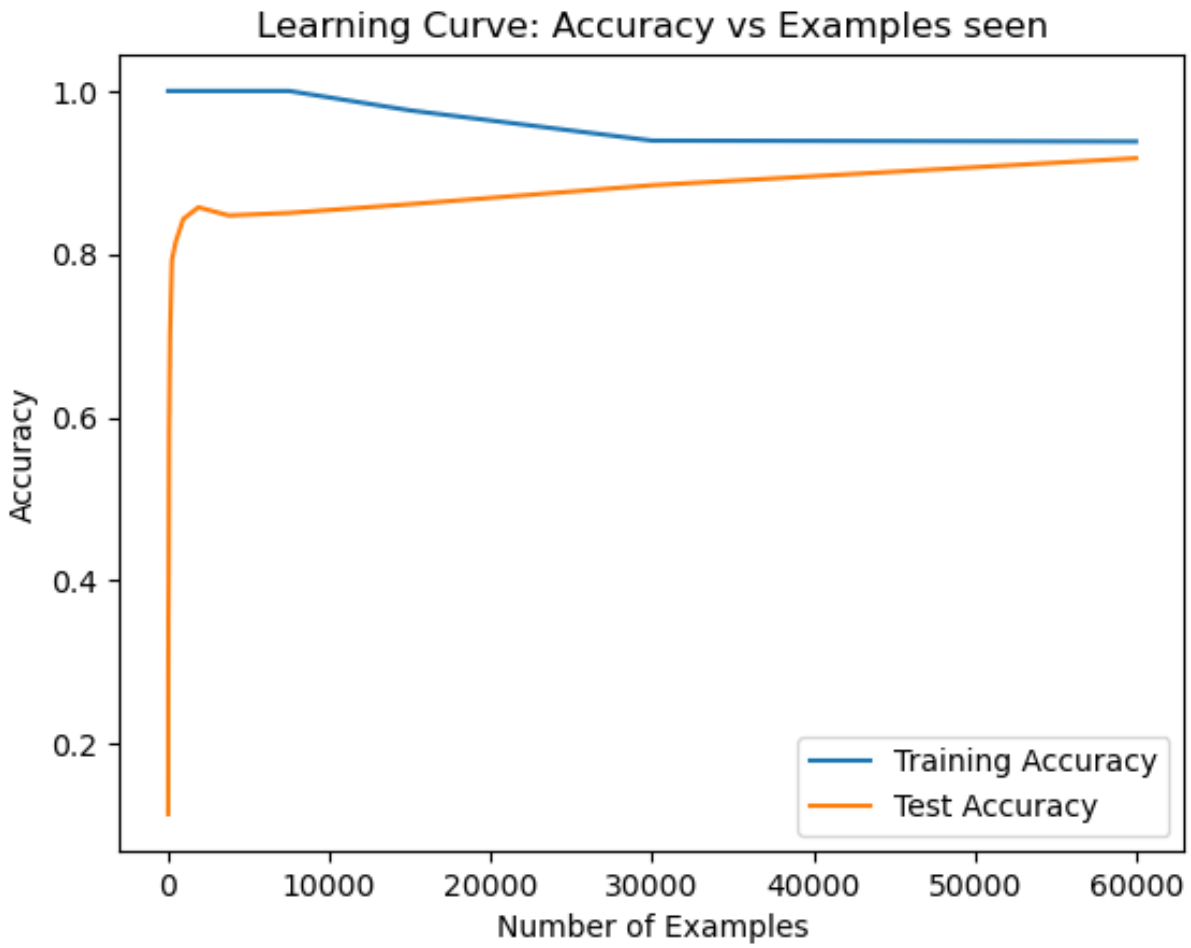
$$= P[y=k]$$

Where C is np.max(W_X) and <w_k,x> is an element of the W_X matrix (suppose x is pth observation; x=x_p). Hence, you can see that subtracting a constant C from each <w_i, x_p> component of the W_X matrix, ultimately does not impact the the final predicted probabilities as the e^-C terms will cancel out, leaving you with the original P(y=k) probability (where you use W_X elements in the exponent, without subtracting by the np.max).

(2) This is an optimization over using W_X because it will make the exponents smaller. Since e^x grows exponentially fast, we want smaller exponents to reduce the likelihood of overflow. This is especially the case for the denominator of the probability equation, since we are adding up all the exponentials. Also, working with smaller numbers is nicer because the numerical errors will be smaller too, so we get more accurate computations. For example, if your numerical method has an error of 10% and the actual result is 0.5, you could get a value like 0.55, but if your actual result is 500 then you could get a value like 550, which has a much larger error of 50.
Hence, by subtracting the max from all elements, we get to work with smaller numbers which reduces the likelihood of overflow errors and we reduce the size of numerical errors, all without affecting the final predicted probabilities.

Qsr4 (10%)

Use the learningCurve function in runClassifier.py to plot the accuracy of the classifier as a function of the number of examples seen. Include the plot in your write-up. Do you observe any overfitting or underfitting? Discuss and expain what you observe.



At first you can see the softmax regression model is severely overfitting as it has an accuracy of 100% on the training data, but it performs poorly on the test data with an accuracy of 10% - 60%. However, as we add more data we reduce the variance of our model as it starts to learn from more diverse examples rather than just a select few. This results in a lower variance model, without really compromising the complexity/bias, hence giving us a better test accuracy as you can see above. The training error decreases as the number of examples increases because the model starts to generalize rather than memorize; there are more examples to predict so it can't perfectly predict all the training data like it used to be able to, so it generalizes as best as it can which leads to some training error. There doesn't seem to be any sign of

underfitting as the training accuracy is always respectable at around +90% (the model is always capturing some pattern).
Hence we can see that our model is a high complexity (low bias) model, which is made better by providing more data as it reduces the variance of our model (which in turn, results in better test accuracies of ~90%).

Qnn1.3 Run in epochs
Report your final accuracy on the dev set. You can either use the default setting or tune the architecture (number of layers, size of layers and loss function) and hyperparameters (lr, batch_size, max_epoch).

Using the default setting, I got a final accuracy of 94.72% on the dev set.

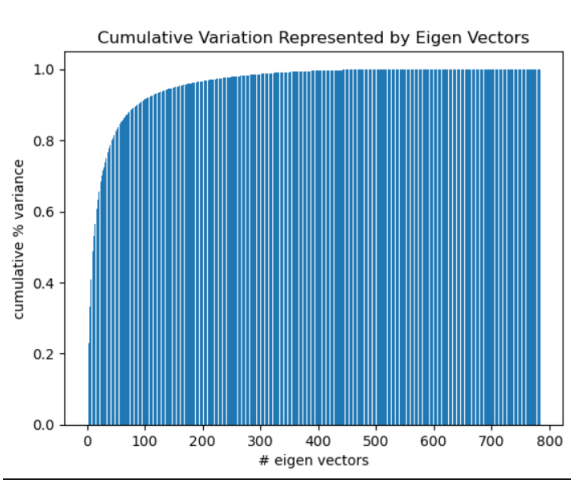
Qnn1.4 (No implementation needed for this question). When initializing the weight matrix, in some cases it may be appropriate to initialize the entries as small random numbers rather than all zeros. Give one reason why this may be a good idea.

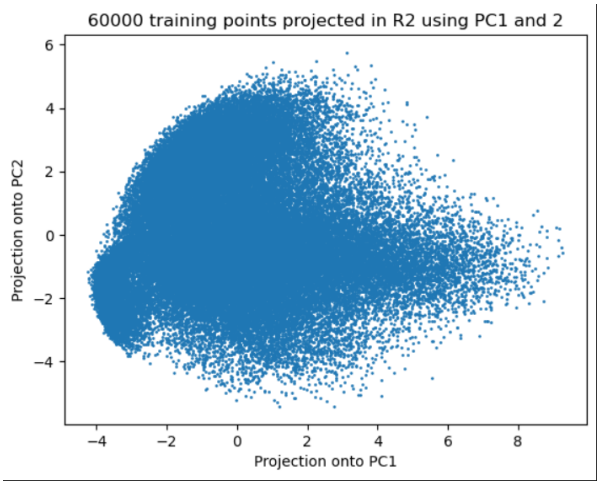
One reason why you don't initialize the weight matrix to be all 0's is that (if bias is also 0, or you're not using bias), after the first forward propagation each hidden layer's output would be the same since Wx (+b if b = 0) = 0 since W is 0. Then activating the 0 could give you 0 using relu or .5 using sigmoid. Regardless of what activation function you used, each vector output of a hidden layer would be uniformly the same (i.e. all 0's or all 0.5's), so the NN only outputs uniform results, regardless of your input vector. This is bad because in back propagation the gradient of Loss/W depends on the gradient on z/W where z = (W * output_of_prev_layer + b). Clearly, this reduces to the fact that gradient of Loss/W depends on something * the output of the previous hidden layer, which is uniformly the same value if W = 0. Hence, when you update W of a given layer using gradient descent, all the values will move together so you're W's won't be very useful (and it won't move at all if you use relu since then output_of_prev_layer = 0

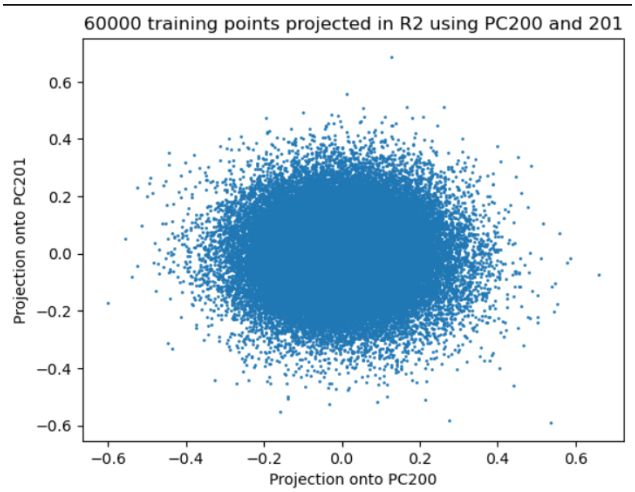
always). Also, doing multiple random initializations helps you find better minimas since the weight space isn't convex.

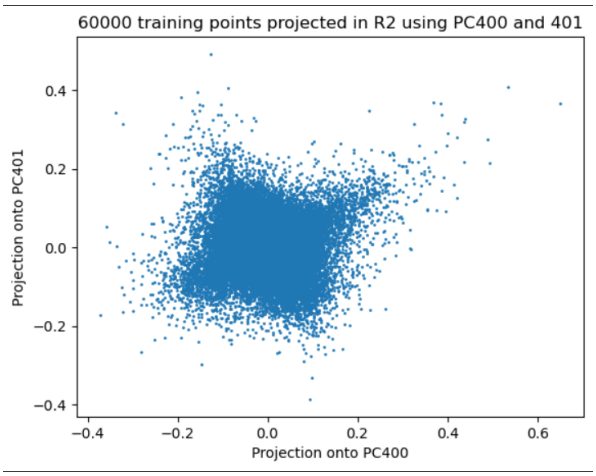
Qnn2 (Extra-Credit 15%) Try something new.

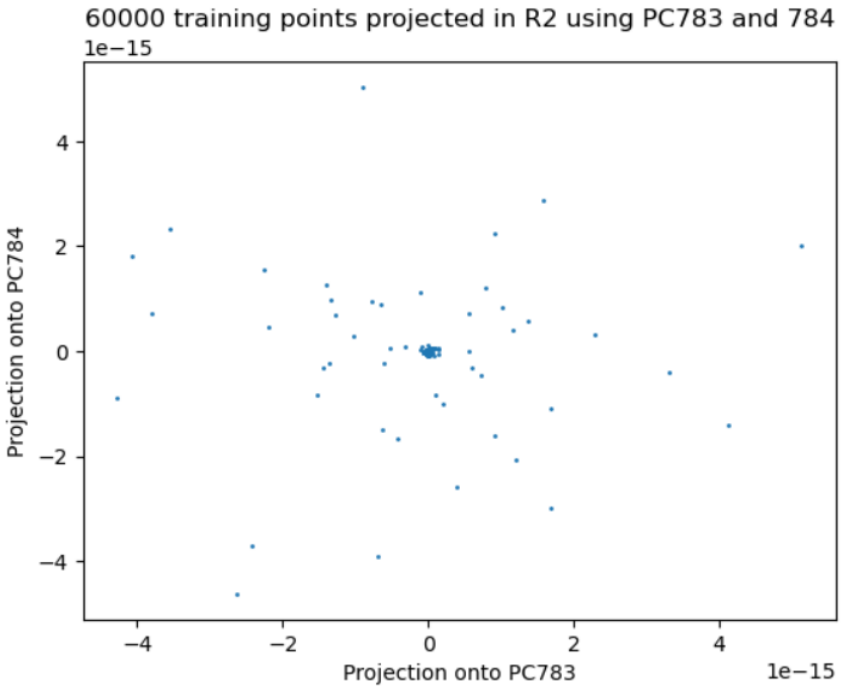
(1) Do dimension reduction with PCA. Try with different dimensions. Can you observe the trade-off in time and acc? Plot training time v.s. dimension, testing time v.s dimension and acc v.s. dimension. Visualize the principal components.

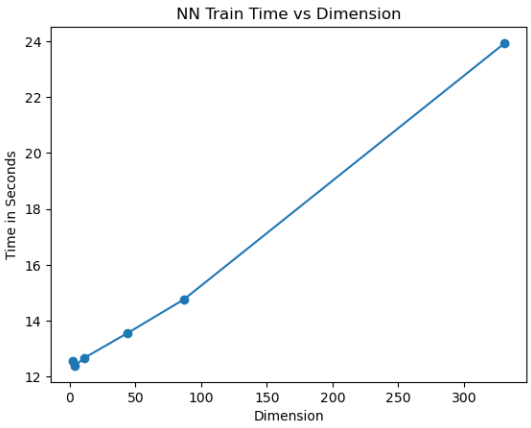


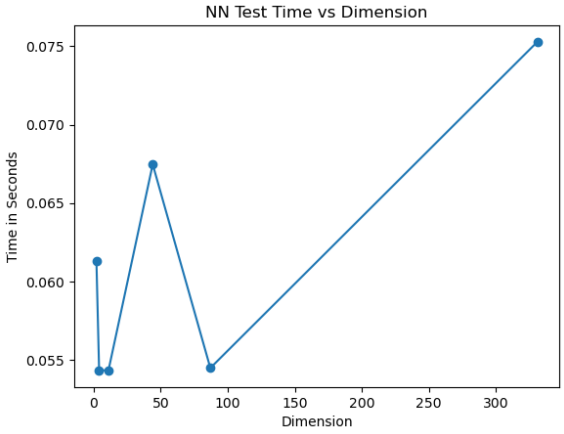


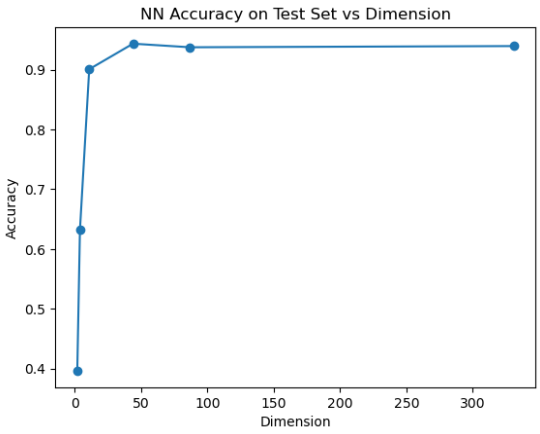


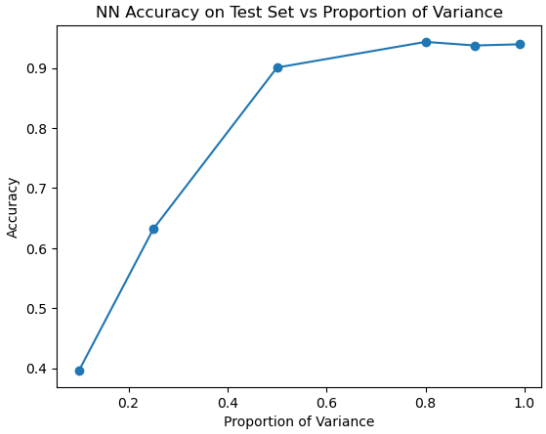



60000 training points projected in R2 using PC783 and 784
1e−15
Projection onto PC784
Projection onto PC783
1e−15


NN Train Time vs Dimension
Time in Seconds
Dimension


NN Test Time vs Dimension
Time in Seconds
Dimension


NN Accuracy on Test Set vs Dimension
Accuracy
Dimension


NN Accuracy on Test Set vs Proportion of Variance
Accuracy
Proportion of Variance

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 reduced the dimensionality of the mnist data set using the sklearn PCA library. I first ran the PCA on the training data to find all 784 principal components. Then I took the first principal components that made up 10%, 25%, 50%, 80%, 90%, and 99% of the variance of the training data. I used a NN with a relu activation function, 2 hidden layers, both with 256 hidden units (the same setting as in run_nn.py). I then provided the neural network the reduced training data as well as the reduced test data (the test data was projected/embedded into the same space/dimension using the training data's principal components) and computed the accuracies and times. Note that we could not use the test data's principal components since that would project it into a different space than the embedded training data, nor could we use the principle components of a combined train + test data set as that would be cheating since we would be using the test set to gain information (principle components).
The results show that lower dimensional data takes less time to train. This makes sense because it means that the first layer has less computations during forward propagation as your input layer matrix is smaller, and similarly during back propagation less gradients need to be calculated. This ends up shaving off a lot of time because training is an iterative process (you do forward and back propagation repeatedly over many batches), so saving a little time many times over accumulates. However, lower dimension data doesn't really improve test times which makes sense because making a prediction is a one time process and the only speed up you could get is from the original input layer which has a smaller weight matrix. However, this improvement is negligible due to modern hardware and it's why there is no apparent trend between dimension and prediction time.
You can also note that as dimensionality increases so does prediction accuracy. This makes sense because a greater dimension implies that more of the original training data is preserved (i.e. more features to work with). You can also see in the related variance vs accuracy graph, that as more variance is preserved in the projected data, the accuracy improves. Hence we can conclude that since training time and dimension has a direct relationship, and since dimension and accuracy has a direct relationship: training time and accuracy have a direct relationship. So the trade off is, more dimensions costs more training time, but produces better accuracies. The numbers prove it as well as ~24 seconds of train time resulted in 94% accuracy, whereas ~12 seconds of train time resulted in 30% accuracy. However there is a catch. Clearly, test accuracy cannot keep linearly increasing as dimensions increases, otherwise you could get 100%

accuracy on every data set (by just expanding # of features). So, even though test accuracy and dimensions are directly related, test accuracy levels off eventually. This also means that even if you train the model longer, eventually the improvement in test accuracy will level off. This can be seen in the results as well where ~14, ~15, ~24 seconds of training time all resulted in a model with ~94% accuracy.
You can also see that for the mnist data set, you don't need a lot of dimensions (relative to the total 784) to get a good accuracy. This makes sense as some of the pixels like the ones on the border may have 0 impact on the actual number written. This might explain why ~50 dimensions got a slightly better test accuracy than ~100 or ~350 dimensions did (since there's less noise to look at). However, you do need enough dimensions to represent a majority of the data's variance (at least 50%) to get good accuracies (+90% on test set). Hence, we can see that proportion of variance is generally a better way to select the number of principal components, rather than picking an arbitrary fraction of the total number of features.