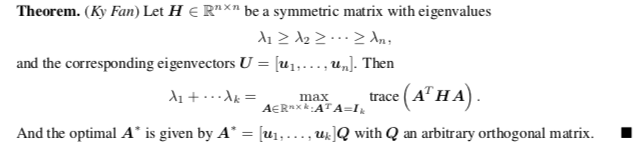
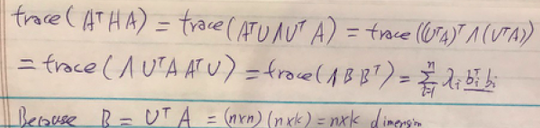
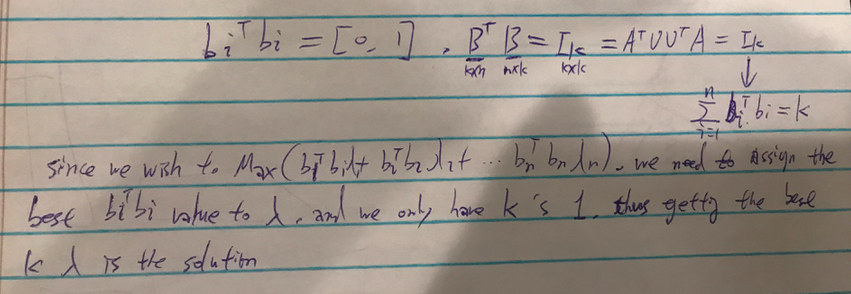


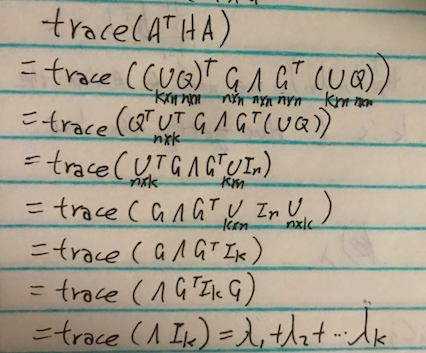
1. Proof







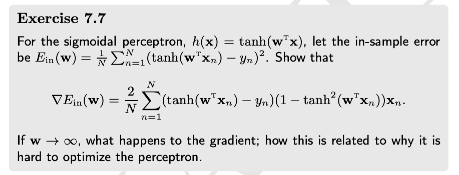
Since the **sum of bi^T \* bi** == K, and the single **bi^T \* bi** is large than 0 and smaller than 1, we choose to make the first K **bi^T \* bi** as 1, in order to maximize the K’s Lamda



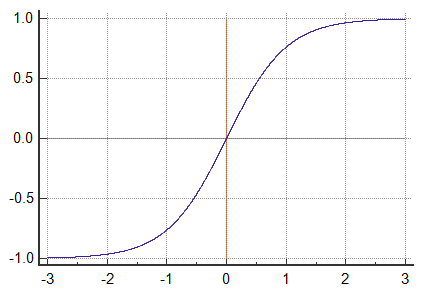
For the second proof, we put A\* into the formula – and decompose it as above. As the dimention of A\* is k\*n, and after calculating with trace, the result become as 

Here we sorted the diagonal by eigenvalue in the 2nd step



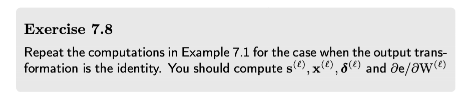


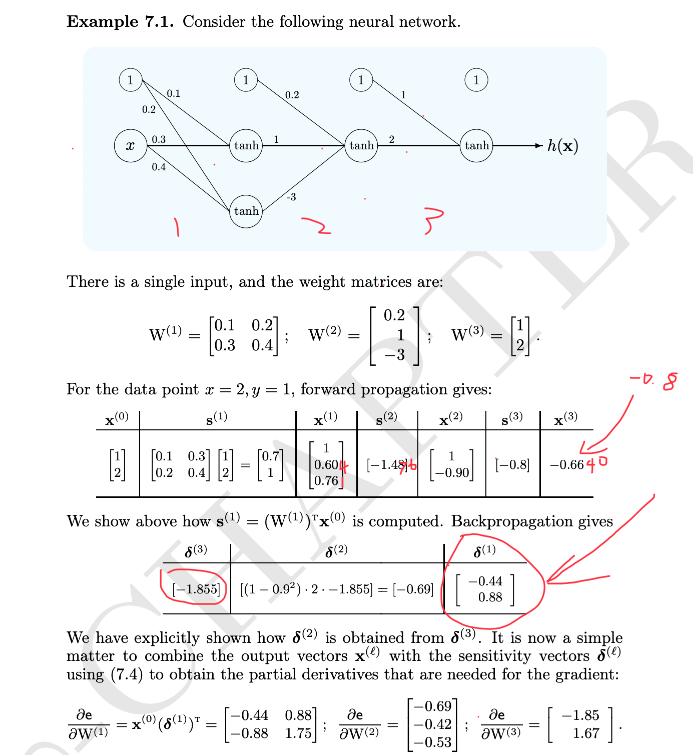
Since the tanh function is like below:

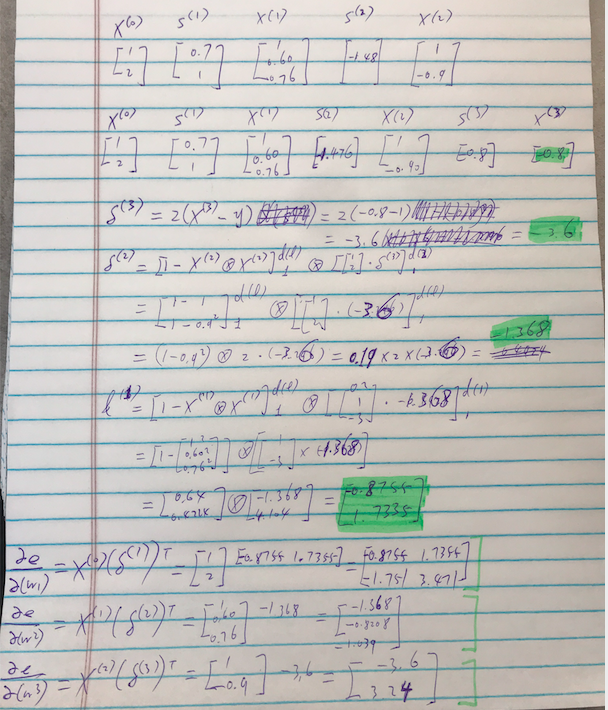


When w become extremely large, the Tanh is close to 1. And the gradient is close to (1-y^n)\*(1-1) such that the gradient is infinitely close to 0 🡪 which make the gradient almost unmovable. Other than that, when the gradient become extremely small, the machine will get overflow issue.

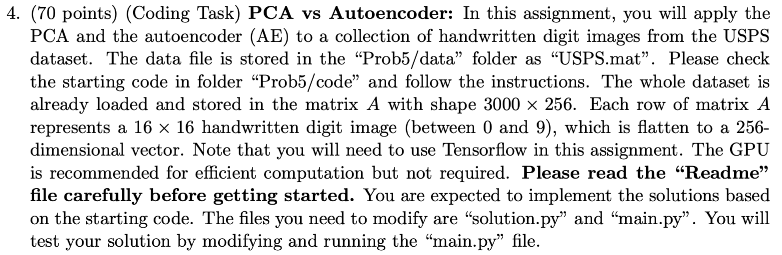










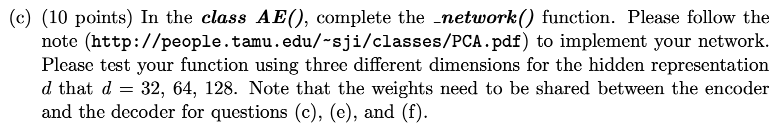


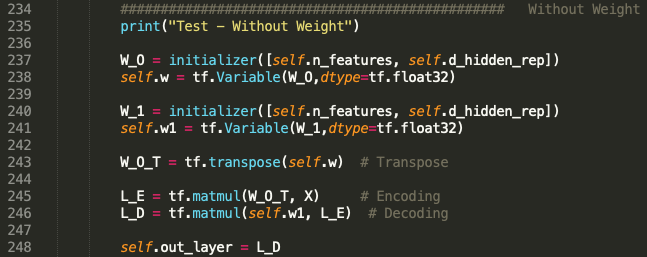






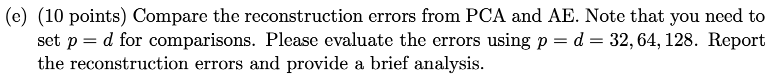




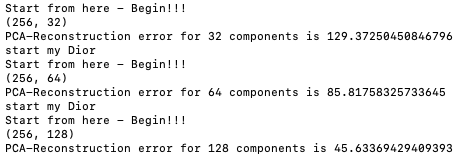








Result for the PCA:



Result for the AE:

AE-Reconstruction error for 32-dimensional hidden representation is 129.85673654754385

AE-Reconstruction error for 64-dimensional hidden representation is 86.21493390107862

AE-Reconstruction error for 128-dimensional hidden representation is 46.51216214968478

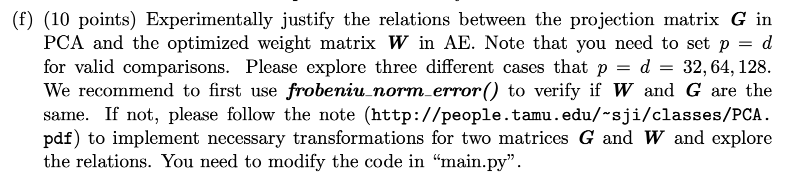
As we can see here the errors are similar.

The errors for PCA are: 129.37250, 85.175, 45.633

The errors for AE are: 129.4809, 86.07625,46.202

For PCA, it already use SVD to calculate the best principle components. Thus the reconstruction result is the best already. However, for AE, we start from a random weight – which is a raw data. After that we use gradient decent to train the weight with input data, and gradually the weight will be close to the best solution.

On the other hand, more components in PCA, or more hidden representatives mean that the reconstruction will be better – though it will consume more time.



🡪 For 32: the F\_norm is 8.020168445009627

🡪 For 64: the F\_norm is 11.40370080192625

🡪 For 128: the F\_norm is 16.05811228056582

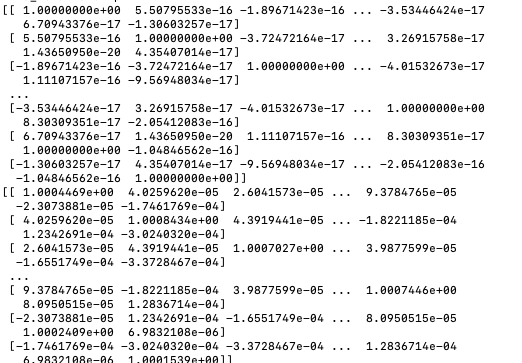
Here we can see the the W and G is not the same. From the note we noticed that “optimal W” can be different from a PCA solution. However, they can be calculated from another through an orthogonal matrix and the sets of possible solutions are the same 🡪 We believe G = W\*M where M is an orthogonal matrix. Thus, W^(-1)\*G = I

After computing as below:

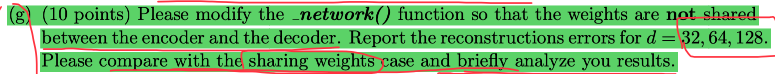
print((G.T).dot(G))

print((final\_w.T).dot(final\_w))

I found the result is a symmetry matrix:



So, (G.T).dot(G) and (final\_w.T).dot(final\_w) are both symmetry matrix



32 🡪 129.6810994637552

64 🡪 86.023737197696

128 🡪 46.03190729363156

Compared with “sharing weight” as below:

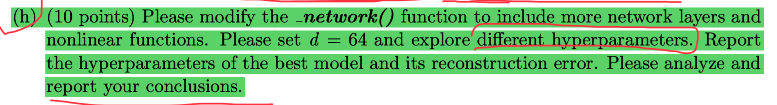
AE-Reconstruction error for 32-dimensional hidden representation is 129.85673654754385

AE-Reconstruction error for 64-dimensional hidden representation is 86.21493390107862

AE-Reconstruction error for 128-dimensional hidden representation is 46.51216214968478

We can see the “without sharing weight” is slightly better than “sharing weight”

The different shows that, even we use different weight before and after, the gradient decent can always help to train the weight with minimum the local error. Same as encoding, the decoding also adopt network to repeatly make weight closer to the true weight





When batch = 128, epoch = 300, the reconstruction error is: 86.12235684771204

When batch = 64, epoch = 600, the reconstruction error is: 86.347322198

When batch = 256, epoch = 300, the reconstruction error is: 86.128856522

When batch = 64, epoch = 300, the reconstruction error is: 86.406252425

When batch = 128, epoch = 150, the reconstruction error is: 86.391555939

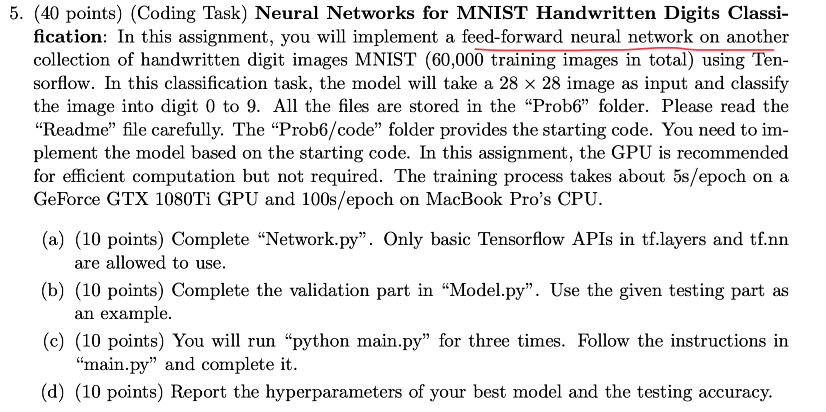
When batch = 128, epoch = 600, the reconstruction error is: 86.16059012058963

When batch = 128, epoch = 400, the reconstruction error is: 86.15082

When batch = 128, epoch = 250, the reconstruction error is: 86.279654065

When batch = 100, epoch = 250, the reconstruction error is: 86.23215161064034

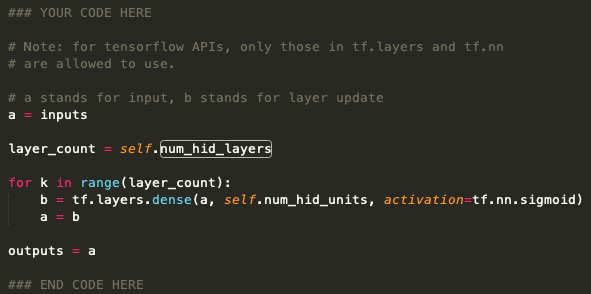
From the analysis, when the epoch is 300, and batch is 128, the result is the best. Since the shape of the data size is 256\*3000. We believe in this situation, 4% of the data size as batch to calculate the gradient decent is the best model. If we repeat the training for too much, the error goes up again.



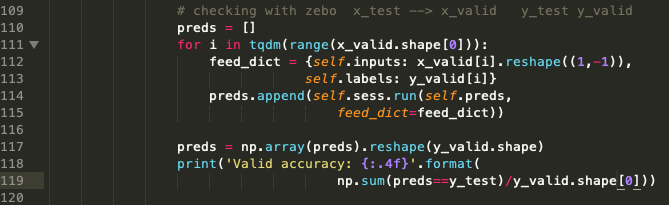
Answer:

1. See code

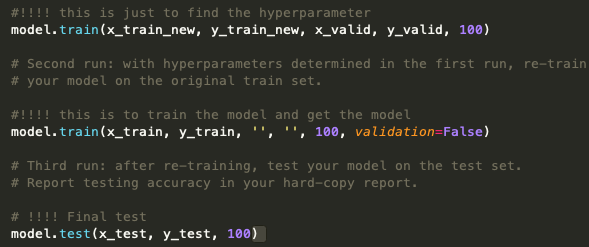
outputs = tf.layers.dense(inputs, self.num\_hid\_units, activation=tf.nn.tanh)

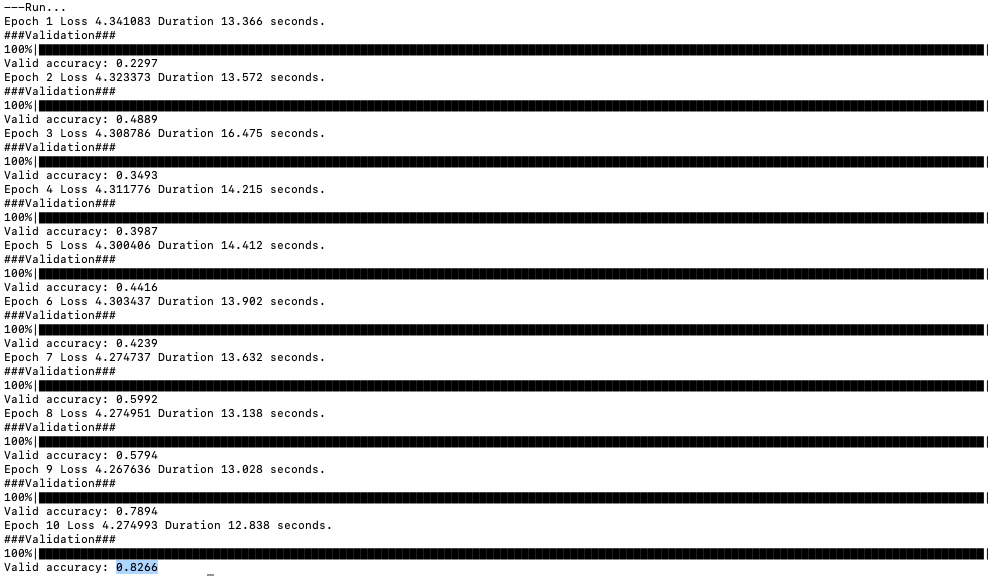












1. The best fit:
2. 'num\_hid\_layers' 🡪 definitely 2

flags.DEFINE\_integer('num\_hid\_layers', 2, 'the number of hidden layers')

flags.DEFINE\_integer('num\_hid\_units', 512, 'the number of hidden units in hidden layers')

flags.DEFINE\_integer('batch\_size', 128, 'training batch size')

flags.DEFINE\_integer('num\_classes', 10, 'number of classes')

flags.DEFINE\_string('modeldir', 'model', 'model directory')