6.036 Project 2

Dimitris Koutentakis

11 April 2017

For this project I used the following:

- Python 3.6.0
- Numpy 1.12.1
- Scipy 0.19.0
- PyYAML 3.12
- Cython 0.25.2
- h5py 2.7.0
- theano 0.9.0
- keras 2.0.2

Problem 1

All implemented code is attached.

4. The final error after running main.py, is:

$$\epsilon = 0.1005$$

5. The temperature parameter τ affects the magnitude of the θ vector we get. In specific, a larger τ would lead to a smaller updates, and thus to a θ that is smaler than otherwise. A larger value of τ would have a larger probability of a smaller θ and the opposite.

6. Changing the teperature parameter gives the following results:

   - For $\tau = 0.5$, we have an error of 0.084.
   - For $\tau = 1$, we get an error of 0.1005.
   - For $\tau = 2$, we get an error of 0.1261.

   It seems that the larger the temperature factor, the larger the error rate. This suggests that the probability distribution is normal and the magnitude change with the value of τ

9. When classifying the points by their $(mod3)$ labels, the error drops by a considerable amount. The new error we get is

$$\epsilon = 0.0768$$

   If on the other hand we also train the samples on the $(mod3)$ values, then the error jumps up to a value of:

$$\epsilon = 0.1872$$

It is pretty obvious why the error changes so much by applying $(mod 3)$ at different places. For example when training the points in the full space, but only checking their labels in the 3 dimensional space, it means that some of that error gets lost. Any points with label $Y = 1$ that were mis-classified as a 4 or a 7, will end up counting as "correct", as the $(mod 3)$ value of all these points is the same.
On the other hand, when we train the points on 3 values, then each misclassification ends up counting for more.

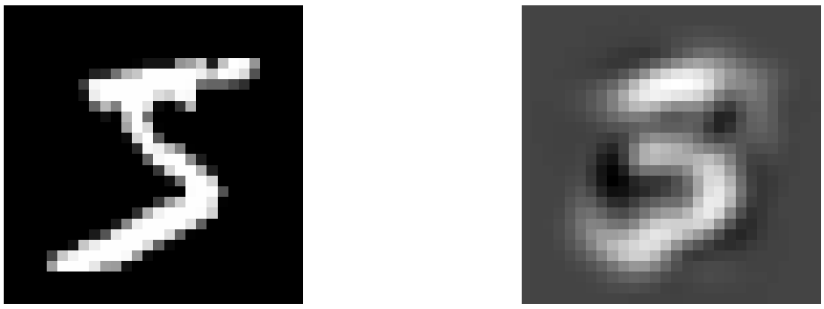
Problem 2

All code for Problem 2 is attached.

3. By implementing the PCA feature, and recalculating the error of the regression with the PCA data, we get an error that is very close to the original error. In specific, we now get an error of:

$$\epsilon = 0.1483$$

1. The original and reconstructed pictures can be viewed below:



(a) Original picture (b) Recunstructed picture

Figure 1: First reconstruction

(a) Original picture (b) Recunstructed picture

Figure 2: Second reconstruction

6. The cubic feature mapping $\Phi(x)$, is the following:

$$\Phi(x) = [x_1^3, \sqrt{3}x_1^2x_2, \sqrt{3}x_1^2, \sqrt{3}x_1x_2^2, \sqrt{6}x_1x_2, \sqrt{3}x_1, x_2^3, \sqrt{3}x_2^2, \sqrt{3}x_2, 1]$$

7. After implementing the cubic feature Kernel, we get an error much lower than any other method so far. The cubic feature kernel error is:

$$\epsilon = 0.0865$$

Problem 3

All code for problem 3 is attached.

4. In order to improve the efficiency of the training and accuracy of our network, we have to update the learning rate. This can be done in at least two ways.

   - Decrease the learning rate η as the number of updates increases,
   - Decrease the learning rate η when the magnitude of the gradient decreases.

5. The danger of having too many hidden units in your network is twofold.

   - First, computation cost increases significantly and
   - Secondly there is a large risk of overfitting to our data.

6. If we run the code for a significantly larger number of epochs, the training accuracy will keep getting better. The more the epochs, the better the training accuracy.

   On the other hand, as we increase the number of epochs, the test accuracy will initially increase and at some point will start declining, as we will be overfitting our data.

7. In order to hit the sweetspot where we train the network well, but do not overfit our data, we will have to try several different values of epochs and pick the one that does not overfit, while presenting the best performance.

Problem 4

The code for Problem 4 is attached.

1. The initial accuracy we get after the first training, is:

   Accuracy on test set: 0.9168

2. I tried several things in order to increase the accuracy, however it seemed like just tweaking the learning rate and the momentum factors was enough to increase the accuracy to above 98%. The changes I made were:

   - Increase learning rate to `lr=0.1`
   - Change momentum factor to: `momentum = 0.65`

   These changes ended up giving me an accuracy of:

   Accuracy on test set: 0.9806

3. After implementing all of the layers for the model, we getting the following accuracies:

   Test data accuracy: 0.9333

   Training data accuracy: 0.9847

Problem 5

1. The model.compile line activates our network model and configures the learning process with the parameters we pass. In specific, it has the 'categorical_crossentropy' loss function, uses stochastic gradient descent to optimize the model and accurazy in order to gauge how good it performs.
   The model.fit line does the approximation and the training of the model. In our case, it uses the training images as well as the two labels (for the two images). It runs for 30 epochs and has a batch size of 64.

2. For the observation section, I was able to increase the accuracy quite a bit.

   - Initially, mlp.py returned:
     `'dense\_2\_acc': 0.90400000000000003, 'dense\_3\_acc': 0.8932499999999999`
   - For the first model, I added a layer of 100 neurons to mlp.py. The new accuracies were:
     `'dense_3_acc': 0.91100000000000003, 'dense_4_acc': 0.90200000000000002.`
     The time did not change much.
   - For the second architecture I added two layers of 1000 neurons each, which resulted in:
     `'dense_4_acc': 0.91900000000000004, 'dense_5_acc': 0.90275000000000005`
     and a slightly longer runtime.
   - for my last test, I added a convolution, a pooling and a flattening layer, and reduced the number of epochs to 16. This resulted in the largest increase in acuracy increase, giving:
     `dense_2_acc: 0.9633 - dense_3_acc: 0.9534` . This method

however took much longer to run (more than 10 times longer than the other methods).

In the end, I believe that my second attempt had the best result for the runtime.