ECE1504: Assignment 1

Linear and Logistic Regression

Due: 29 Oct 2019

Tianyi Xie 1000570679

Yisi Zou 1001826321

Contribution percentage is the same for both group members

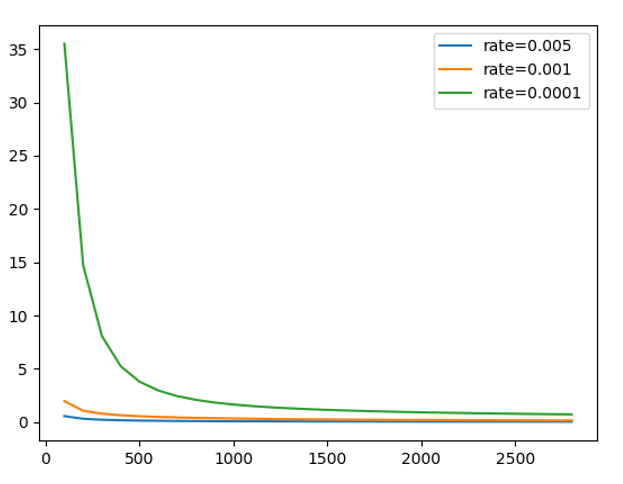
Part1 Linear Regression



The TensorFlow Script is attached in appendix 1.1

Because the training set has size 3500 and the mini batch has size 500, after shuffle the dataset at the beginning of each epoch, we partition the data set to 3500/500=7 mini batch of size 500, and each epoch consists of 3500/500=7 iterations. According to the problem, we should have 20000 iterations and one iteration is an SGD update through the entire pass of one mini-batch, so the total number of epochs should be total number of iteration over the iteration of each epochs, which is 20000/ (3500/500) ≈ 2857.14 epochs. Because the number of epochs should be an integer, we choose 2858 epochs which is the smallest integer greater than 2857.14.

Here is the plot for training loss function vs. number of epochs



As we can see from the plot above, among these three different rates, the best learning rate is 0.005 because for the same number of epochs, the training loss is smallest, and the curve converges fastest. So, we assumed the training convergence is better for larger training rate values. However, the assumption was proved to be incorrect because when we tried the learning rate of 0.01, the loss function diverges. This shows that choosing a suitable learning rate is important to train linear regression model using SGD algorithm. Too small learning rate can make the loss function converges slowly and too large learning rate can lead to divergence.

Note: We assume allow\_smaller\_final\_batch=True throughout the entire homework. We also tried when it is False, and the difference between these two methods is not significant enough to affect our conclusion. In addition, we notice that the MSE loss equation in the hand out is different from the default tf.losses.mean\_squared\_error() function by a factor of 1/2. We also tried including this factor, yet the difference between the outcomes is not significant enough to affect our conclusion. Thus we stay with the default function.



The TensorFlow Script is attached in appendix 1.2

We assume the learning rate chosen from point 1 is 0.005 which is the best learning rate found in part 1.1.

For batch size=500, the training dataset is partitioned to 3500/500=7 mini batches and each epoch consists of 3500/500=7 iterations. The total number of epochs is 20000/ (3500/500) ≈ 2857.14 epochs. Since the number of epochs should be an integer, we choose 2858 epochs which is the smallest integer greater than 2857.14.

For batch size=1500, the training dataset is partitioned to 3500/1500≈ 2.33 mini batches and each epoch consists of 3500/1500 ≈ 2.33 iterations. Since iteration should be an integer, we choose 3 iterations which is the smallest integer greater than 3500/1500. The total number of epochs is 20000/3 ≈ 6666.67 epochs. Because the number of epochs should be an integer, we choose 6667 epochs which is the smallest integer greater than 6666.67.

For batch size=3500, the training dataset is partitioned to 3500/3500=1 mini batch and each epoch consists of 3500/3500=1 iteration. The total number of epochs is 20000/ (3500/3500) = 20000 epochs.

Here is the result for different batch size

|  |  |  |  |
| --- | --- | --- | --- |
| Batch size (B) | Final epoch number | Final training MSE | Training time(s) |
| 500 | 2858 | 0.046957213 | 88.78387046 |
| 1500 | 6667 | 0.046495214 | 184.7604892254 |
| 3500 | 20000 | 0.047861014 | 505.188759089 |

As we can see from the table above, the final training MSE are almost the same for different batch size. The best mini-batch size is 500 because it has the least amount of training time. Based on our observation, the size of batch does not have a big influence for the final training MSE. However, if we choose smaller batch size, the training time can be reduced obviously. So, it is important to choose proper batch size to reduce training time.

The TensorFlow Script is attached in appendix 1.3

Here is the result for different weight decay coefficient

|  |  |  |  |
| --- | --- | --- | --- |
| Weight decay coefficient () | Final loss | Validation accuracy | Test Accuracy |
| 0. | 0.047116764 | 94% | 94.48% |
| 0.001 | 0.0512425 | 95% | 95.17% |
| 0.1 | 0.03094033 | 98% | 97.24% |
| 1 | 0.040750615 | 97% | 97.24% |

As we can see from the table, the validation accuracy for weight decay coefficient are 94%, 95%, 98% and 97%. The best based on the validation accuracy is 0.1 and the test accuracy for this is 97.24%.

When  is small, we are overfitting the training set. Therefore, the accuracy on the validation set is smaller than the best set of parameters. When is big, we are under-fitting the training set. Therefore, the accuracy on the validation set is also smaller than the best set of parameters.

The reason why we need to tune  using validation set is because the optimizer will always try to minimize the cost on the training set. Therefore, we would get similar cost (thus similar accuracy) for different value of if we only look at the training sets. To find the best, we need to test our parameters using a different set of data. That is why we are using validation sets.



The TensorFlow Script is attached in appendix 1.4 and normal\_equation

Compare the one with in question 1.2 for final training MSE and computation time and question 1.3 for accuracy

Here is the result of final training MSE, accuracy and computation time for SGD and normal equation

|  |  |  |  |
| --- | --- | --- | --- |
|  | Final training MSE | Validation accuracy | Computation time |
| SGD | 0.046957213 | 94% | 88.78387046 |
| Normal equation | 0.04750695 | 96% | 226.09647989 |

As we can see from the table above, SGD is more practical when we have lots of training samples because for the almost same training MSE and validation accuracy, the computation time of using SGD is much less than using normal equation. Comparing to the normal equation, SGD makes faster steps, thus even though it takes random steps, it is faster than the normal equation over all.

Part2 logistic regression

2.1 Binary cross-entropy loss

2.1.1

The TensorFlow Script is attached in appendix 2.1.1

According to the problem, we used and

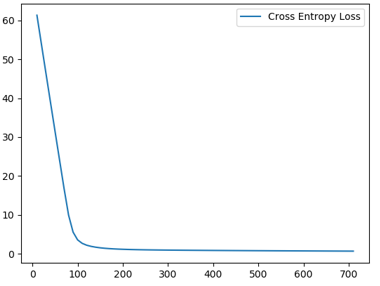
We used the “increase by 3” method to tune the learning rate, and choose learning rate to be 0.03, 0.01, 0.003, and 0.001.

Here is the result for different learning rates

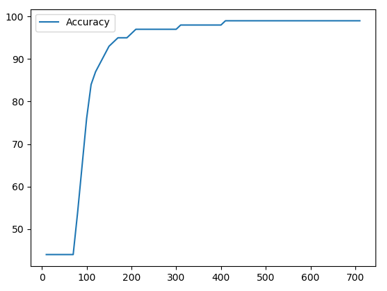
|  |  |  |  |
| --- | --- | --- | --- |
| Learning rate ( | Cross entropy loss | Validation accuracy | result |
| 0.03 | 0.077531 | 97% | overfitting |
| 0.01 | 0.243563 | 98% | overfitting |
| 0.003 | 0.656476 | 99% | best |
| 0.001 | 1.080894 | 97% | underfitting |

From the table above, we found that we have highest validation accuracy at learning rate=0.003. The test accuracy in this case is 97.24%. When the learning rate is smaller than 0.003, we underfit the training set; and when the learning rate is larger than 0.003, we overfit it. Overfitting and underfitting both causes the accuracy on the validation set smaller than the best set of parameters. Choosing the suitable learning rate can reduce the rate of overfitting and underfitting.

Here is the plot of training cross entropy vs. the number of epochs



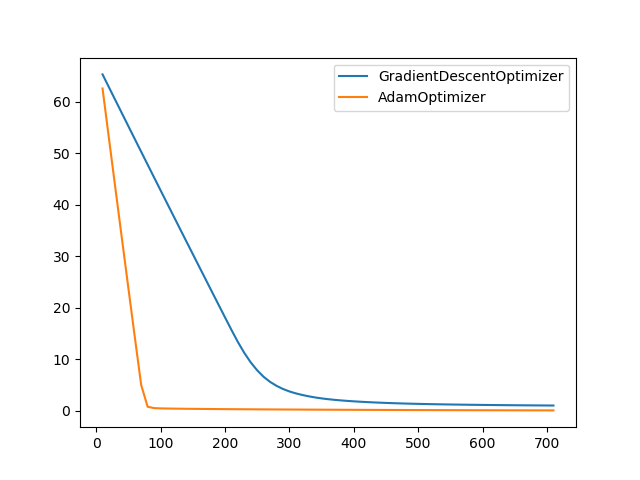
Here is the plot of validation accuracy vs. the number of epochs



2.1.2

The TensorFlow Script is attached in appendix 2.1.2

Here is the plot of training cross entropy curve using Adam Optimizer and SGD



From the plot, we found the training across entropy loss using Adam optimizer converges faster than using SGD.

From the website, <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>, we found a lot benefits of using Adam. Compared with SGD which keeps the same learning rate during training, Adam can adapt the learning rate from the average first moment and the average of the second moment of the gradients. Also, Adam optimizer calculates an exponential moving average of the gradient and the squared gradient.

By changing the step size of each gradient descent, Adam optimizer computes more efficient. Compared with using SGD, the training loss can converges faster by using Adam optimizer.

2.1.3

The TensorFlow Script is attached in appendix 2.1.3 and normal\_equation

Below is a table for linear regression using normal equation

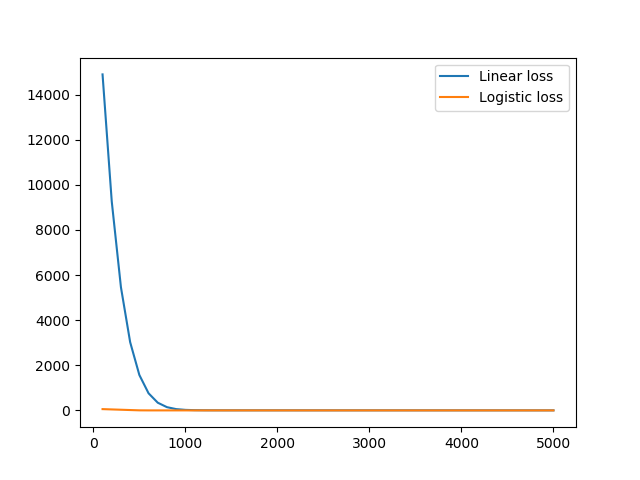
|  |  |  |  |
| --- | --- | --- | --- |
| Cost | Training accuracy | Validation  accuracy | Test  accuracy |
| 0.061081 | 95.60% | 95% | 93.79% |

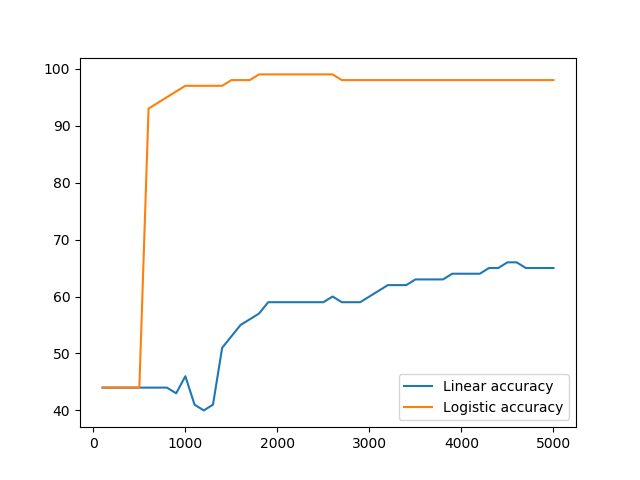
Below is a table for logistic regression

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Learning rate | Cost | Training accuracy | Validation  accuracy | Test  accuracy | Comment |
| 0.03 | 0.001935 | 99.97% | 98% | 97.24% | Overfitting |
| 0.01 | 0.006318 | 99.89% | 99% | 97.93% | Best |
| 0.003 | 0.017103 | 99.43% | 97% | 97.93% |  |
| 0.001 | 0.042125 | 98.40% | 97% | 97.93% | Underfitting |

As shown in the table, logistic regression generally has higher accuracy comparing to linear regression

Here is the plot of training cross entropy for linear and logistic regression



Here is the plot of validation accuracy for linear and logistic regression 

As shown in the graphs above, logistic regression (using cross-entropy loss) converges way faster than linear regression (using MSE loss). The validation set accuracy is also way higher. Although we could see that there is an overfitting problem for logistic regression, which is caused by having zero weight decay. We also compute the losses of the validation sets to show the convergence errors. For linear regression, the convergence error is 1.782572. For logistic regression, it is 0.028406. Clearly logistic regression has a smaller convergence error comparing to linear regression.

2.1 Multi-class classification

2.2.1

The TensorFlow Script is attached in appendix 2.2.1

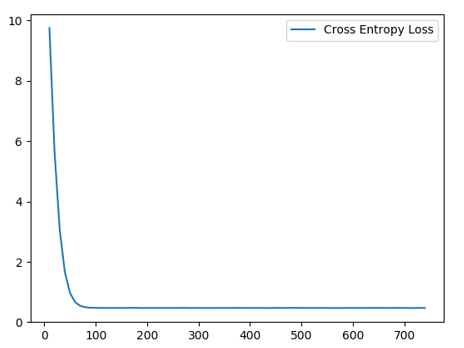
Here is the table for the record of tuning the learning value:

|  |  |  |  |
| --- | --- | --- | --- |
| Learning rate | Loss | Validation accuracy | Testing accuracy |
| 0.01 | 0.504793 | 89.50% | 89.34% |
| 0.003 | 0.479290 | 89.10% | 88.97% |
| 0.001 | 0.472063 | 90.50% | 89.38% |
| 0.0003 | 0.471696 | 90.00% | 89.23% |

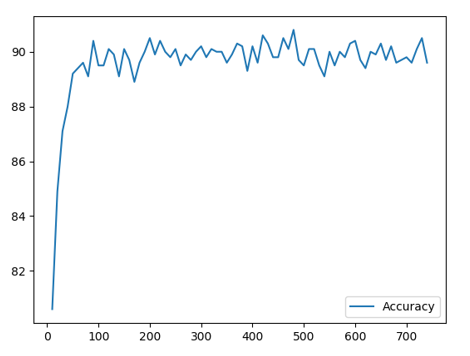
Similar as before, we use “Increase by 3” method to tune the learning rate. It turns out that when learning rate is 0.03, the cross entropy loss diverges. Therefore we only shows the cases when learning rates are smaller than 0.03. It turns out that the best learning rate is 0.001, and the corresponding test classification accuracy is 89.38%.

After we have tuned the learning rate, we found the best value of learning rate is 0.001.

Here is the curve of cross-entropy loss vs. the number of epochs

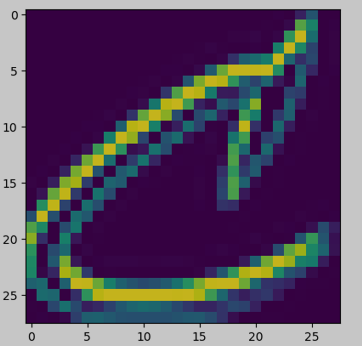


Here is the curve of classification accuracy vs. the number of epochs



The performance of multi-class classification is worse than the binary class problem. Both the validation accuracy and the test accuracy is lower than the previous problem (part 1 and part 2.1).

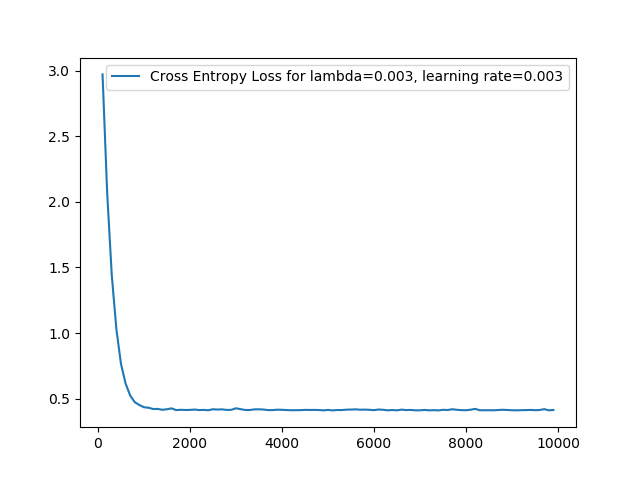
Here is example of miss-labelled picture



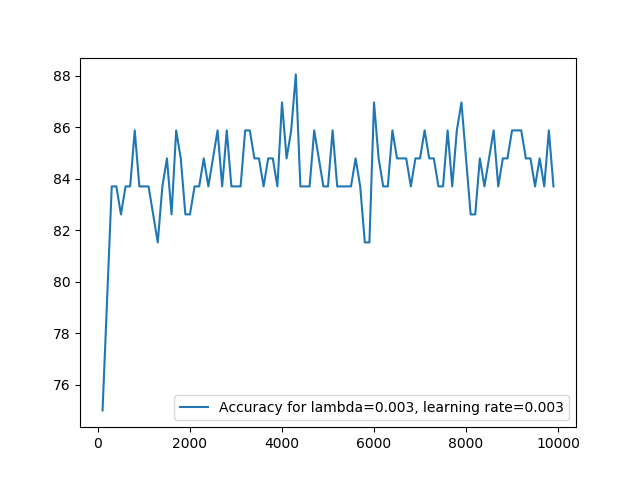
The pictures shows the letter ‘c’, however, it miss-labelled as ‘E’. We found it pretty reasonable because that ‘C’ is not in a regular curly shape. To improve the performance, making the neural network “deeper” should be a good approach.

The TensorFlow Script is attached in appendix 2.2.2

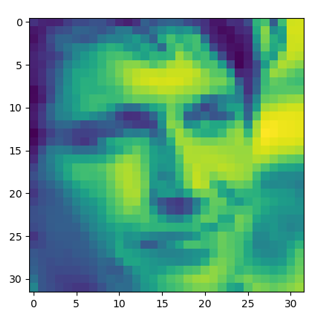
Again we use “Increase by 3” method to tune the learning rate. We have tried 5 different learning rate and 5 different weight decay coefficient. It turns out that when learning rate is 0.003 and weight decay coefficient is 0.003, we observed the best validation and test accuracy. The best test accuracy is 89.25%, and the graphs are shown below.

Here is the curve of cross-entropy loss vs. the number of epochs

Here is the curve of classification accuracy vs. the number of epochs



Also, one failure case is also shown below.



The ground truth is 1, which means it is a picture of Gerard Butler. But it is miss-predicted as 4, or Daniel Radcliffe. It is sort of reasonable for the algorithm to make such a mistake, as they are both male and the picture has low resolution. Once again, making the neural network “deeper” could improve the performance.

**APPENDIX**

**PART 1**

**1.1**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

# Initialize parameters

learning\_rates = [0.005, 0.001, 0.0001]

batch\_size = 500

decay\_coef = 0

iteration = 20000

display\_step = 100

train\_size = 3500

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

MSE\_loss = tf.losses.mean\_squared\_error(Y\_hat, Y)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = MSE\_loss + weight\_decay\_loss

# Iterate for different learning rates

for learning\_rate in learning\_rates :

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == train\_size :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 100 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(epoch\_counter)

loss\_val.append(curr\_loss)

# Update the training batch

if batch\_index + batch\_size > train\_size :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = train\_size

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter+1, "Final loss =", final\_loss)

# Plot the graph

plt.plot(epoch\_num, loss\_val, label='rate='+str(learning\_rate))

# Show the graph

plt.legend()

plt.show()

**1.2**

import numpy as np

import tensorflow as tf

import time

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

# Initialize parameters

learning\_rate = 0.005

batch\_sizes = [500, 1500, 3500]

decay\_coef = 0

iteration = 20000

display\_step = 1000

train\_size = 3500

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

MSE\_loss = tf.losses.mean\_squared\_error(Y\_hat, Y)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = MSE\_loss + weight\_decay\_loss

# Iterate for different batch sizes

for batch\_size in batch\_sizes :

# Start the timer before the Initialization of the optimizer

start\_time = time.time()

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == train\_size :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 1000 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the training batch

if batch\_index + batch\_size > train\_size :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = train\_size

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss and time spent

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter+1, "Final loss =", final\_loss)

end\_time = time.time()

# Show the time spent for the training

print("Time spent =", end\_time-start\_time)

**1.3**

import numpy as np

import tensorflow as tf

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

testData = np.asarray(testData)

testData = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))

# Initialize parameters

learning\_rate = 0.005

batch\_size = 500

decay\_coefs = [0, 0.001, 0.1, 1]

iteration = 20000

display\_step = 1000

train\_size = 3500

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Iterate for different decay coefficients

for decay\_coef in decay\_coefs :

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

MSE\_loss = tf.losses.mean\_squared\_error(Y\_hat, Y)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = MSE\_loss + weight\_decay\_loss

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == train\_size :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 1000 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the training batch

if batch\_index + batch\_size > train\_size :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = train\_size

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

# compute accuracy for validation and test data

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = (predict\_soft >= 0.5).astype(int)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct) # / 100 \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = (test\_predict\_soft >= 0.5).astype(int)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / 145 \* 100

# Show the final loss, accuracy of the validation set and test set

print("Epoch =", epoch\_counter+1, "Final loss =", final\_loss)

print("validation accuracy =", accuracy, "per cent")

print("test\_accuracy = {:.2f} per cent".format(test\_accuracy))

**1.4**

import numpy as np

import tensorflow as tf

import time

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

# Initialize parameters

learning\_rate = 0.005

decay\_coef = 0

iteration = 20000

display\_step = 1000

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

MSE\_loss = tf.losses.mean\_squared\_error(Y\_hat, Y)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = MSE\_loss + weight\_decay\_loss

# Start the timer before the Initialization of the optimizer

start\_time = time.time()

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

for i in range(iteration) :

# Run the optimizer

sess.run(optimizer, feed\_dict={X:trainData, Y:trainTarget})

# Display logs per 1000 epochs

if i % display\_step == display\_step-1 :

# Compute current loss

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", i+1, "loss =", "{:.9f}".format(curr\_loss))

print("Optimization Finished!")

# compute final loss and time spent

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Final loss =", final\_loss)

end\_time = time.time()

# compute accuracy for validation data

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = (predict\_soft >= 0.5).astype(int)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct) # / 100 \* 100

# Show the results

print("validation accuracy =", accuracy, "per cent")

print("Time spent =", end\_time-start\_time)

**PART 2**

**2.1.1**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

testData = np.asarray(testData)

testData = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))

# Initialize parameters

learning\_rates = [0.03, 0.01, 0.003, 0.001]

batch\_size = 500

decay\_coef = 0.01

iteration = 5000

display\_step = 10

train\_size = 3500

test\_size = 145

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

cross\_entropy = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=Y, logits=Y\_hat)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = cross\_entropy\_loss + weight\_decay\_loss

# Iterate for different learning rates

for learning\_rate in learning\_rates :

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

accur\_val = []

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == train\_size :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 10 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss and accuracy

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = (predict\_soft >= 0.5).astype(int)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct)

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(epoch\_counter)

loss\_val.append(curr\_loss)

accur\_val.append(accuracy)

# Update the training batch

if batch\_index + batch\_size > train\_size :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = train\_size

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

# compute accuracy for validation and test data

valid\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

valid\_predict = (valid\_predict\_soft >= 0.5).astype(int)

valid\_correct = (valid\_predict == validTarget).astype(int)

valid\_accuracy = np.sum(valid\_correct) # / 100 \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = (test\_predict\_soft >= 0.5).astype(int)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / test\_size \* 100

print("Epoch =", epoch\_counter+1, "Final loss =", final\_loss)

print("valid\_accuracy =", valid\_accuracy, "per cent")

print("test\_accuracy = {:.2f} per cent".format(test\_accuracy))

# Plot and show the graphs

plt.plot(epoch\_num, loss\_val, label='Cross Entropy Loss')

plt.legend()

plt.show()

plt.plot(epoch\_num, accur\_val, label='Accuracy')

plt.legend()

plt.show()

**2.1.2**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

# Initialize parameters

learning\_rate = 0.001

batch\_size = 500

decay\_coef = 0.01

iteration = 5000

display\_step = 10

train\_size = 3500

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

cross\_entropy = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=Y, logits=Y\_hat)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = cross\_entropy\_loss + weight\_decay\_loss

# Store the initializers for different optimize models

optimizers = []

optimizers.append(tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss, name='GradientDescentOptimizer'))

optimizers.append(tf.train.AdamOptimizer(learning\_rate).minimize(loss, name='AdamOptimizer'))

# Iterate for different optimize models

for optimizer in optimizers :

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == train\_size :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 10 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = (predict\_soft >= 0.5).astype(int)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct)

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(epoch\_counter)

loss\_val.append(curr\_loss)

# Update the training batch

if batch\_index + batch\_size > train\_size :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = train\_size

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

print("Epoch =", epoch\_counter+1, "Final loss =", final\_loss)

# Plot the graph

plt.plot(epoch\_num, loss\_val, label=optimizer.name)

# Show the graph

plt.legend()

plt.show()

**2.1.3**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

testData = np.asarray(testData)

testData = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))

# Initialize parameters

learning\_rates = [0.03, 0.01, 0.003, 0.001]

decay\_coef = 0

iteration = 5000

display\_step = 100

train\_size = 3500

test\_size = 145

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.float32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 1]), name="W")

b = tf.Variable(tf.random\_uniform([1,1]), name="b")

# Compute the losses

Y\_hat = tf.add(tf.matmul(X, W), b)

MSE\_loss = tf.losses.mean\_squared\_error(Y\_hat, Y)

cross\_entropy = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=Y, logits=Y\_hat)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss\_Linear = MSE\_loss + weight\_decay\_loss

loss\_Logistic = cross\_entropy\_loss + weight\_decay\_loss

losses = {}

losses["Linear"] = loss\_Linear

losses["Logistic"] = loss\_Logistic

plot\_data = {}

for learning\_rate in learning\_rates :

# Iterate for different losses

for name, loss in losses.items() :

optimizer = tf.train.AdamOptimizer(learning\_rate).minimize(loss, name='AdamOptimizer')

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

accur\_val = []

for i in range(iteration) :

# Run the optimizer

sess.run(optimizer, feed\_dict={X:trainData, Y:trainTarget})

# Display logs per 100 epochs

if i % display\_step == display\_step-1 :

# Compute current loss and accuracy

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = (predict\_soft >= 0.5).astype(int)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct)

print("Epoch =", i+1, "loss =", "{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(i+1)

loss\_val.append(curr\_loss)

accur\_val.append(accuracy)

print("Optimization Finished!")

# Compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

# Compute train, validation and test accuracy

train\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:trainData})

train\_predict = (train\_predict\_soft >= 0.5).astype(int)

train\_correct = (train\_predict == trainTarget).astype(int)

train\_accuracy = np.sum(train\_correct) / train\_size \* 100

valid\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

valid\_predict = (valid\_predict\_soft >= 0.5).astype(int)

valid\_correct = (valid\_predict == validTarget).astype(int)

valid\_accuracy = np.sum(valid\_correct) # / 100 \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = (test\_predict\_soft >= 0.5).astype(int)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / test\_size \* 100

# Show the results

print("Final loss =", final\_loss)

print("train\_accuracy = {:.2f} per cent".format(train\_accuracy))

print("valid\_accuracy =", valid\_accuracy, "per cent")

print("test\_accuracy = {:.2f} per cent".format(test\_accuracy))

# Store the data for plotting

plot\_data[name+"\_epoch"] = epoch\_num

plot\_data[name+"\_loss"] = loss\_val

plot\_data[name+"\_accuracy"] = accur\_val

# Plot and show the graphs

for name in losses :

plt.plot(plot\_data[name+"\_epoch"],

plot\_data[name+"\_loss"], label=name+" loss")

plt.legend()

plt.show()

for name in losses :

plt.plot(plot\_data[name+"\_epoch"],

plot\_data[name+"\_accuracy"], label=name+" accuracy")

plt.legend()

plt.show()

**2.2.1**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx]/255, Target[randIndx]

trainData, trainTarget = Data[:15000], Target[:15000]

validData, validTarget = Data[15000:16000], Target[15000:16000]

testData, testTarget = Data[16000:], Target[16000:]

validData\_copy = validData

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

testData = np.asarray(testData)

testData = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))

# Initialize parameters

N\_train = 15000

N\_valid = 16000 - 15000

N\_test = 18720 - 16000

learning\_rates = [0.01, 0.003, 0.001, 0.0003]

batch\_size = 500

decay\_coef = 0.01

iteration = 22500 # 750 epoch, similar as previous problem

display\_step = 10

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.int32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 10]), name="W")

b = tf.Variable(tf.random\_uniform([1, 10]), name="b")

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

# tf.nn.softmax\_cross\_entropy\_with\_logits is out of date

cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(labels=tf.one\_hot(Y, 10, dtype=tf.float32), logits=Y\_hat)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = cross\_entropy\_loss + weight\_decay\_loss

# Iterate for different learning rates

for learning\_rate in learning\_rates :

optimizer = tf.train.AdamOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

accur\_val = []

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == N\_train :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], trainTarget[randIndx]

# Display logs per 10 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss and accuracy

curr\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = np.argmax(predict\_soft, axis=1)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct) / N\_valid \* 100

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(epoch\_counter)

loss\_val.append(curr\_loss)

accur\_val.append(accuracy)

# Update the training batch

if batch\_index + batch\_size > N\_train :

batch\_X, batch\_Y = trainData[batch\_index:], trainTarget[batch\_index:]

batch\_index = N\_train

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

# compute accuracy for validation and test data

valid\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

valid\_predict = np.argmax(valid\_predict\_soft, axis=1)

valid\_correct = (valid\_predict == validTarget).astype(int)

valid\_accuracy = np.sum(valid\_correct) / N\_valid \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = np.argmax(test\_predict\_soft, axis=1)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / N\_test \* 100

# Display a miss-predicted picture and the prediction

if learning\_rate == 0.01 :

valid\_one\_fail = np.argmin(valid\_correct)

plt.imshow(validData\_copy[valid\_one\_fail])

plt.show()

print("Ground truth is:", chr(65+validTarget[valid\_one\_fail]))

print("Miss-predicted as:", chr(65+valid\_predict[valid\_one\_fail]))

# Show the results

print("Final loss =", final\_loss)

print("Validation accuracy = {:.2f} per cent".format(valid\_accuracy))

print("Test accuracy = {:.2f} per cent".format(test\_accuracy))

# Plot and show the graphs

plt.plot(epoch\_num, loss\_val, label='Cross Entropy Loss')

plt.legend()

plt.show()

plt.plot(epoch\_num, accur\_val, label='Accuracy')

plt.legend()

plt.show()

**2.2.2**

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

# Loading data

def data\_segmentation(data\_path, target\_path, task):

# task = 0 >> select the name ID targets for face recognition task

# task = 1 >> select the gender ID targets for gender recognition task

data = np.load(data\_path)/255

data = np.reshape(data, [-1, 32\*32])

target = np.load(target\_path)

np.random.seed(45689)

rnd\_idx = np.arange(np.shape(data)[0])

np.random.shuffle(rnd\_idx)

trBatch = int(0.8\*len(rnd\_idx))

validBatch = int(0.1\*len(rnd\_idx))

trainData, validData, testData = data[rnd\_idx[1:trBatch],:], \

data[rnd\_idx[trBatch+1:trBatch + validBatch],:], \

data[rnd\_idx[trBatch + validBatch+1:-1],:]

trainTarget, validTarget, testTarget = target[rnd\_idx[1:trBatch], task], \

target[rnd\_idx[trBatch+1:trBatch + validBatch], task], \

target[rnd\_idx[trBatch + validBatch + 1:-1], task]

return trainData, validData, testData, trainTarget, validTarget, testTarget

trainData, validData, testData, trainTarget, validTarget, testTarget = \

data\_segmentation("data\_facescrub.npy", "target\_facescrub.npy", task = 0)

validData\_copy = np.reshape(validData, [-1, 32, 32])

trainTarget = np.transpose(trainTarget)

validTarget = np.transpose(validTarget)

testTarget = np.transpose(testTarget)

# Initialize parameters

N\_train = trainData.shape[0]

N\_valid = validData.shape[0]

N\_test = testData.shape[0]

learning\_rates = [0.01, 0.003, 0.001, 0.0003, 0.0001] # 0.003

batch\_size = 300

decay\_coefs = [0.1, 0.03, 0.01, 0.003, 0.001] # 0.003

iteration = 30000

display\_step = 100

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X")

Y = tf.placeholder(tf.int32, name="Y")

# Initialize w and b

W = tf.Variable(tf.random\_uniform([trainData.shape[1], 6]), name="W")

b = tf.Variable(tf.random\_uniform([1, 6]), name="b")

# Iterate for different weight decay coefficients

for decay\_coef in decay\_coefs :

# Iterate for different learning rates

for learning\_rate in learning\_rates :

# Compute the loss

Y\_hat = tf.add(tf.matmul(X, W), b)

# tf.nn.softmax\_cross\_entropy\_with\_logits is out of date

cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits\_v2( \

labels=tf.one\_hot(Y, 6, dtype=tf.float32), logits=Y\_hat)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

weight\_decay\_loss = decay\_coef \* tf.nn.l2\_loss(W)

loss = cross\_entropy\_loss + weight\_decay\_loss

optimizer = tf.train.AdamOptimizer(learning\_rate).minimize(loss)

# Initialize

init = tf.global\_variables\_initializer()

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init)

# Initialize counters for epochs

epoch\_counter = 0

batch\_index = 0

# Initialize plotting coordinates

epoch\_num = []

loss\_val = []

accur\_val = []

for i in range(iteration) :

# Update the training epoch if last epoch is finished

if batch\_index == N\_train :

# Update the counters

epoch\_counter += 1

batch\_index = 0

# Shuffle the training set

randIndx = np.arange(len(trainData))

np.random.shuffle(randIndx)

trainData, trainTarget = trainData[randIndx], \

trainTarget[randIndx]

# Display logs per 500 epochs

if epoch\_counter % display\_step ==0 :

# Compute current loss and accuracy

curr\_loss = sess.run(loss, feed\_dict={X:trainData,

Y:trainTarget})

predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

predict = np.argmax(predict\_soft, axis=1)

correct = (predict == validTarget).astype(int)

accuracy = np.sum(correct) / N\_valid \* 100

print("Epoch =", epoch\_counter, "loss =",

"{:.9f}".format(curr\_loss))

# Update the coordinates

epoch\_num.append(epoch\_counter)

loss\_val.append(curr\_loss)

accur\_val.append(accuracy)

# Update the training batch

if batch\_index + batch\_size > N\_train :

batch\_X, batch\_Y = trainData[batch\_index:],\

trainTarget[batch\_index:]

batch\_index = N\_train

else :

batch\_X, batch\_Y = trainData[batch\_index:batch\_index + batch\_size], \

trainTarget[batch\_index:batch\_index + batch\_size]

batch\_index += batch\_size

# Run the optimizer

sess.run(optimizer, feed\_dict={X:batch\_X, Y:batch\_Y})

print("Optimization Finished!")

# compute final loss

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

# compute accuracy for validation and test data

valid\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

valid\_predict = np.argmax(valid\_predict\_soft, axis=1)

valid\_correct = (valid\_predict == validTarget).astype(int)

valid\_accuracy = np.sum(valid\_correct) / N\_valid \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = np.argmax(test\_predict\_soft, axis=1)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / N\_test \* 100

# When learning\_rate is 0.003 and decay\_coef is 0.003,

# plot the graphs as required.

# Also display a miss-predicted picture and the prediction.

if learning\_rate == 0.003 and decay\_coef == 0.003 :

# Plot and show the graphs

plt.plot(epoch\_num, loss\_val, label='Cross Entropy Loss for lambda='+\

str(decay\_coef)+', learning rate='+str(learning\_rate))

plt.legend()

plt.show()

plt.plot(epoch\_num, accur\_val, label='Accuracy for lambda='+ \

str(decay\_coef)+', learning rate='+str(learning\_rate))

plt.legend()

plt.show()

valid\_one\_fail = np.argmin(valid\_correct)

plt.imshow(validData\_copy[valid\_one\_fail])

plt.show()

print("Ground truth is:", validTarget[valid\_one\_fail])

print("Miss-predicted as:", valid\_predict[valid\_one\_fail])

# Show the results

print("Final loss =", final\_loss)

print("Validation accuracy = {:.2f} per cent".format(valid\_accuracy))

print("Test accuracy = {:.2f} per cent".format(test\_accuracy))

**normal\_equation**

import numpy as np

import tensorflow as tf

import time

# Loading data

with np.load("notMNIST.npz") as data :

Data, Target = data["images"], data["labels"]

posClass = 2

negClass = 9

dataIndx = (Target==posClass) + (Target==negClass)

Data = Data[dataIndx]/255

Target = Target[dataIndx].reshape(-1,1)

Target[Target==posClass] = 1

Target[Target==negClass] = 0

np.random.seed(521)

randIndx = np.arange(len(Data))

np.random.shuffle(randIndx)

Data, Target = Data[randIndx], Target[randIndx]

trainData, trainTarget = Data[:3500], Target[:3500]

validData, validTarget = Data[3500:3600], Target[3500:3600]

testData, testTarget = Data[3600:], Target[3600:]

trainData = np.asarray(trainData)

trainData = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))

validData = np.asarray(validData)

validData = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))

X= np.hstack((trainData, np.ones((3500,1))))

X=X.astype('float32')

validData= np.hstack((validData, np.ones((100,1))))

# Initialize parameters

decay\_coefs = 0

train\_size = 3500

test\_size = 145

X= np.hstack((trainData, np.ones((3500,1))))

X\_Trans=tf.tofloat(tf.transpose(X))

XX = tf.matmul(X\_Trans, X.astype)

XX\_inv = tf.matrix\_inverse(XX)

product = tf.matmul(XX\_inv, tf.transpose(X))

w = tf.matmul(product,trainTarget)

# Initialize placeholders

data = tf.placeholder(tf.float32, name="data")

Y = tf.placeholder(tf.float32, name="Y")

# Compute the loss

Y\_hat = tf.matmul(data, w)

MSE\_loss = 0.5\*tf.losses.mean\_squared\_error(Y\_hat,Y )

weight\_decay\_loss = decay\_coefs \* tf.nn.l2\_loss(w)

loss = MSE\_loss + weight\_decay\_loss

with tf.Session() as sess:

# Start the timer before the Initialization of the optimizer

start\_time = time.time()

final\_loss = sess.run(loss, feed\_dict={X:trainData, Y:trainTarget})

end\_time = time.time()

# Compute train, validation and test accuracy

train\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:trainData})

train\_predict = (train\_predict\_soft >= 0.5).astype(int)

train\_correct = (train\_predict == trainTarget).astype(int)

train\_accuracy = np.sum(train\_correct) / train\_size \* 100

valid\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:validData})

valid\_predict = (valid\_predict\_soft >= 0.5).astype(int)

valid\_correct = (valid\_predict == validTarget).astype(int)

valid\_accuracy = np.sum(valid\_correct) # / 100 \* 100

test\_predict\_soft = sess.run(Y\_hat, feed\_dict={X:testData})

test\_predict = (test\_predict\_soft >= 0.5).astype(int)

test\_correct = (test\_predict == testTarget).astype(int)

test\_accuracy = np.sum(test\_correct) / test\_size \* 100

# Show the results

print("Final loss =", final\_loss)

print("train\_accuracy = {:.2f} per cent".format(train\_accuracy))

print("valid\_accuracy =", valid\_accuracy, "per cent")

print("test\_accuracy = {:.2f} per cent".format(test\_accuracy))

print("Time spent =", end\_time-start\_time)