**ECE1504: Assignment 2 Neural Networks**

Due: 26 Nov 2018

Tianyi Xie 1000570679

Yisi Zou 1001826321

Contribution percentage is the same for both group members

Part 1

1.2.1 Neural Network Architecture [20 pt.]

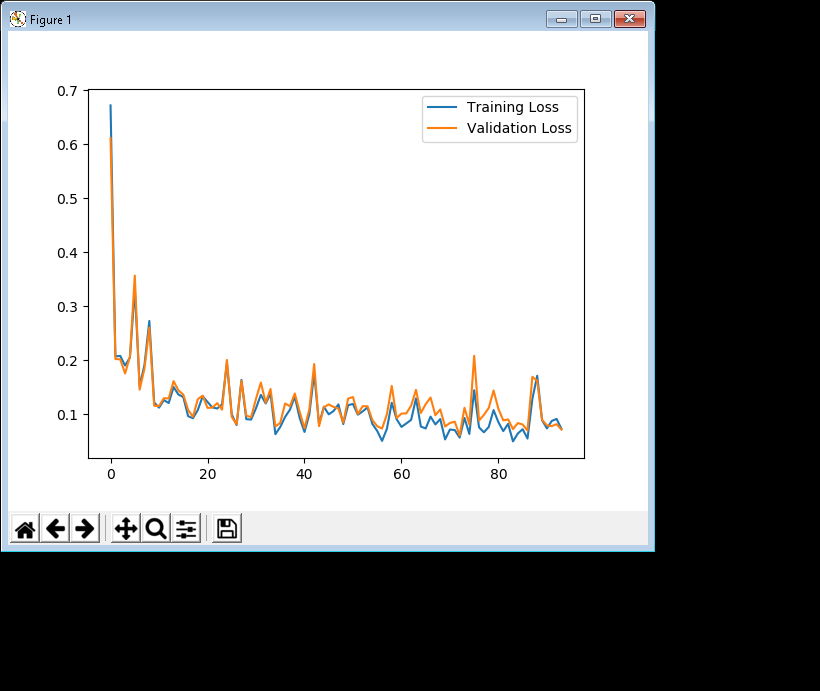
See the code in appendix 1.2.1-2

1.2.2 Training [15 pt.]

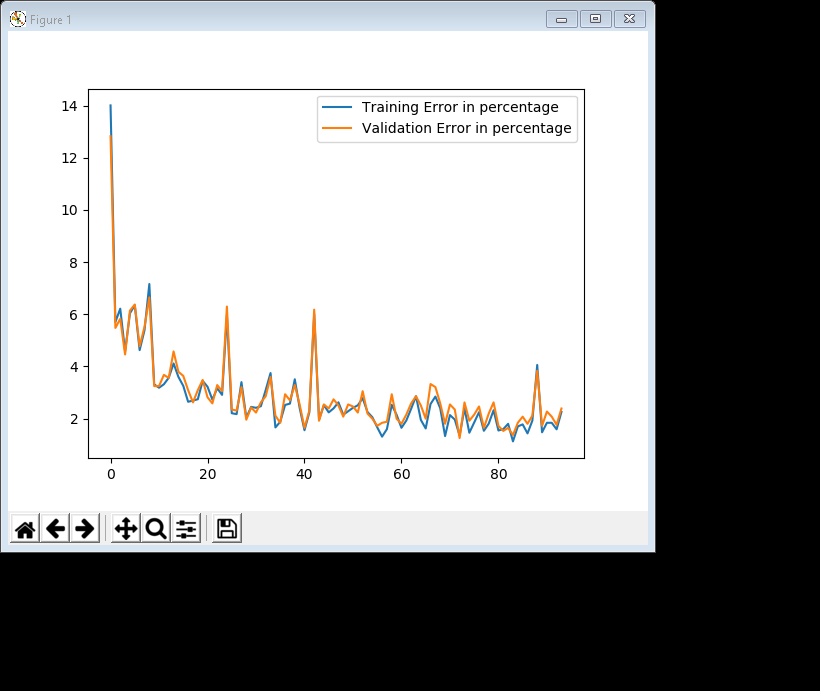
See the code in appendix 1.2.1-2

* Batch size=20
* Number of epochs=1000
* Learning rate=0.005
* Maximum number of epochs without progress=20

Here is the training and validation loss function vs. the number of epochs

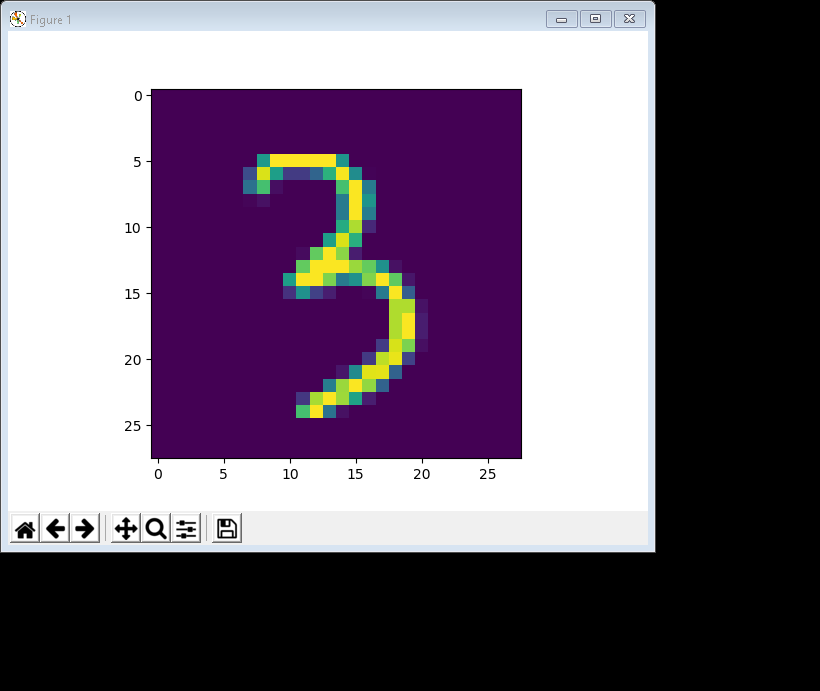


Here is the training and validation classification error vs. the number of epochs



The precision we can achieve over the test set is 98.70%. It takes 73 iterations to converge.

Here is one sample from the test set for which the estimated label is not the same as its true label.



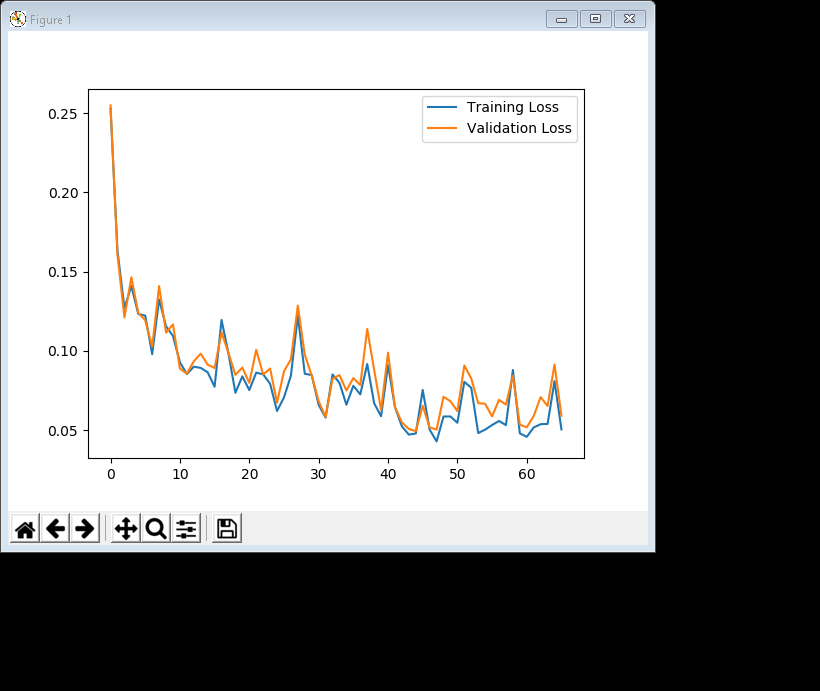
The estimated label is 4 while the true label is 3.

1.2.3 Tuning Hyperparameters [5 pt.]

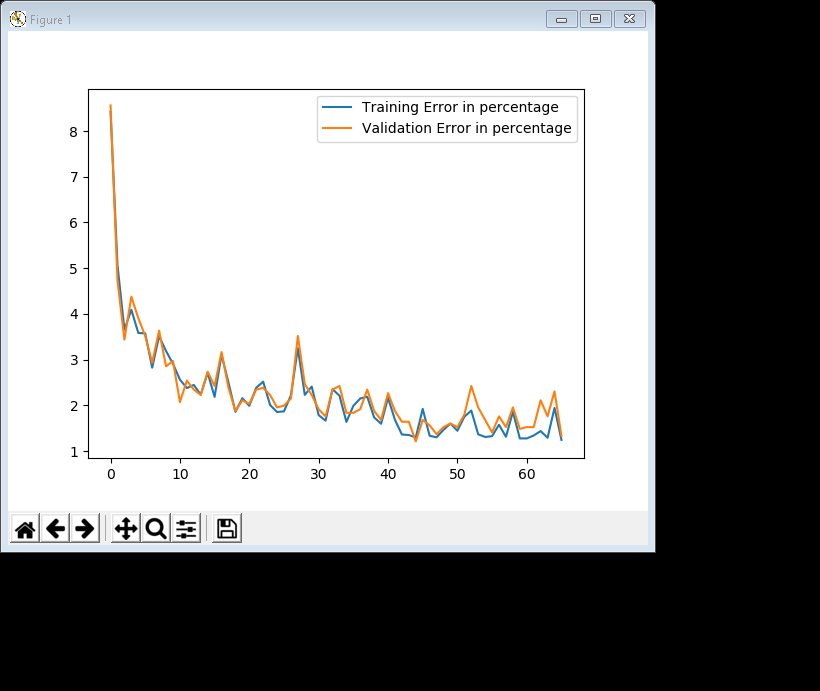
1) See the code in appendix 1.2.3.1

* Batch size=20
* Number of epochs=1000
* Number of neurons per layer=50
* Learning rate=0.005
* Activation function ReLU
* Maximum number of epochs without progress=20

Here is the training and validation loss function vs. the number of epochs



Here is the training and validation classification error vs. the number of epochs

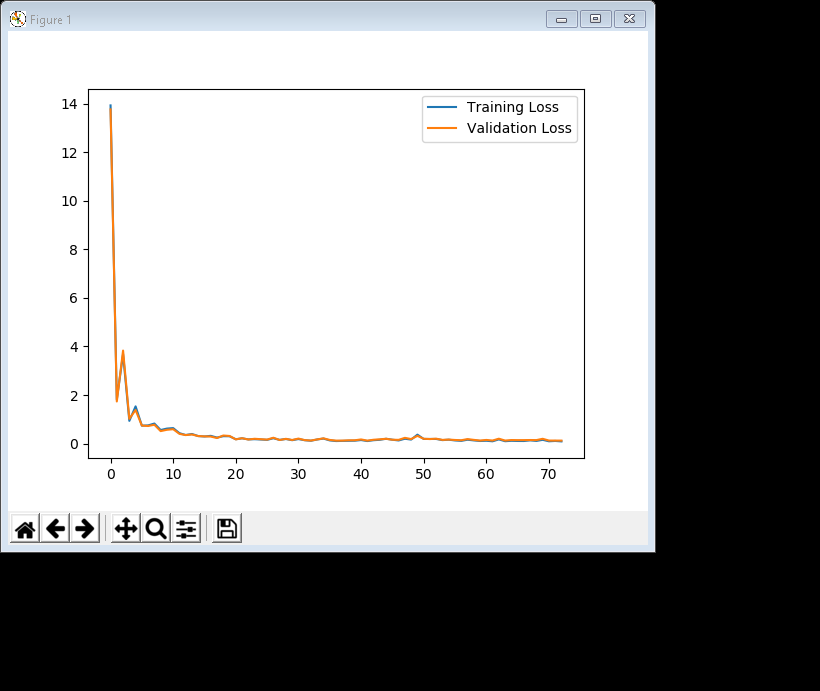


The precision we can achieve over the test set is 98.11%. It takes 68 iterations to converge.

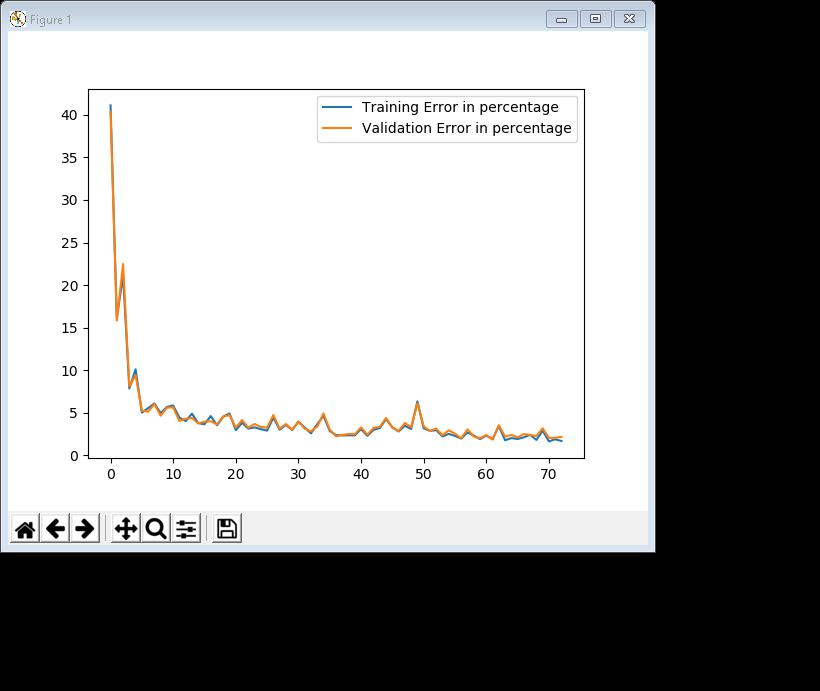
2) See the code in appendix 1.2.3.2

* Batch size=500
* Number of epochs=1000
* Number of neurons per layer=140
* Learning rate=0.1
* Activation function Leaky ReLU with alpha=0.1
* Maximum number of epochs without progress=30

Here is the training and validation loss function vs. the number of epochs



Here is the training and validation classification error vs. the number of epochs



The precision we can achieve over the test set is 97.86%. It takes 72 iterations to converge.

1.2.4 Batch Normalization [10 pt.]

See the code in appendix 1.2.4

Momentum can help to reduce the noise in gradient update term and converges faster to optimal value. It helps SGD to navigate along the relevant directions and soften the oscillations in irrelevant. It adds a friction of the direction of the precious step to current step. This achieves the amplification of speed in the current direction and soften the oscillation in the run direction along the relevant directions.

The way of updating mean and variance using momentum is as follow:

running\_mean = momentum \* running\_mean + (1 - momentum) \* sample\_mean  
running\_var = momentum \* running\_var + (1 - momentum) \* sample\_var

Here are the comparisons of performance in terms of cross entropy and precision on the training set and validation set

|  |  |
| --- | --- |
|  |  |
|  |  |

As we can see from the plots and the table below, both training and validation precision increase as momentums increase. Similarly, the cross entropy losses of both training and validation set decrease as momentums increase. Test accuracies are also reported in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Momentum | Train precision | Train loss | Valid precision | Valid loss | Test accuracy |
| 0.85 | 98.23% | 0.062747 | 98.36% | 0.055147 | 98.38% |
| 0.9 | 98.67% | 0.047420 | 98.55% | 0.041708 | 98.79% |
| 0.95 | 98.97% | 0.038702 | 99.02% | 0.039354 | 99.05% |
| 0.99 | 99.06% | 0.031642 | 98.87% | 0.030816 | 99.14% |

1.2.5 Dropout [5 pt.]

Here are the comparisons of performance in terms of cross entropy and precision on the training set and validation set, between the models in 1.2.2 and 1.2.4. For the part using the model in 1.2.4, we choose momentum = 0.99, as it has the highest test accuracy.

|  |  |
| --- | --- |
| 1.2.2 | 1.2.4 |
|  |  |
|  |  |

As we can see from the plots and the table below, both training and validation precision decrease as dropout rates increase in both models. Similarly, the cross entropy losses of both training and validation set increase as dropout rates increase in both models. Test accuracies are also reported in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model-dropout | Train precision | Train loss | Valid precision | Valid loss | Test accuracy |
| 1.2.2-0.1 | 97.79% | 0.090755 | 97.97% | 0.083478 | 98.15% |
| 1.2.2-0.3 | 97.48% | 0.091208 | 97.19% | 0.099475 | 97.80% |
| 1.2.4-0.1 | 99.11% | 0.030980 | 98.71% | 0.039979 | 99.22% |
| 1.2.4-0.3 | 98.68% | 0.046515 | 98.67% | 0.044631 | 98.60% |

1.2.6 Impact of Regularization Methods [5 pt.]

Among all three models, batch normalization + dropout has the best outcome. It seems that while training the dropout model, we add too much noise in the training parameters. Adding batch normalization on top of dropout can cancel out the noise and get better results.

Part 2 transfer Learning [40 pt.]

2.2.1 Reusing the model [10 pt.]

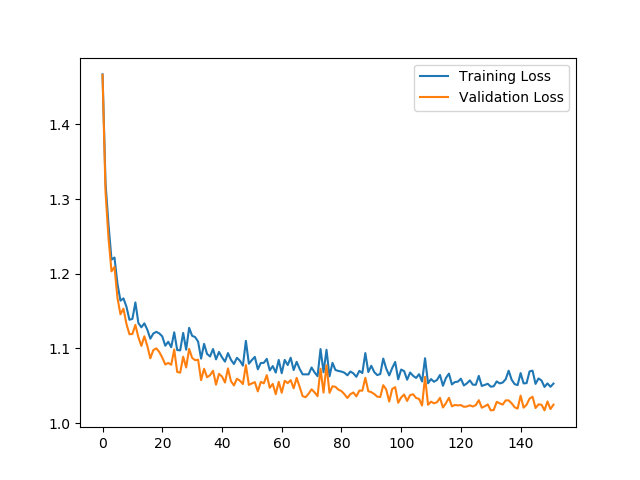
See the code in appendix 2.2.1-2

2.2.2 Training [10 pt.]

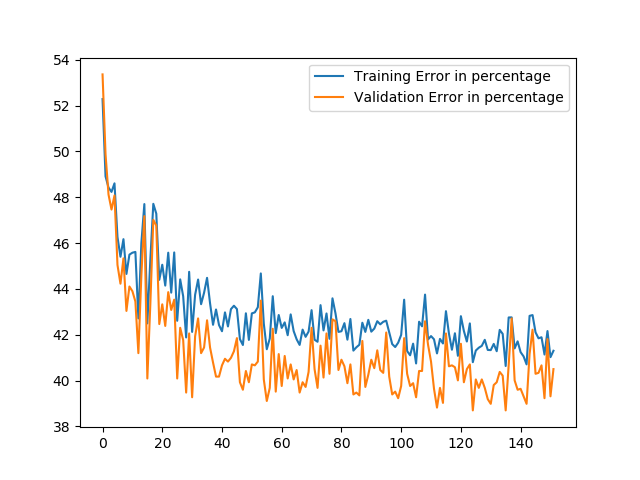
See the code in appendix 2.2.1-2

For the full training set:

Here is the training and validation loss function vs. the number of epochs



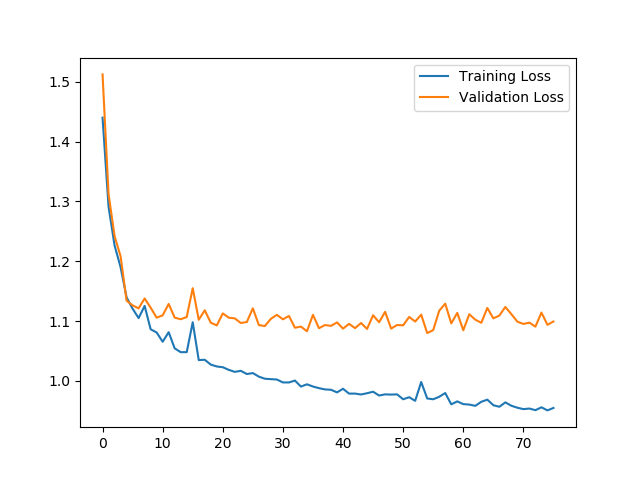
Here is the training and validation classification error vs. the number of epochs



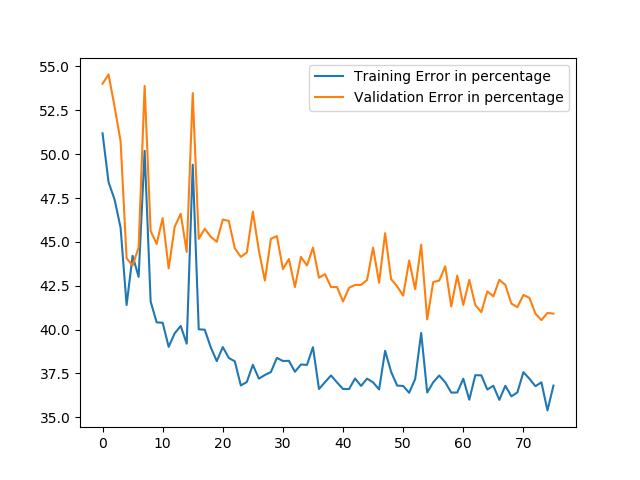
The precision we can achieve over the test set is 58.65%. It takes 151 iterations to converge.

For 100 images per digit:

Here is the training and validation classification error vs. the number of epochs



Here is the training and validation loss function vs. the number of epochs



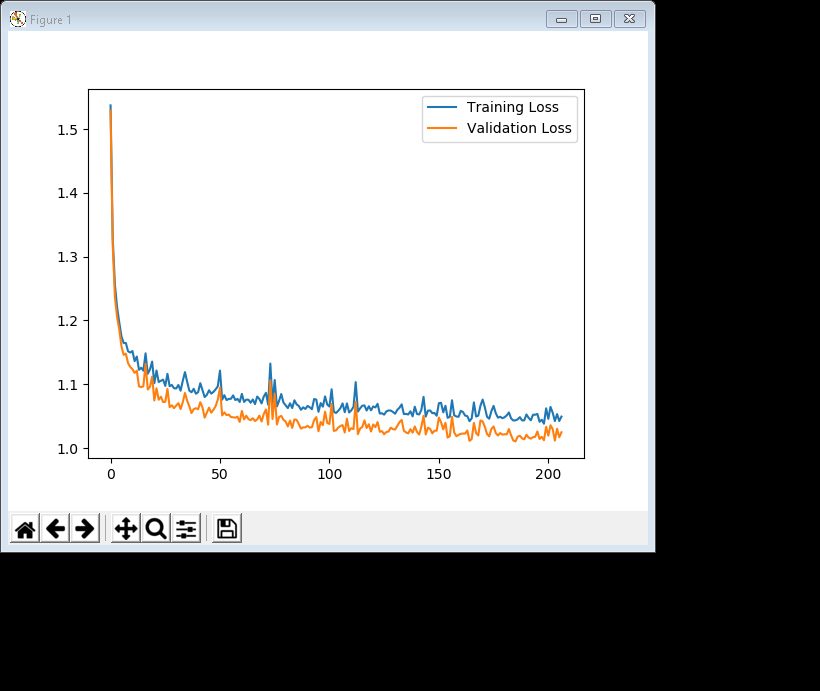
The precision we can achieve over the test set is 55.50%. It takes 75 iterations to converge.

Comparing to the case that we train on the entire training set, training using 100 images per digits converges faster, but the accuracy is lower.

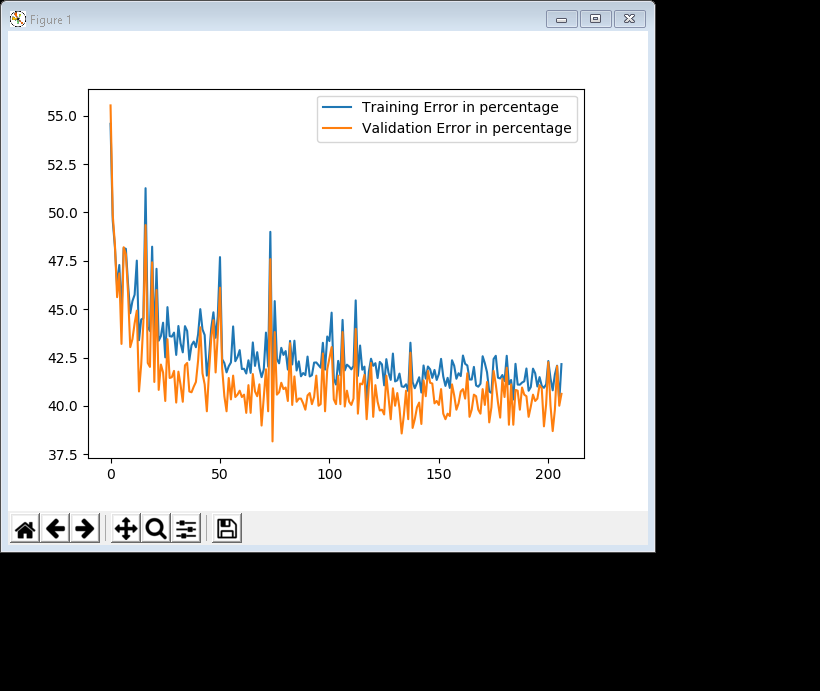
2.2.3 Removing the last hidden layer [10 pt.]

See the code in appendix 2.2.3

Here is the training and validation classification error vs. the number of epochs



Here is the training and validation loss function vs. the number of epochs



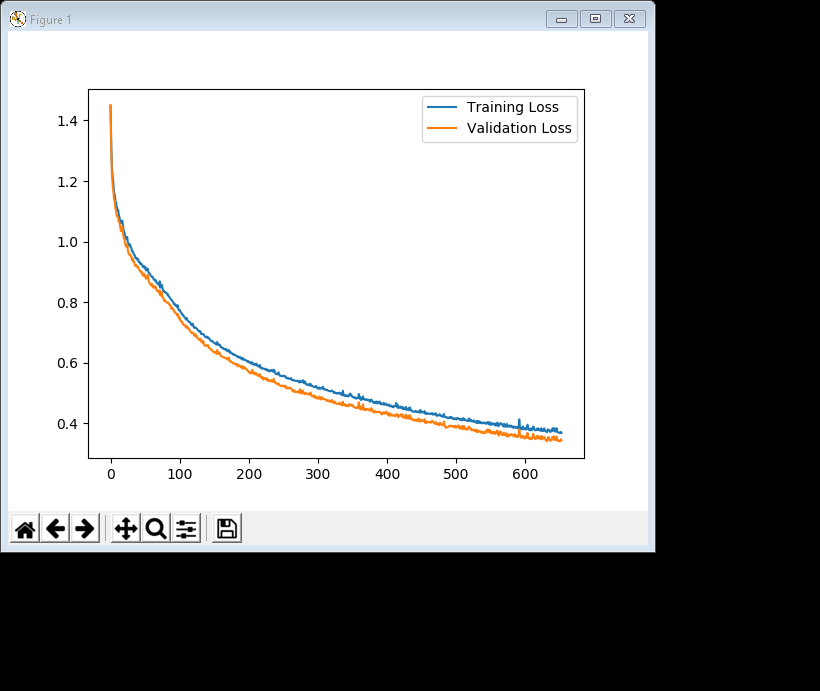
The precision we can achieve over the test set is 58.86%. It takes 206 iterations to converge.

According to our results, removing the last layer eventually resulted in better accuracy. This is because high-level features in a different model is not helpful for a new task.

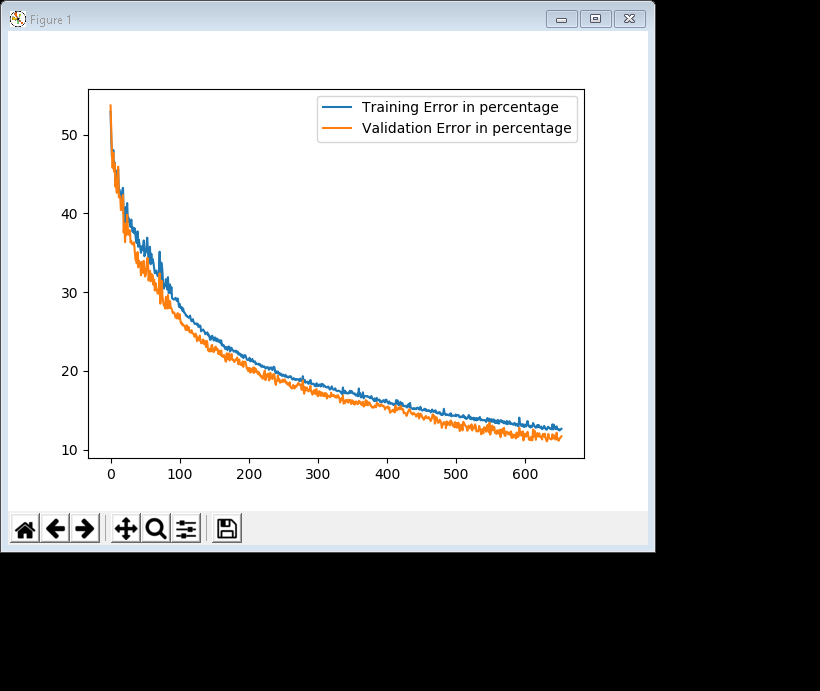
2.2.4 Unfreezing hidden layers 3 and 4 [10 pt.]

See the code in appendix 2.2.4

Here is the training and validation classification error vs. the number of epochs



Here is the training and validation loss function vs. the number of epochs



The precision we can achieve over the test set is 86.44%. It takes 652 iterations to converge

According to our results, after unfreezing the third and fourth layer, the accuracy is much higher than before. Again this is because high-level features in a different model is not helpful for a new task.

APPENDIX

PART1

1.2.1-2

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model0.ckpt"

node\_num = 100

output\_num = 5

learning\_rate = 0.005

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64, name="batch\_size")

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next(name="batch\_xy")

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H1")

H2 = tf.layers.dense(H1, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H2")

H3 = tf.layers.dense(H2, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H3")

H4 = tf.layers.dense(H3, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H4")

H5 = tf.layers.dense(H4, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H5")

Y\_hat = tf.layers.dense(H5, output\_num, name="Y\_hat")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

loss = tf.reduce\_mean(cross\_entropy, name="loss")

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct, name="accuracy")

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize saver

saver = tf.train.Saver()

# Save the dataset initialization operation with name

dataset\_init\_op = iter.make\_initializer(dataset, name='dataset\_init')

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(dataset\_init\_op, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(dataset\_init\_op, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc, test\_correct, test\_predict = sess.run([accuracy, correct, predict])

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

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

plt.imshow(X\_test1\_pic[test\_one\_fail])

plt.show()

print("Ground truth is:", y\_test1[test\_one\_fail])

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

1.2.3.1

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model1.ckpt"

node\_num = 50

output\_num = 5

learning\_rate = 0.005

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64)

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train1.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next()

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, activation=tf.nn.relu,

kernel\_initializer=He\_init, name="H1")

H2 = tf.layers.dense(H1, node\_num, activation=tf.nn.relu,

kernel\_initializer=He\_init, name="H2")

H3 = tf.layers.dense(H2, node\_num, activation=tf.nn.relu,

kernel\_initializer=He\_init, name="H3")

H4 = tf.layers.dense(H3, node\_num, activation=tf.nn.relu,

kernel\_initializer=He\_init, name="H4")

H5 = tf.layers.dense(H4, node\_num, activation=tf.nn.relu,

kernel\_initializer=He\_init, name="H5")

Y\_hat = tf.layers.dense(H5, output\_num, name="outputs")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

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

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct)

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize saver

saver = tf.train.Saver()

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(iter.initializer, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(iter.initializer, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

1.2.3.2

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model2.ckpt"

node\_num = 140

output\_num = 5

learning\_rate = 0.1

epoch\_num = 1000

train\_batch\_size = 500

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 30

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64)

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train1.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next()

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, kernel\_initializer=He\_init, name="H1")

H1\_out = tf.nn.leaky\_relu(H1, alpha=0.1)

H2 = tf.layers.dense(H1\_out, node\_num, kernel\_initializer=He\_init, name="H2")

H2\_out = tf.nn.leaky\_relu(H1, alpha=0.1)

H3 = tf.layers.dense(H2\_out, node\_num, kernel\_initializer=He\_init, name="H3")

H3\_out = tf.nn.leaky\_relu(H1, alpha=0.1)

H4 = tf.layers.dense(H3\_out, node\_num, kernel\_initializer=He\_init, name="H4")

H4\_out = tf.nn.leaky\_relu(H1, alpha=0.1)

H5 = tf.layers.dense(H4\_out, node\_num, kernel\_initializer=He\_init, name="H5")

H5\_out = tf.nn.leaky\_relu(H1, alpha=0.1)

Y\_hat = tf.layers.dense(H5\_out, output\_num, name="outputs")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

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

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct)

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize saver

saver = tf.train.Saver()

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(iter.initializer, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(iter.initializer, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

1.2.4

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model\_bn.ckpt"

node\_num = 100

output\_num = 5

learning\_rate = 0.005

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

momentums = [0.85, 0.9, 0.95, 0.99]

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64)

training = tf.placeholder\_with\_default(False, shape=(), name='training')

momentum = tf.placeholder(tf.float32)

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train1.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next()

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H1")

BN1 = tf.layers.batch\_normalization(H1, training=training, momentum=momentum)

H2 = tf.layers.dense(BN1, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H2")

BN2 = tf.layers.batch\_normalization(H2, training=training, momentum=momentum)

H3 = tf.layers.dense(BN2, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H3")

BN3 = tf.layers.batch\_normalization(H3, training=training, momentum=momentum)

H4 = tf.layers.dense(BN3, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H4")

BN4 = tf.layers.batch\_normalization(H4, training=training, momentum=momentum)

H5 = tf.layers.dense(BN4, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H5")

BN5 = tf.layers.batch\_normalization(H5, training=training, momentum=momentum)

Y\_hat = tf.layers.dense(BN5, output\_num, name="outputs")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

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

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct)

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize optimizer for batch normalization

BN\_optimizer = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

# Initialize saver

saver = tf.train.Saver()

# Iterate for different momentums

for curr\_momentum in momentums :

# Initialize parameters for early stopping

check\_count = 0

best\_loss = np.infty

best\_acc = 0

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_acc\_axis = []

valid\_loss\_axis = []

valid\_acc\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run([optimizer, BN\_optimizer],

feed\_dict={training: True, momentum:curr\_momentum})

# Compute current loss and accuracy

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy],

feed\_dict={momentum:curr\_momentum})

epoch\_acc = epoch\_acc \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"precision =", "{:.2f}".format(epoch\_acc), "per cent")

# Calculate validation loss and accuracy for this epoch

sess.run(iter.initializer, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy],

feed\_dict={momentum:curr\_momentum})

valid\_acc = valid\_acc \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_acc\_axis.append(epoch\_acc)

valid\_loss\_axis.append(valid\_loss)

valid\_acc\_axis.append(valid\_acc)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_acc = valid\_acc

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(iter.initializer, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc, test\_correct, test\_predict = sess.run([accuracy, correct,

predict], feed\_dict={momentum:curr\_momentum})

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation acc =", "{:.2f}".format(best\_acc), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss, momentum =' + str(curr\_momentum))

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss, momentum =' + str(curr\_momentum))

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_acc\_axis, label='Training Precision in percentage, momentum =' + str(curr\_momentum))

plt.plot(epoch\_axis, valid\_acc\_axis, label='Validation Precision in percentage, momentum =' + str(curr\_momentum))

plt.legend()

plt.show()

1.2.5.1

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model\_dp.ckpt"

node\_num = 100

output\_num = 5

learning\_rate = 0.005

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

dropout\_rates = [0.1, 0.3]

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64)

training=tf.placeholder\_with\_default(False, shape=(), name='training')

drop\_rate = tf.placeholder(tf.float32)

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train1.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next()

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H1")

H1\_drop = tf.layers.dropout(H1, drop\_rate, training=training)

H2 = tf.layers.dense(H1\_drop, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H2")

H2\_drop = tf.layers.dropout(H2, drop\_rate, training=training)

H3 = tf.layers.dense(H2\_drop, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H3")

H3\_drop = tf.layers.dropout(H3, drop\_rate, training=training)

H4 = tf.layers.dense(H3\_drop, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H4")

H4\_drop = tf.layers.dropout(H4, drop\_rate, training=training)

H5 = tf.layers.dense(H4\_drop, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H5")

H5\_drop = tf.layers.dropout(H5, drop\_rate, training=training)

Y\_hat = tf.layers.dense(H5\_drop, output\_num, name="outputs")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

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

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct)

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize saver

saver = tf.train.Saver()

# Iterate for different dropout rates

for dropout\_rate in dropout\_rates :

# Initialize parameters for early stopping

check\_count = 0

best\_loss = np.infty

best\_acc = 0

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_acc\_axis = []

valid\_loss\_axis = []

valid\_acc\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer, feed\_dict={training:True,

drop\_rate:dropout\_rate})

# Compute current loss and accuracy

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy],

feed\_dict={drop\_rate:0.0}) # kill the dropout for testing

epoch\_acc = epoch\_acc \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"accuracy =", "{:.2f}".format(epoch\_acc), "per cent")

# Calculate validation loss and accuracy for this epoch

sess.run(iter.initializer, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy],

feed\_dict={drop\_rate:0.0}) # kill the dropout for testing

valid\_acc = valid\_acc \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_acc\_axis.append(epoch\_acc)

valid\_loss\_axis.append(valid\_loss)

valid\_acc\_axis.append(valid\_acc)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_acc = valid\_acc

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(iter.initializer, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc, test\_correct, test\_predict = sess.run([accuracy, correct,

predict], feed\_dict={drop\_rate:0.0}) # kill the dropout for testing

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation precision =", "{:.2f}".format(best\_acc), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss, dropout rate=' + str(dropout\_rate))

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss, dropout rate=' + str(dropout\_rate))

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_acc\_axis,

label='Training Precision in percentage, dropout rate=' + str(dropout\_rate))

plt.plot(epoch\_axis, valid\_acc\_axis,

label='Validation Precision in percentage, dropout rate=' + str(dropout\_rate))

plt.legend()

plt.show()

1.2.5.2

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test = X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

# Initialize parameters

path = "./model\_dpbn.ckpt"

node\_num = 100

output\_num = 5

learning\_rate = 0.005

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

best\_momentum = 0.85

dropout\_rates = [0.1, 0.3]

# Modify data sets

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

X\_test1\_pic = np.reshape(X\_test1, [-1, 28, 28])

# Initialize placeholders

X = tf.placeholder(tf.float32, name="X", shape=[None,28\*28])

Y = tf.placeholder(tf.int64, name="Y", shape=[None,])

batch\_size = tf.placeholder(tf.int64)

training = tf.placeholder\_with\_default(False, shape=(), name='training')

momentum = tf.placeholder(tf.float32)

drop\_rate = tf.placeholder(tf.float32)

# Initialize the data set

dataset = tf.data.Dataset.from\_tensor\_slices((X, Y))

# Shuffle, repeat, and batch the examples.

dataset = dataset.shuffle(X\_train1.shape[0]).repeat().batch(batch\_size)

# Create iterator

iter = dataset.make\_initializable\_iterator()

batch\_X, batch\_Y = iter.get\_next()

# Construct a neural network

H1 = tf.layers.dense(batch\_X, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H1")

H1\_drop = tf.layers.dropout(H1, drop\_rate, training=training)

BN1 = tf.layers.batch\_normalization(H1\_drop, training=training, momentum=momentum)

H2 = tf.layers.dense(BN1, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H2")

H2\_drop = tf.layers.dropout(H2, drop\_rate, training=training)

BN2 = tf.layers.batch\_normalization(H2\_drop, training=training, momentum=momentum)

H3 = tf.layers.dense(BN2, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H3")

H3\_drop = tf.layers.dropout(H3, drop\_rate, training=training)

BN3 = tf.layers.batch\_normalization(H3\_drop, training=training, momentum=momentum)

H4 = tf.layers.dense(BN3, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H4")

H4\_drop = tf.layers.dropout(H4, drop\_rate, training=training)

BN4 = tf.layers.batch\_normalization(H4\_drop, training=training, momentum=momentum)

H5 = tf.layers.dense(BN4, node\_num, activation=tf.nn.elu,

kernel\_initializer=He\_init, name="H5")

H5\_drop = tf.layers.dropout(H5, drop\_rate, training=training)

BN5 = tf.layers.batch\_normalization(H5\_drop, training=training, momentum=momentum)

Y\_hat = tf.layers.dense(BN5, output\_num, name="outputs")

# Calculate the loss

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

labels=tf.one\_hot(batch\_Y, 5, dtype=tf.float32), logits=Y\_hat)

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

# Calculate the accuracy

predict = tf.argmax(Y\_hat, axis=1)

correct = tf.cast(tf.equal(batch\_Y, predict), tf.float32)

accuracy = tf.reduce\_mean(correct)

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

# Initialize tensorflow

init\_g = tf.global\_variables\_initializer()

init\_l = tf.local\_variables\_initializer()

# Initialize optimizer for batch normalization

BN\_optimizer = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

# Initialize saver

saver = tf.train.Saver()

# Iterate for different momentums

for dropout\_rate in dropout\_rates :

# Initialize parameters for early stopping

check\_count = 0

best\_loss = np.infty

best\_acc = 0

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_acc\_axis = []

valid\_loss\_axis = []

valid\_acc\_axis = []

# Start training

with tf.Session() as sess :

# Run the initializer

sess.run(init\_g)

sess.run(init\_l)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run([optimizer, BN\_optimizer], feed\_dict={training:True,

drop\_rate:dropout\_rate, momentum:best\_momentum})

# Compute current loss and accuracy

sess.run(iter.initializer, feed\_dict={X:X\_train1, Y:y\_train1,

batch\_size:X\_train1.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy],

feed\_dict={drop\_rate:0.0, momentum:best\_momentum}) # kill the dropout for testing

epoch\_acc = epoch\_acc \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"precision =", "{:.2f}".format(epoch\_acc), "per cent")

# Calculate validation loss and accuracy for this epoch

sess.run(iter.initializer, feed\_dict={X:X\_valid1, Y:y\_valid1,

batch\_size:X\_valid1.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy],

feed\_dict={drop\_rate:0.0, momentum:best\_momentum}) # kill the dropout for testing

valid\_acc = valid\_acc \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_acc\_axis.append(epoch\_acc)

valid\_loss\_axis.append(valid\_loss)

valid\_acc\_axis.append(valid\_acc)

if valid\_loss < best\_loss :

save\_path = saver.save(sess, path)

best\_loss = valid\_loss

best\_acc = valid\_acc

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

saver.restore(sess, path)

sess.run(iter.initializer, feed\_dict={X:X\_test1, Y:y\_test1,

batch\_size:X\_test1.shape[0]})

test\_acc, test\_correct, test\_predict = sess.run([accuracy, correct,

predict], feed\_dict={drop\_rate:0.0, momentum:best\_momentum}) # kill the dropout for testing

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation acc =", "{:.2f}".format(best\_acc), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss, dropout rate=' + str(dropout\_rate))

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss, dropout rate=' + str(dropout\_rate))

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_acc\_axis,

label='Training Precision in percentage, dropout rate=' + str(dropout\_rate))

plt.plot(epoch\_axis, valid\_acc\_axis,

label='Validation Precision in percentage, dropout rate=' + str(dropout\_rate))

plt.legend()

plt.show()

PART2

2.2.1-2

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import time

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test)=tf.keras.datasets.mnist.load\_data()

X\_train=X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test=X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train=y\_train.astype(np.int32)

y\_test=y\_test.astype(np.int32)

X\_valid, X\_train=X\_train[:5000], X\_train[5000:]

y\_valid, y\_train=y\_train[:5000], y\_train[5000:]

X\_train2\_full=X\_train[y\_train>=5]

y\_train2\_full=y\_train[y\_train>=5]-5

X\_valid2\_full=X\_valid[y\_valid>=5]

y\_valid2\_full=y\_valid[y\_valid>=5]-5

X\_test2=X\_test[y\_test>=5]

y\_test2=y\_test[y\_test>=5]-5

# Initialize parameters

path = "./model0.ckpt"

new\_path = "./transfer0.ckpt"

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

# Import graph

saver = tf.train.import\_meta\_graph(path+".meta")

# Initialize new saver

new\_saver = tf.train.Saver()

# Restore placeholders

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

Y = tf.get\_default\_graph().get\_tensor\_by\_name("Y:0")

batch\_size = tf.get\_default\_graph().get\_tensor\_by\_name("batch\_size:0")

learning\_rate = tf.get\_default\_graph().get\_tensor\_by\_name("Adam/learning\_rate:0")

loss = tf.get\_default\_graph().get\_tensor\_by\_name("loss:0")

accuracy = tf.get\_default\_graph().get\_tensor\_by\_name("accuracy:0")

graph = tf.get\_default\_graph()

# Restore the dataset initialization operation

dataset\_init\_op = graph.get\_operation\_by\_name('dataset\_init')

# Freeze lower layers

with tf.name\_scope("train"):

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

train\_vars=tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES,

scope="Y\_hat")

optimizer = optimizer.minimize(loss, var\_list=train\_vars)

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Start training

with tf.Session() as sess :

# Restore the session

saver.restore(sess, path)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:X\_train2\_full.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_valid2\_full, Y:y\_valid2\_full,

batch\_size:X\_valid2\_full.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = new\_saver.save(sess, new\_path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

new\_saver.restore(sess, new\_path)

sess.run(dataset\_init\_op, feed\_dict={X:X\_test2, Y:y\_test2,

batch\_size:X\_test2.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

################################################################################

# Reduce the number of samples per digits

def sample\_n\_instances\_per\_class(X, y, n) :

Xs, ys = [], []

for label in np.unique(y):

idx = (y == label)

Xc = X[idx][:n]

yc = y[idx][:n]

Xs.append(Xc)

ys.append(yc)

return np.concatenate(Xs), np.concatenate(ys)

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

check\_count = 0

best\_loss = np.infty

best\_error = 0

new\_path = "./transfer0\_1.ckpt"

# Start training

with tf.Session() as sess :

# Restore the session

saver.restore(sess, path)

for epoch in range(epoch\_num) :

# train for one epoch

feed\_X, feed\_y = sample\_n\_instances\_per\_class(X\_train2\_full,

y\_train2\_full, 100)

sess.run(dataset\_init\_op, feed\_dict={X:feed\_X, Y:feed\_y,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

feed\_X, feed\_y = sample\_n\_instances\_per\_class(X\_train2\_full,

y\_train2\_full, 100)

sess.run(dataset\_init\_op, feed\_dict={X:feed\_X, Y:feed\_y,

batch\_size:X\_train2\_full.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

feed\_X, feed\_y = sample\_n\_instances\_per\_class(X\_valid2\_full,

y\_valid2\_full, 100)

sess.run(dataset\_init\_op, feed\_dict={X:feed\_X, Y:feed\_y,

batch\_size:X\_valid2\_full.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = new\_saver.save(sess, new\_path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

new\_saver.restore(sess, new\_path)

feed\_X, feed\_y = sample\_n\_instances\_per\_class(X\_test2,

y\_test2, 100)

sess.run(dataset\_init\_op, feed\_dict={X:feed\_X, Y:feed\_y,

batch\_size:X\_test2.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

2.2.3

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import time

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

reset\_graph()

# Load data sets

(X\_train, y\_train), (X\_test, y\_test)=tf.keras.datasets.mnist.load\_data()

X\_train=X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test=X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train=y\_train.astype(np.int32)

y\_test=y\_test.astype(np.int32)

X\_valid, X\_train=X\_train[:5000], X\_train[5000:]

y\_valid, y\_train=y\_train[:5000], y\_train[5000:]

X\_train2\_full=X\_train[y\_train>=5]

y\_train2\_full=y\_train[y\_train>=5]-5

X\_valid2\_full=X\_valid[y\_valid>=5]

y\_valid2\_full=y\_valid[y\_valid>=5]-5

X\_test2=X\_test[y\_test>=5]

y\_test2=y\_test[y\_test>=5]-5

# Initialize parameters

path = "./model0.ckpt"

new\_path = "./transfer1.ckpt"

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

output\_num = 5

# Import graph

saver = tf.train.import\_meta\_graph(path+".meta")

# Initialize new saver

new\_saver = tf.train.Saver()

# Restore placeholders

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

Y = tf.get\_default\_graph().get\_tensor\_by\_name("Y:0")

batch\_size = tf.get\_default\_graph().get\_tensor\_by\_name("batch\_size:0")

learning\_rate = tf.get\_default\_graph().get\_tensor\_by\_name("Adam/learning\_rate:0")

H4 = tf.get\_default\_graph().get\_tensor\_by\_name("H4/Elu:0")

loss = tf.get\_default\_graph().get\_tensor\_by\_name("loss:0")

accuracy = tf.get\_default\_graph().get\_tensor\_by\_name("accuracy:0")

graph = tf.get\_default\_graph()

# overwrite the output layer

Y\_hat = tf.layers.dense(H4, output\_num, name="Y\_hat")

# Restore the dataset initialization operation

dataset\_init\_op = graph.get\_operation\_by\_name('dataset\_init')

# Freeze lower layers

with tf.name\_scope("train"):

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

train\_vars=tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES,

scope="Y\_hat")

optimizer = optimizer.minimize(loss, var\_list=train\_vars)

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Start training

with tf.Session() as sess :

# Restore the session

saver.restore(sess, path)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:X\_train2\_full.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_valid2\_full, Y:y\_valid2\_full,

batch\_size:X\_valid2\_full.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = new\_saver.save(sess, new\_path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

new\_saver.restore(sess, new\_path)

sess.run(dataset\_init\_op, feed\_dict={X:X\_test2, Y:y\_test2,

batch\_size:X\_test2.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()

2.2.4

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import time

He\_init = tf.variance\_scaling\_initializer()

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# Load data sets

(X\_train, y\_train), (X\_test, y\_test)=tf.keras.datasets.mnist.load\_data()

X\_train=X\_train.astype(np.float32).reshape(-1,28\*28)/255.0

X\_test=X\_test.astype(np.float32).reshape(-1,28\*28)/255.0

y\_train=y\_train.astype(np.int32)

y\_test=y\_test.astype(np.int32)

X\_valid, X\_train=X\_train[:5000], X\_train[5000:]

y\_valid, y\_train=y\_train[:5000], y\_train[5000:]

X\_train2\_full=X\_train[y\_train>=5]

y\_train2\_full=y\_train[y\_train>=5]-5

X\_valid2\_full=X\_valid[y\_valid>=5]

y\_valid2\_full=y\_valid[y\_valid>=5]-5

X\_test2=X\_test[y\_test>=5]

y\_test2=y\_test[y\_test>=5]-5

# Initialize parameters

path = "./model0.ckpt"

new\_path = "./transfer2.ckpt"

epoch\_num = 1000

train\_batch\_size = 20

train\_batch\_num = int(epoch\_num / train\_batch\_size)

max\_check = 20

output\_num = 5

# Import graph

saver = tf.train.import\_meta\_graph(path+".meta")

# Initialize new saver

new\_saver = tf.train.Saver()

# Restore placeholders

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

Y = tf.get\_default\_graph().get\_tensor\_by\_name("Y:0")

batch\_size = tf.get\_default\_graph().get\_tensor\_by\_name("batch\_size:0")

learning\_rate = tf.get\_default\_graph().get\_tensor\_by\_name("Adam/learning\_rate:0")

H4 = tf.get\_default\_graph().get\_tensor\_by\_name("H4/Elu:0")

loss = tf.get\_default\_graph().get\_tensor\_by\_name("loss:0")

accuracy = tf.get\_default\_graph().get\_tensor\_by\_name("accuracy:0")

graph = tf.get\_default\_graph()

# overwrite the output layer

Y\_hat = tf.layers.dense(H4, output\_num, name="Y\_hat")

# Restore the dataset initialization operation

dataset\_init\_op = graph.get\_operation\_by\_name('dataset\_init')

# Freeze lower layers

with tf.name\_scope("train"):

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

train\_vars=tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES,

scope="H3|H4|Y\_hat")

optimizer = optimizer.minimize(loss, var\_list=train\_vars)

# Initialize plotting coordinates

epoch\_axis = []

epoch\_loss\_axis = []

epoch\_error\_axis = []

valid\_loss\_axis = []

valid\_error\_axis = []

check\_count = 0

best\_loss = np.infty

best\_error = 0

# Start training

with tf.Session() as sess :

# Restore the session

saver.restore(sess, path)

for epoch in range(epoch\_num) :

# train for one epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:train\_batch\_size})

for batch in range(train\_batch\_num) :

sess.run(optimizer)

# Compute current loss and error

sess.run(dataset\_init\_op, feed\_dict={X:X\_train2\_full, Y:y\_train2\_full,

batch\_size:X\_train2\_full.shape[0]})

epoch\_loss, epoch\_acc = sess.run([loss, accuracy])

epoch\_error = (1.0 - epoch\_acc) \* 100

print("Epoch =", epoch, "loss =", "{:.9f}".format(epoch\_loss),

"error =", "{:.2f}".format(epoch\_error), "per cent")

# Calculate validation loss and error for this epoch

sess.run(dataset\_init\_op, feed\_dict={X:X\_valid2\_full, Y:y\_valid2\_full,

batch\_size:X\_valid2\_full.shape[0]})

valid\_loss, valid\_acc = sess.run([loss, accuracy])

valid\_error = (1.0 - valid\_acc) \* 100

# Update the coordinates

epoch\_axis.append(epoch)

epoch\_loss\_axis.append(epoch\_loss)

epoch\_error\_axis.append(epoch\_error)

valid\_loss\_axis.append(valid\_loss)

valid\_error\_axis.append(valid\_error)

if valid\_loss < best\_loss :

save\_path = new\_saver.save(sess, new\_path)

best\_loss = valid\_loss

best\_error = valid\_error

check\_count = 0

else :

check\_count += 1

if check\_count > max\_check :

print("Early stopping!")

break

# Calculate test precision using the best model

new\_saver.restore(sess, new\_path)

sess.run(dataset\_init\_op, feed\_dict={X:X\_test2, Y:y\_test2,

batch\_size:X\_test2.shape[0]})

test\_acc = sess.run(accuracy)

precision = test\_acc \* 100

print("Test precision is {:.2f} per cent".format(precision))

# Show the final validation results

print("Final validation loss =", "{:.9f}".format(best\_loss))

print("Final validation error =", "{:.2f}".format(best\_error), "per cent")

# Plot and show the graphs

plt.plot(epoch\_axis, epoch\_loss\_axis, label='Training Loss')

plt.plot(epoch\_axis, valid\_loss\_axis, label='Validation Loss')

plt.legend()

plt.show()

plt.plot(epoch\_axis, epoch\_error\_axis, label='Training Error in percentage')

plt.plot(epoch\_axis, valid\_error\_axis, label='Validation Error in percentage')

plt.legend()

plt.show()