COMPGI13: Assignment 1

Antonio Remiro Azócar, MSc Machine Learning

14 February 2017

Note

A one-week deadline extension was granted due to medical issues.
The error/cost learning curves for 1a-d present training values over the minibatches. An additional graph is presented at the end of P1 exemplifying error tracking over the full training set (for **(1b)**).

Code

The implemented programs are printed in the appendix. See the `README` file for more information.
Tools: Code was developed using Python 3.6.0 and TensorFlow version 0.12.1. The following packages were utilised: matplotlib 2.0.0 (pyplot) for learning curve and confusion matrix plotting, scikit-learn 0.18.1 for confusion matrix functionality and numpy 1.12.0 for numerical operations.

- P1: `exercise1a.py`, `exercise1b.py`, `exercise1c.py`, `exercise1d.py`

Parameter optimisation

Hyperparameters are tuned for each model individually and the models are trained until convergence. Learning curves have been utilised to monitor the training process and decide on suitable hyperparameters; 'regularising' if necessary by early stopping to avoid overfitting/'plateauing'. A minibatch size of 50, a SGD step length of 0.5 and a total number of 5,000-6,000 training iterations work well to train all models in P1. Note that one 'training iteration' refers to a training epoch over a given minibatch (not over the full set of examples).

1 MNIST with TensorFlow

Training and testing rates for all models at the end of the optimisation are presented at the end of the section. The TensorFlow random seed is fixed at 555 to facilitate result reproducibility.

(1a) 1 linear layer, followed by a softmax. The weights/biases are initialised as zeros. The training and testing errors are reported via Figure 1. Figure 1 presents a plot of the errors as a function of iterations during training.

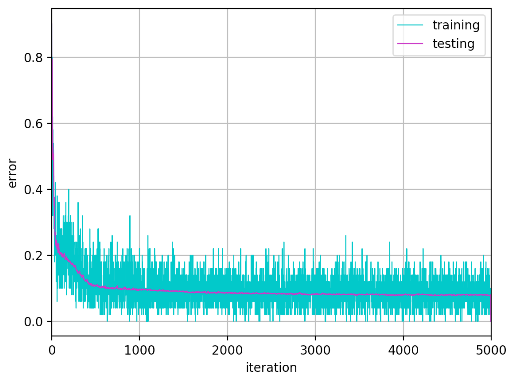

Figure 1: Plot of training/testing errors as a function of iterations during training for **(1a)**.

The cross-entropy loss (training and testing) is also plotted as a function of training iterations. This plot is presented in Figure 2.

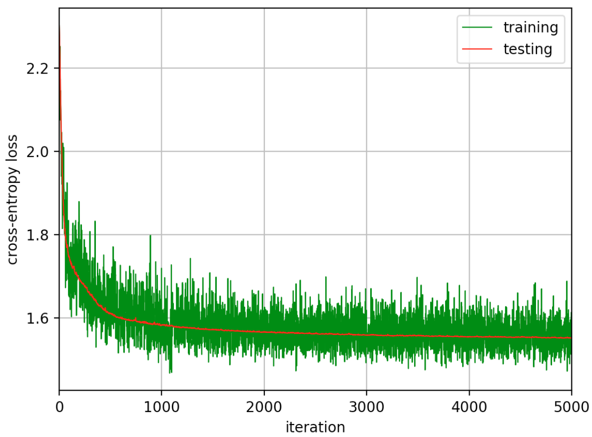

Figure 2: Plot of training/testing cross-entropy loss as a function of training iterations for **(1a)**.

The greater variance in training values wrt testing values in Figure 1 and Figure 2 corresponds to the much smaller size of the 'minibatch' training sets wrt to the full test set. Figure 3 presents the confusion matrix for all classes over **all iterations** for the **test set**. Note that the colour map does not represent error probabilities; it represents the actual number of misclassified points. The right-hand-side bar provides a key. Also note that values have not been normalised with respect to the true labels. Correct classifications hugely outweigh errors; they have been omitted to obtain a better sense of the error spread. We can see that the most common errors involve mistaking a 5 (true) for a 3 or an 8 (predicted).

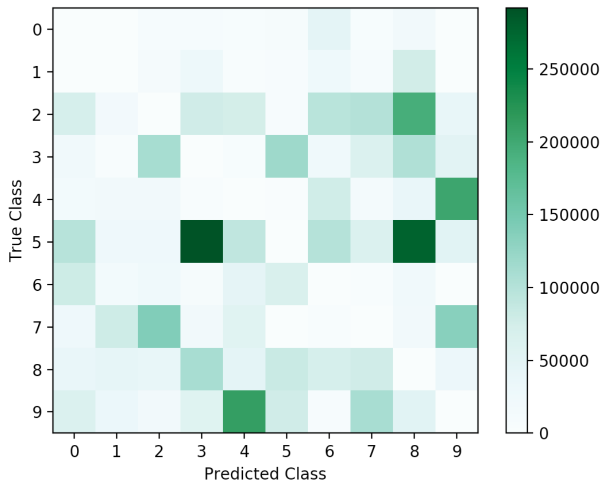

Figure 3: Confusion matrix over all iterations for the test set for **(1a)**.

(1b) 1 hidden layer (128 units) with a ReLU non-linearity, followed by a softmax. The weights are initialised using zero-mean random values from a normal distribution with 0.1 standard deviation. For the hidden layer, the biases are initialised in the same manner but with a standard deviation of 1. The linear layer biases are initialised as zeros. The training and testing errors are reported via Figure 4. Figure 4 presents a plot of the errors as a function of training iterations. Training in this case is performed over 6,000 iterations. The cross-entropy loss (training and testing) is also plotted as a function of iterations during training. This plot is presented in Figure 5. Figure 6 presents the confusion matrix for all classes over **all iterations** for the **test set**. We can observe that the new model makes a lesser number of total mistakes (see right-hand-side bar), having learned to confuse less a 5 for an 8 for example.

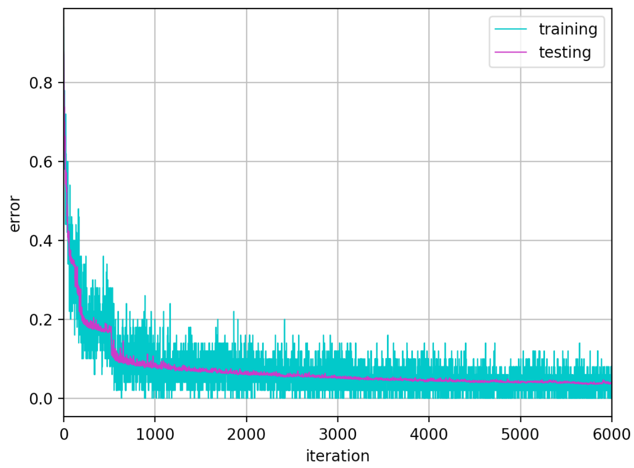

Figure 4: Plot of training/testing errors as a function of training iterations for **(1b)**.

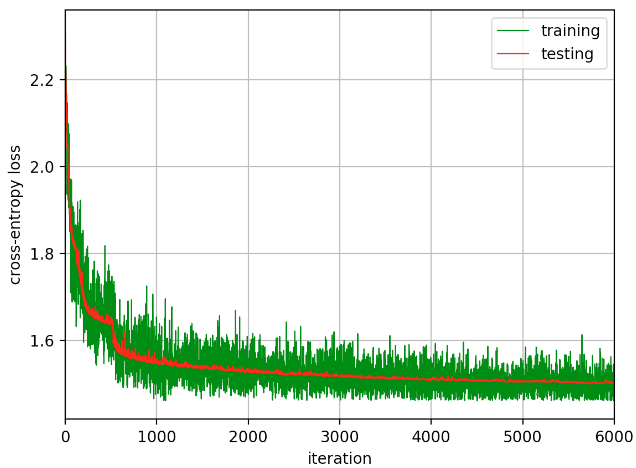

Figure 5: Plot of training/testing cross-entropy loss as a function of iterations during training for **(1b)**.

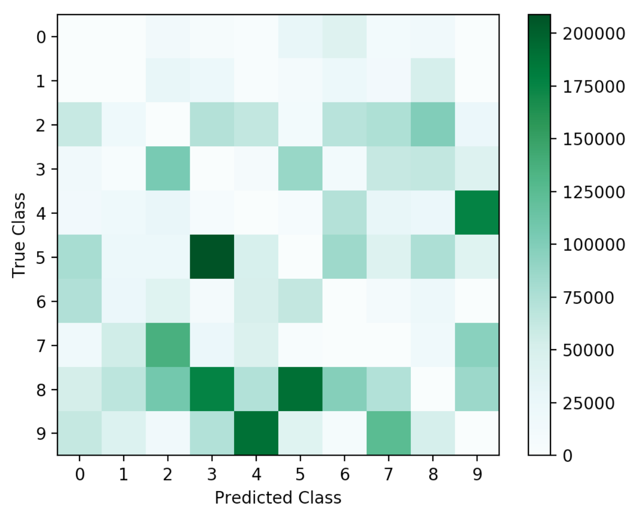

Figure 6: Confusion matrix over all iterations for the test set for **(1b)**.

(1c) 2 hidden layers (256 units) each with a ReLU non-linearity, followed by a softmax. The weights are initialised using zero-mean random values from a normal distribution with 0.1 standard deviation. For the hidden layers, the biases are initialised in the same manner but with a standard deviation of 1. The linear layer biases are initialised as zeros. The training and testing errors for these settings are reported via Figure 7. Figure 7 presents a plot of the errors as a function of iterations during training.

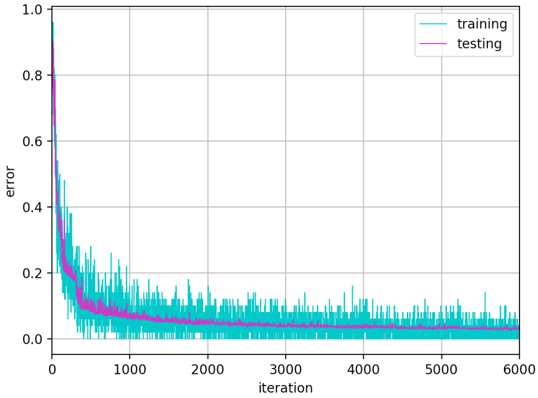

Figure 7: Plot of training/testing errors as a function of iterations during training for **(1c)**.

The cross-entropy loss (training and testing) is also plotted as a function training iterations. This plot is presented in Figure 8.

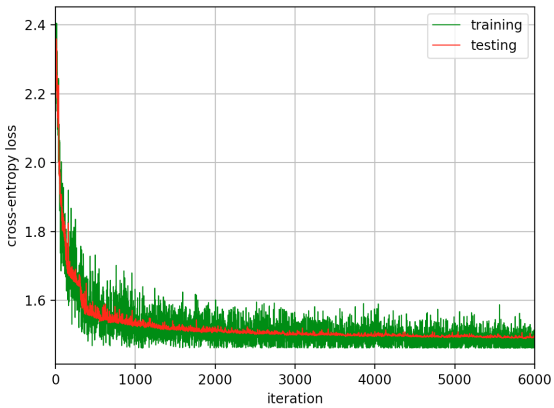

Figure 8: Plot of cross-entropy loss as a function of training iterations for **(1c)**.

Figure 9 presents the confusion matrix for all classes over **all iterations** for the **test set**. We can observe that accuracy has improved further, with the model improving in its classification of 'fives'.

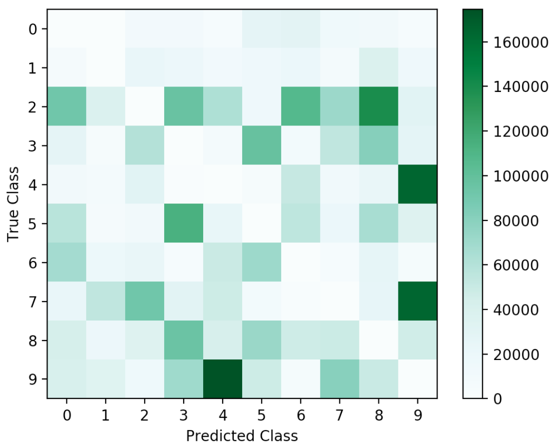

Figure 9: Confusion matrix over all iterations for the test set for **(1c)**.

(1d) 3 layer convolutional model (2 convolutional layers followed by max pooling), followed by 1 non-linear layer (256 units), followed by softmax. The weights of the hidden ReLU layer and the linear layer are initialised using zero-mean random values from a normal distribution with 0.1 standard deviation. For the convolutional layers, the weights are initialised from a truncated zero-mean normal distribution with the same standard deviation. The biases for ReLU and linear layer are generated in the same manner as for **1b-c**. For the convolutional layers, these are generated as constant 0.1s. The training and testing errors for these settings are reported via Figure 10. Figure 10 presents a plot of the errors as a function of iterations during training. In this case, accuracy measurements have been taken every 40 training iterations to make the procedure less computationally expensive.

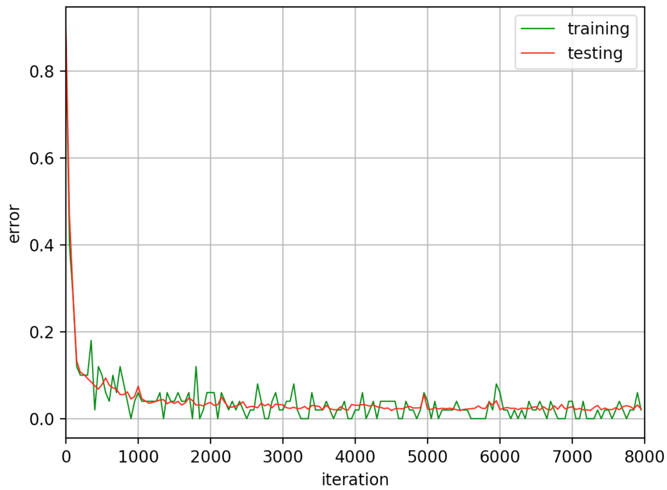

Figure 10: Plot of training/testing error as a function of training iterations for **(1d)**.

Figure 11 presents the confusion matrix for all classes over measurements made every 40 iterations for the **test set**. Note that sampling biases may be present in this representation. The total number of errors is strongly dependent on mistakes made in the first epochs. Given the frequency at which measurements are taken, factors such as weight initialisation or the random seed play a larger role in shaping the matrix.

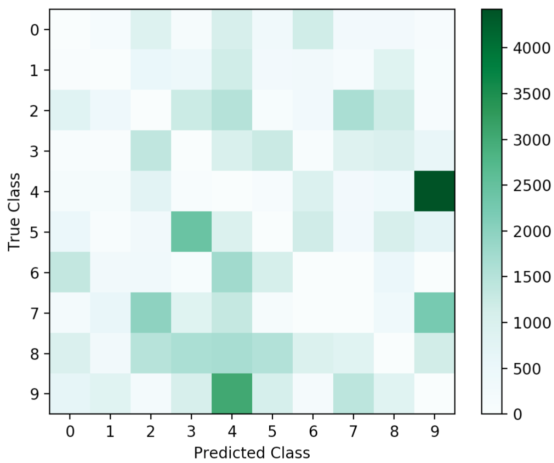

Figure 11: Confusion matrix over selected iterations for the test set for **(1d)**

Final error table

The training and testing errors at the end of the optimisation for P1 are recorded in Table 1.

Exercise	Training error (%)	Test error (%)
1a (5,000 epochs)	8.12	7.95
1b (6,000 epochs)	3.16	3.96
1c (6,000 epochs)	1.95	2.95
1d (8,000 epochs)	3.33	3.46

Table 1: Training/testing errors for P1 at the end of the optimisation.

As expected, it appears that the more complex models perform better. Errors for **1a-c** are reasonably close to previously recorded values. The errors for **1d** were worse than expected, being approximately 2% off from optimality. This suggests that either hyperparameter optimisation was too coarse or that alternative weight/bias initialisations should be considered. Models **1b,c** appear to overfit on the training set.

Additional Graph

Insofar, the presented graphs feature training values over each minibatch. The graph below (Figure 12) provides the training errors over the full train set and the testing errors over the full test set over all iterations for **(1b)**.

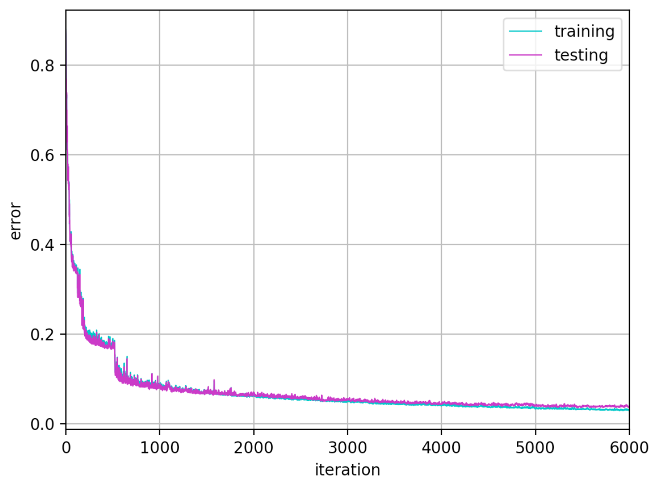

Figure 12: Plot of training (over entire set)/testing error as a function of training iterations for **(1b)**.

2 MNIST without TensorFlow

(2ai) The derivative of the loss function with respect to the scores z, $\frac{\partial loss}{\partial z}$, is computed as follows. Firstly, consider the loss of a single example i; the softmax classifier computes this loss as,

$$loss_i = -\log\Big(\frac{\exp(z_i[y_i])}{\sum_{c=1}^{10}\exp(z_i[c])}\Big),$$

where z is the input array to the softmax layer and $z[c]$ denotes the c-th entry of such. Note that the term in parentheses is equivalent to the softmax output, which we will denote $o_i[k]$ (for example i, array element k). Then,

$$o_i[k] = \frac{\exp(z_i[k])}{\sum_{c=1}^{10}\exp(z_i[c])},$$

and,

$$loss_i = -\log(o_i[y_i]).$$

The gradient for example i, $\frac{\partial loss}{\partial z_i[k]}$, can be decomposed by using the chain rule, with,

$$\frac{\partial loss}{\partial z_i} = \sum_j \frac{\partial loss}{\partial o_j}\frac{\partial o_j}{\partial z_i}.$$

having omitted the array indexing $[k]$. After some algebra, note that,

$$\frac{\partial o_j}{\partial z_i} = o_i(1-o_i), \qquad i = j,$$

$$\frac{\partial o_j}{\partial z_i} = -o_i o_j, \qquad i \neq j,$$

Then for the original derivative,

$$\frac{\partial loss}{\partial z_i} = \sum_j \frac{\partial loss}{\partial o_j}\frac{\partial o_j}{\partial z_i} = -\sum_j \frac{\partial \log o_j}{\partial z_i} = -\sum_j \frac{1}{o_j}\frac{o_j}{z_i},$$

$$= -(1-o_i) - \sum_{j\neq i}\frac{(-o_j o_i)}{o_j} = -(1-o_i) + \sum_{j\neq i} o_i = o_i - \overrightarrow{1} \quad (i=j).$$

In the original framework, provided $y_i = k$, $\frac{\partial loss}{\partial z_i} = o_i - \overrightarrow{1}$.

(2aii) The derivative of the loss function with respect to the inputs x, $\frac{\partial loss}{\partial x}$ for the **(1a)** setup, is computed via back-propagation. Note x_j are the inputs of the linear layer, we denote softmax as inputs z_k (outputs of the linear layer) and softmax outputs as o_k. Via the chain rule,

$$\frac{\partial loss}{\partial x_j} = \sum_k \frac{\partial loss}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial x_j}.$$

Since in the linear layer, $z_k = \sum_j w_{jk}x_j + b_{0k}$, $\frac{\partial z_k}{\partial x_j} = w_{jk}$. From the previous exercise, we know $\frac{\partial loss}{\partial z_k} = \sum_j \frac{\partial loss}{\partial o_j}\frac{\partial o_j}{\partial z_k} = o_k - \overrightarrow{1}$. Hence,

$$\frac{\partial loss}{\partial x_j} = \sum_k (o_k - \overrightarrow{1})w_{jk}, \quad \text{i.e. } \frac{\partial loss}{\partial x_j[n]} = \sum_k (o_k[n] - \overrightarrow{1})w_{jk} \text{ for each array element } n.$$

Similarly, the derivative of the loss function wrt the weights, $\frac{\partial loss}{\partial w}$, is computed via,

$$\frac{\partial loss}{\partial w_{jk}} = \frac{\partial loss}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}} = (o_k - \overrightarrow{1})x_j$$

i.e. $\frac{\partial loss}{\partial w_{jk}} = \sum_{n=1}^{N}(o_k[n] - \overrightarrow{1})x_j[n]$, summing over N array elements. In the same manner, for the derivative of the loss function wrt the bias, $\frac{\partial loss}{\partial b}$, via,

$$\frac{\partial loss}{\partial b_{0k}} = \frac{\partial loss}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial b_{0k}} = (o_k - \overrightarrow{1}).$$

(2aiii) The derivative of a convolution layer wrt to its parameters W is computed as follows. Firstly, we denote the gradients of the convolutional layer as δ_{ij} (e.g. $\delta_{11}, \delta_{12}, \delta_{21}, \delta_{22}$) - with respect to the output. Two updates will be performed: one on the weights W_{ij} and another on the delta gradients. Then, the derivative of the layer loss L wrt W can be computed via the chain rule, with,

$$\frac{\partial L}{\partial W_{ij}^{(l)}} = \sum_{i'}\sum_{j'} \frac{\partial L}{\partial o_{i'j'}^{(l)}}\frac{\partial o_{i'j'}^{(l)}}{\partial W_{ij}^{(l)}} = \sum_{i'}\sum_{j'} \delta_{i'j'}^{l}\frac{\partial o_{i'j'}^{(l)}}{\partial W_{ij}^{(l)}},$$

where $o_{ij}^{(l)}$ is the output vector at layer l, and by definition, $\frac{\partial L}{\partial o_{i'j'}^{(l)}} = \delta_{i'j'}^{l}$. Now, we know,

$$o_{i'j'}^{(l)} = W_{i'j'}^{(l)}x_{i'j'}^{(l)} + b^{(l)},$$

where $b^{(l)}$ are the specified biases for the layer and $x_{i'j'}^{(l)}$ is the reshaped input 4-d tensor. We then have,

$$\frac{\partial o_{i'j'}^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial}{\partial W_{ij}^{(l)}}\Big(\sum_{i''}\sum_{j''} W_{i''j''}^{(l)}x_{(i'-i'')(j'-j'')}^{(l)} + b^{(l)}\Big).$$

This expression can be presented as,

$$\frac{\partial o^{(l)}_{i'j'}}{\partial W^{(l)}_{ij}} = \frac{\partial}{\partial W^{(l)}_{ij}} \Big(\sum_{i''} \sum_{j''} W^{(l)}_{i''j''} A^{(l)}(o^{(l-1)}_{(i'-i'')(j'-j''))}) + b^{(l)} \Big),$$

where A is an activation mapping intaking the output of the previous layer. Expanding and computing partial derivatives for the expression above yields zeros other than when $i = i''$ and $j = j''$ for the weights. Therefore, this implies,

$$\frac{\partial o^{(l)}_{i'j'}}{\partial W^{(l)}_{ij}} = \frac{\partial}{\partial W^{(l)}_{ij}} \Big(W^{(l)}_{ij} A^{(l)}(o^{(l-1)}_{(i'-i)(j'-j)}) \Big) = A^{(l)}(o^{(l-1)}_{(i'-i)(j'-j)}),$$

since $i' - i'' = i' - i$, $j' - j" = j' - j$. We can substitute the expression above into our original equation to give,

$$\frac{\partial L}{\partial W^{(l)}_{ij}} = \sum_{i'} \sum_{j'} \delta^{(l)}_{i'j'} A^{(l)}(o^{(l-1)}_{(i'-i)(j'-j)}) = \delta^{(l)}_{ij} * A^{(l)}(o^{(l-1)}_{(-i)(-j)})$$

Note that an 180 degree rotation of the kernel transforms $(o^{(l-1)}_{(-i)(-j)})$ to $(o^{(l-1)}_{(i)(j)})$. Hence,

$$\frac{\partial L}{\partial W^{(l)}_{ij}} = \delta^{(l)}_{ij} * A^{(l)}(\ rot_{180}\ o^{(l-1)}_{(ij)}).$$

The derivative of the convolution layer loss wrt to its input x (4-dim tensor) is computed as follows. Via backpropagation (chain rule), we have,

$$\frac{\partial L}{\partial x^{(l)}_{ij}} = \sum_{i'} \sum_{j'} \frac{\partial L}{\partial o^{(l)}_{i'j'}} \frac{\partial o^{(l)}_{i'j'}}{\partial W^{(l)}_{i'j'}} \frac{\partial o^{(l)}_{i'j'}}{\partial x^{(l)}_{ij}}$$

From the previous question , and since $\frac{\partial o^{(l)}_{i'j'}}{\partial x^{(l)}_{ij}} = W^{(l)}_{i'j'}$,

$$\frac{\partial L}{\partial x^{(l)}_{ij}} = \Big(\delta^{(l)}_{ij} * A^{(l)}(\ rot_{180}\ o^{(l-1)}_{(ij)}) \Big) \sum_{i'} \sum_{j'} W^{(l)}_{i'j'}.$$

2b-e not attempted due to time contraints - other assignments.

Appendix

The model restoring procedure has been commented out for all exercises.

exercise1a.py

```
"""
Neural network model to classify MNIST digits consisting of 1 linear layer,
    followed by a
softmax.
"""


# for compatibility
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# data directory
data_dir = "./MNIST_data/"

# for saving the model
```

```
model_folder = "savedmodels/"
# question
subdirectory = "1a/"
model_filename = model_folder + subdirectory + "model_1a.ckpt"

# load and read MNIST data, import tensorflow
from tensorflow.examples.tutorials.mnist import input_data
# import dataset with one-hot class encoding
print("Loading the data......")
mnist = input_data.read_data_sets(data_dir, one_hot=True)
print("Data has been loaded. ")
import tensorflow as tf

# import sklearn, matplotlib for confusion matrix generation functionality
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# import numpy for train/test accuracy/loss tracking
import numpy as np

# operating system interface to save cost/accuracy history.
import os

# set a random seed for reproducibility
tf.set_random_seed(555)

# define hyper-parameter values
n_attributes = 784
n_classes = 10
optimiser_step_size = 0.5
tot_training_iterations = 5000
minibatch_size = 50
print_update_every = 200

# one-hot encoding converted to single number class by storing
# index of highest element.
mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])

# placeholder for data x, (784 pixels/attributes)
# placeholder for label y
x = tf.placeholder(tf.float32, [None, n_attributes])
y = tf.placeholder(tf.float32, [None, n_classes])
# placeholder for true single number class (label)
y_klass = tf.placeholder(tf.int64, [None])

# neural network model building
# define variables - initialise weights, biases as tensors full of zeros
weights = tf.Variable(tf.zeros([n_attributes, n_classes]))
biases = tf.Variable(tf.zeros([n_classes]))
# linear regression layer
y_temp = tf.add(tf.matmul(x, weights), biases)
# softmax performed
y_pred = tf.nn.softmax(y_temp)

# one-hot encoding -> predicted numerical label (class), index of largest
    element in row
y_pred_klass = tf.argmax(y_pred, dimension = 1)
# model is trained using cross-entropy loss function
```

```
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=y_pred))
# stochastic gradient descent for minimising objective
optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(cost)
# define the accuracy - percentage of times correct prediction
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# error is 1-accuracy
error = 1 - accuracy

# save weights, biases for train.Saver
tf.add_to_collection('vars', weights)
tf.add_to_collection('vars', biases)

saver = tf.train.Saver()

with tf.Session() as sess:
    # operation to initialise all variables
    tf.global_variables_initializer().run()
    print ("==============================")
    print("Training started........")
    # initialisation of arrays keeping track of loss/error history
    train_loss_history = np.zeros(tot_training_iterations)
    test_loss_history = np.zeros(tot_training_iterations)
    train_error_history = np.zeros(tot_training_iterations)
    test_error_history = np.zeros(tot_training_iterations)
    tot_confmat = np.zeros((n_classes,n_classes))
    # loop over each training iteration
    for iteration in range(tot_training_iterations):
        # load 'minibatch_size' examples in each training iteration
        xbatch, ybatch = mnist.train.next_batch(minibatch_size)
        # error/loss for test and train calculated for each iteration
        train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
        train_error_history[iteration] = train_error
        train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
        train_loss_history[iteration] = train_loss
        test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
        test_error_history[iteration] = test_error
        test_loss = cost.eval(feed_dict={x: mnist.test.images,y: mnist.test.labels})
        test_loss_history[iteration] = test_loss
        # print error/loss values every 'print_update_every' iteration  to track progress
        if iteration % print_update_every == 0:
            print("iteration %d, training error %g" % (iteration, train_error))
            print("iteration %d, training loss %g" % (iteration, train_loss))
            print("iteration %d, testing error %g" % (iteration, test_error))
            print("iteration %d, testing loss %g" % (iteration, test_loss))
        # saves error/loss history to directory
        np.savez(os.getcwd() + "/traininghistory1a", train_loss_history=train_loss_history,
                    test_loss_history=test_loss_history,
                    train_error_history = train_error_history,
                    test_error_history = test_error_history)
        # run optimiser for each batch - update gradients
        optimiser.run(feed_dict={x: xbatch, y: ybatch})
```

```python
        # groups test set images, one-hot encoding, true numerical labels of test set
        feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_klass: mnist.test.klass}
        # test set true numerical classes (labels)
        true_klass = mnist.test.klass
        # test set predicted numerical classes (labels)
        pred_klass = sess.run(y_pred_klass, feed_dict=feed_dict_test)
        # sci-kit learn functionality to generate a confusion matrix
        confmat = confusion_matrix(y_true=true_klass, y_pred=pred_klass)
        tot_confmat = np.add(tot_confmat, confmat)
    saver.save(sess, model_filename)
    print("Training finished.")
    print("==============================\n")
    # in final confusion matrix, the diagonal is set to zeros.
    # correct classifications hugely outweigh errors and are ignored in the confusion matrix plot
    for i in range(n_classes):
        tot_confmat[i,i] = 0
    ############
    # confusion matrix plots
    ############
    # text print of confusion matrix
    print(tot_confmat)
    # image print of confusion matrix
    plt.figure()
    plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
    # make plot nice
    plt.colorbar()
    tick_marks = np.arange(n_classes)
    plt.xticks(tick_marks, range(n_classes))
    plt.yticks(tick_marks, range(n_classes))
    plt.xlabel('Predicted Class')
    plt.ylabel('True Class')
    plt.savefig('1aconfusionmatrix.png')
    ############
    # learning curve plots for cross entropy loss
    ############
    plt.figure()
    traininghistory = np.load(os.getcwd() + "/traininghistory1a.npz")
    trainingloss = traininghistory["train_loss_history"]
    testingloss = traininghistory["test_loss_history"]
    axisx = np.arange(tot_training_iterations)
    plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
    plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("cross-entropy loss")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1alosslearningcurve.png')
    ###
    ### learning curve for error
    ###
    plt.figure()
    traininghistory2 = np.load(os.getcwd() + "/traininghistory1a.npz")
    trainingerror = traininghistory2["train_error_history"]
    testingerror = traininghistory2["test_error_history"]
```

```
    axisx2 = np.arange(tot_training_iterations)
    plt.plot(axisx2, trainingerror, "c-", linewidth=0.8, label="training")
    plt.plot(axisx2, testingerror, "m--", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("error")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1aerrorlearningcurve.png')


"""
### RESTORE MODEL AND OBTAIN FINAL ACCURACY

print("Restoring and testing model")
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "
        model_1a.ckpt.meta")
    new_saver.restore(sess, tf.train.latest_checkpoint(model_folder +
        subdirectory + './'))
    all_vars = tf.get_collection('vars')
    weights = all_vars[0]
    biases = all_vars[1]
    test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.
        labels})
    print("The final test error is %g" % (test_error))

"""
```

exercise1b.py

```
"""
Neural network model to classify MNIST digits consisting of 1 hidden layer
    (128
units) with a ReLu non-linearity, followed by a softmax.
"""
# for compatibility
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# data directory
data_dir = "./MNIST_data/"

# for saving the model
model_folder = "savedmodels/"
# question
subdirectory = "1b/"
model_filename = model_folder + subdirectory + "model_1b.ckpt"

# load and read MNIST data, import tensorflow
from tensorflow.examples.tutorials.mnist import input_data
# import dataset with one-hot class encoding
print("Loading the data......")
mnist = input_data.read_data_sets(data_dir, one_hot=True)
print("Data has been loaded. ")
import tensorflow as tf

# import sklearn, matplotlib for confusion matrix generation functionality
```

```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# import numpy for train/test accuracy/loss tracking
import numpy as np

# operating system interface to save cost/accuracy history.
import os

# set a random seed for reproducibility
tf.set_random_seed(555)

# define hyper-parameter values
n_attributes = 784
n_classes = 10
optimiser_step_size = 0.5
tot_training_iterations = 6000
minibatch_size = 50
print_update_every = 200
# number of units in hidden layer 1
n_units_h1 = 128

# one-hot encoding converted to single number class by storing
# index of highest element.
mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])

# placeholder for data x, (784 pixels/attributes)
# placeholder for label y
x = tf.placeholder(tf.float32, [None, n_attributes])
y = tf.placeholder(tf.float32, [None, n_classes])
# placeholder for true single number class (label)
y_klass = tf.placeholder(tf.int64, [None])


# neural network model building
# define variables - initialise weights, biases
# hidden layer 1
W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="W_h1")
b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
# add ReLu non-linearity to hidden layer 1
hidden1 = tf.nn.relu(tf.matmul(x, W_h1) + b_h1)
# linear layer
W = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_classes]), name="W")
b = tf.Variable(tf.zeros([n_classes]), name="b")
#  put together linear layer
z = tf.matmul(hidden1, W) + b
# softmax performed
y_pred = tf.nn.softmax(z)
# one-hot encoding -> predicted numerical label (class), index of largest element in row
y_pred_klass = tf.argmax(y_pred, dimension=1)
# model is trained using cross-entropy loss function
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=y_pred))
# 0.61 optimal
optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(cost)
# define the accuracy - percentage of times correct prediction
```

```python
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# error is 1-accuracy
error = 1 - accuracy

# save final  weights, biases for train.Saver
tf.add_to_collection('vars', W)
tf.add_to_collection('vars', b)

saver = tf.train.Saver()

# NEURAL NETWORK TRAINING COMMENCES
with tf.Session() as sess:
    # operation to initialise all variables
    tf.global_variables_initializer().run()
    print ("==================================================")
    print("Training started........")
    # initialisation of arrays keeping track of loss/error history
    train_loss_history = np.zeros(tot_training_iterations)
    test_loss_history = np.zeros(tot_training_iterations)
    train_error_history = np.zeros(tot_training_iterations)
    test_error_history = np.zeros(tot_training_iterations)
    tot_confmat = np.zeros((n_classes,n_classes))
    # loop over each training iteration
    for iteration in range(tot_training_iterations):
        # load 'minibatch_size' examples in each training iteration
        xbatch, ybatch = mnist.train.next_batch(minibatch_size)
        # error/loss for test and train calculated for each iteration
        train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
        train_error_history[iteration] = train_error
        train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
        train_loss_history[iteration] = train_loss
        test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
        test_error_history[iteration] = test_error
        test_loss = cost.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
        test_loss_history[iteration] = test_loss
        # print error/loss values every 'print_update_every' iteration  to track progress
        if iteration % print_update_every == 0:
            print("iteration %d, training error %g" % (iteration, train_error))
            print("iteration %d, training loss %g" % (iteration, train_loss))
            print("iteration %d, testing error %g" % (iteration, test_error))
            print("iteration %d, testing loss %g" % (iteration, test_loss))
        # saves error/loss history to directory
        np.savez(os.getcwd() + "/traininghistory1b", train_loss_history=train_loss_history,
                 test_loss_history=test_loss_history,
                 train_error_history=train_error_history,
                 test_error_history=test_error_history)
        # run optimiser for each batch - update gradients
        optimiser.run(feed_dict={x: xbatch, y: ybatch})
        # groups test set images, one-hot encoding, true numerical labels of test set
        feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_klass: mnist.test.klass}
        # test set true numerical classes (labels)
```

```
        true_klass = mnist.test.klass
        # test set predicted numerical classes (labels)
        pred_klass = sess.run(y_pred_klass, feed_dict=feed_dict_test)
        # sci-kit learn functionality to generate a confusion matrix
        confmat = confusion_matrix(y_true=true_klass, y_pred=pred_klass)
        tot_confmat = np.add(tot_confmat, confmat)
    # saving
    saver.save(sess, model_filename)
    print("Training finished.")
    print("==================================================\n")
    # in final confusion matrix, the diagonal is set to zeros.
    # correct classifications hugely outweigh errors and are ignored in the confusion matrix plot
    for i in range(n_classes):
        tot_confmat[i, i] = 0
    ############
    # confusion matrix plots
    ############
    # text print of confusion matrix
    print(tot_confmat)
    # image print of confusion matrix
    plt.figure()
    plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
    # make plot nice
    plt.colorbar()
    tick_marks = np.arange(n_classes)
    plt.xticks(tick_marks, range(n_classes))
    plt.yticks(tick_marks, range(n_classes))
    plt.xlabel('Predicted Class')
    plt.ylabel('True Class')
    plt.savefig('1bconfusionmatrix.png')
    ############
    # learning curve plots for loss/error
    ############
    plt.figure()
    traininghistory = np.load(os.getcwd() + "/traininghistory1b.npz")
    trainingloss = traininghistory["train_loss_history"]
    testingloss = traininghistory["test_loss_history"]
    axisx = np.arange(tot_training_iterations)
    plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
    plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("cross-entropy loss")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1blosslearningcurve.png')
    ###
    ### learning curve for error
    ###
    plt.figure()
    traininghistory2 = np.load(os.getcwd() + "/traininghistory1b.npz")
    trainingerror = traininghistory2["train_error_history"]
    testingerror = traininghistory2["test_error_history"]
    axisx2 = np.arange(tot_training_iterations)
    plt.plot(axisx2, trainingerror, "c-", linewidth=0.8, label="training")
    plt.plot(axisx2, testingerror, "m-", linewidth=0.8, label="testing")
    plt.grid()
```

```
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("error")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1berrorlearningcurve.png')

"""

### RESTORE MODEL AND OBTAIN FINAL ACCURACY

print("Restoring and testing model")
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "model_1b.ckpt.meta")
    new_saver.restore(sess, tf.train.latest_checkpoint(model_folder + subdirectory + './'))
    all_vars = tf.get_collection('vars')
    weights = all_vars[0]
    biases = all_vars[1]
    train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train.labels})
    test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
    print("The final train error is %g" % (train_error))
    print("The final test error is %g" % (test_error))

"""
```

exercise1c.py

```
"""
Neural network model to classify MNIST digits consisting of 2 hidden layers (256
units each) with a ReLu non-linearity, followed by a softmax.
"""
# for compatibility
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# data directory
data_dir = "./MNIST_data/"

# for saving the model
model_folder = "savedmodels/"
# question
subdirectory = "1c/"
model_filename = model_folder + subdirectory + "model_1c.ckpt"

# load and read MNIST data, import tensorflow
from tensorflow.examples.tutorials.mnist import input_data
# import dataset with one-hot class encoding
print("Loading the data......")
mnist = input_data.read_data_sets(data_dir, one_hot=True)
print("Data has been loaded. ")
import tensorflow as tf

# import sklearn, matplotlib for confusion matrix generation functionality
```

```python
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# import numpy for train/test accuracy/loss tracking
import numpy as np

# operating system interface to save cost/accuracy history.
import os

# set a random seed for reproducibility
tf.set_random_seed(555)

# define hyper-parameter values
n_attributes = 784
n_classes = 10
optimiser_step_size = 0.5
tot_training_iterations = 6000
minibatch_size = 50
print_update_every = 200
# number of units in hidden layer 1
n_units_h1 = 256
# number of units in hidden layer 2
n_units_h2 = 256

# one-hot encoding converted to single number class by storing
# index of highest element.
mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])

# placeholder for data x, (784 pixels/attributes)
# placeholder for label y
x = tf.placeholder(tf.float32, [None, n_attributes])
y = tf.placeholder(tf.float32, [None, n_classes])
# placeholder for true single number class (label)
y_klass = tf.placeholder(tf.int64, [None])

# neural network model building
# define variables - initialise weights, biases
# hidden layer 1
W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="
    W_h1")
b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
# add ReLu non-linearity to hidden layer 1
hidden1 = tf.nn.relu(tf.matmul(x, W_h1) + b_h1)
# hidden layer 2
W_h2 = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_units_h2]), name="
    W_h2")
b_h2 = tf.Variable(tf.random_normal([n_units_h2]), name="b_h2")
# add ReLu non-linearity to hidden layer 2
hidden2 = tf.nn.relu(tf.matmul(hidden1, W_h2) + b_h2)
# linear layer
W = tf.Variable(0.1 * tf.random_normal([n_units_h2, n_classes]), name="W")
b = tf.Variable(tf.zeros([n_classes]), name="b")
#  put together linear layer
z = tf.matmul(hidden2, W) + b
# softmax performed
y_pred = tf.nn.softmax(z)
# one-hot encoding -> predicted numerical label (class), index of largest
    element in row
y_pred_klass = tf.argmax(y_pred, dimension=1)
```

```python
# model is trained using cross-entropy loss function
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=y_pred))
# 0.61 optimal
optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(cost)
# define the accuracy - percentage of times correct prediction
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# error is 1-accuracy
error = 1 - accuracy

# save final weights, biases for train.Saver
tf.add_to_collection('vars', W)
tf.add_to_collection('vars', b)

saver = tf.train.Saver()

# NEURAL NETWORK TRAINING COMMENCES

with tf.Session() as sess:
    # operation to initialise all variables
    tf.global_variables_initializer().run()
    print ("========================================")
    print("Training started........")
    # initialisation of arrays keeping track of loss/error history
    train_loss_history = np.zeros(tot_training_iterations)
    test_loss_history = np.zeros(tot_training_iterations)
    train_error_history = np.zeros(tot_training_iterations)
    test_error_history = np.zeros(tot_training_iterations)
    tot_confmat = np.zeros((n_classes,n_classes))
    # loop over each training iteration
    for iteration in range(tot_training_iterations):
        # load 'minibatch_size' examples in each training iteration
        xbatch, ybatch = mnist.train.next_batch(minibatch_size)
        # error/loss for test and train calculated for each iteration
        train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
        train_error_history[iteration] = train_error
        train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
        train_loss_history[iteration] = train_loss
        test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
        test_error_history[iteration] = test_error
        test_loss = cost.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
        test_loss_history[iteration] = test_loss
        # print error/loss values every 'print_update_every' iteration to track progress
        if iteration % print_update_every == 0:
            print("iteration %d, training error %g" % (iteration, train_error))
            print("iteration %d, training loss %g" % (iteration, train_loss))
            print("iteration %d, testing error %g" % (iteration, test_error))
            print("iteration %d, testing loss %g" % (iteration, test_loss))
        # saves error/loss history to directory
        np.savez(os.getcwd() + "/traininghistory1C", train_loss_history=train_loss_history,
                 test_loss_history=test_loss_history,
                 train_error_history=train_error_history,
```

```
                    test_error_history=test_error_history)
        # run optimiser for each batch - update gradients
        optimiser.run(feed_dict={x: xbatch, y: ybatch})
        # groups test set images, one-hot encoding, true numerical labels of test set
        feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_klass: mnist.test.klass}
        # test set true numerical classes (labels)
        true_klass = mnist.test.klass
        # test set predicted numerical classes (labels)
        pred_klass = sess.run(y_pred_klass, feed_dict=feed_dict_test)
        # sci-kit learn functionality to generate a confusion matrix
        confmat = confusion_matrix(y_true=true_klass, y_pred=pred_klass)
        tot_confmat = np.add(tot_confmat, confmat)
    # saving
    print("Training finished.")
    print("====================================\n")
    saver.save(sess, model_filename)
    # in final confusion matrix, the diagonal is set to zeros.
    # correct classifications hugely outweigh errors and are ignored in the confusion matrix plot
    for i in range(n_classes):
        tot_confmat[i, i] = 0
    ############
    # confusion matrix plots
    ############
    # text print of confusion matrix
    print(tot_confmat)
    # image print of confusion matrix
    plt.figure()
    plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
    # make plot nice
    plt.colorbar()
    tick_marks = np.arange(n_classes)
    plt.xticks(tick_marks, range(n_classes))
    plt.yticks(tick_marks, range(n_classes))
    plt.xlabel('Predicted Class')
    plt.ylabel('True Class')
    plt.savefig('confmat1c.png')
    ############
    # learning curve plots for loss/error
    ############
    plt.figure()
    traininghistory = np.load(os.getcwd() + "/traininghistory1c.npz")
    trainingloss = traininghistory["train_loss_history"]
    testingloss = traininghistory["test_loss_history"]
    axisx = np.arange(tot_training_iterations)
    plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
    plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("cross-entropy loss")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1closslearningcurve.png')
    ###
    ###  learning curve for error
    ###
```

```python
    plt.figure()
    traininghistory2 = np.load(os.getcwd() + "/traininghistory1c.npz")
    trainingerror = traininghistory2["train_error_history"]
    testingerror = traininghistory2["test_error_history"]
    axisx2 = np.arange(tot_training_iterations)
    plt.plot(axisx2, trainingerror, "c--", linewidth=0.8, label="training")
    plt.plot(axisx2, testingerror, "m--", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("error")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1cerrorlearningcurve.png')


"""

### RESTORE MODEL AND OBTAIN FINAL ACCURACY

print("Restoring and testing model")
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "model_1c.ckpt.meta")
    new_saver.restore(sess, tf.train.latest_checkpoint(model_folder + subdirectory + './'))
    all_vars = tf.get_collection('vars')
    weights = all_vars[0]
    biases = all_vars[1]
    train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train.labels})
    test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
    print("The final train error is %g" % (train_error))
    print("The final test error is %g" % (test_error))

"""
```

exercise1d.py

```python
"""
Neural network model to classify MNIST digits consisting of a 3 layer convolutional
model (2 convolutional layers followed by max pooling), followed by one non-linear
layer (256 units) followed by a softmax.
"""
# for compatibility
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# data directory
data_dir = "./MNIST_data/"

# for saving the model
model_folder = "savedmodels/"
# question
subdirectory = "1d/"
model_filename = model_folder + subdirectory + "model_1d.ckpt"
```

```
# load and read MNIST data, import tensorflow
from tensorflow.examples.tutorials.mnist import input_data
# import dataset with one-hot class encoding
print("Loading the data......")
mnist = input_data.read_data_sets(data_dir, one_hot=True)
print("Data has been loaded. ")
import tensorflow as tf

# import sklearn, matplotlib for confusion matrix generation functionality
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# import numpy for train/test accuracy/loss tracking
import numpy as np

# operating system interface to save cost/accuracy history.
import os

# set a random seed for reproducibility
tf.set_random_seed(555)

# define hyper-parameter values
n_attributes = 784
n_classes = 10
optimiser_step_size = 0.5
tot_training_iterations = 8000
minibatch_size = 50
print_update_every = 50
# number of units in hidden layer 1
n_units_h1 = 256

# one-hot encoding converted to single number class by storing
# index of highest element.
mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])

# placeholder for data x, (784 pixels/attributes)
# placeholder for label y
x = tf.placeholder(tf.float32, [None, n_attributes])
y = tf.placeholder(tf.float32, [None, n_classes])
# placeholder for true single number class (label)
y_klass = tf.placeholder(tf.int64, [None])

# function which initialises (and returns) bias variables,
# it intakes the specified shape 'config' of the variables and creates
# a tensor of bias values populated with 0.1s (FOR LINEAR LAYER)
def create_biases(config):
    init = tf.constant(0.1, shape=config)
    return tf.Variable(init)

# function which initialises (and returns) weight variables.
# it intakes the specified shape 'config' of the variables and creates
# a tensor of weights from a truncated normal distribution (SD 0.1)
def create_weights(config):
    init = tf.truncated_normal(config, stddev=0.1)
    return tf.Variable(init)

# function which intakes input x and filter W and returns and computes
# their 2D convolution, which is returned - 'SAME' padding and plain (no
```

```
    strides)
def convolution(x, weights):
    return tf.nn.conv2d(x, weights, strides=[1, 1, 1, 1], padding='SAME')

# function which intakes input x and performs max pooling on it with 2z2 sliding
# windows - no overlapping pixels.
def pooling(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# neural network model building
# define variables - initialise weights, biases
# convolution layer 1 w/b
Wconvol1 = create_weights([3, 3, 1, 16])
bconvol1 = create_biases([16])
# before applying layer 1, reshape x to 4d tensor
reshaped1 = tf.reshape(x, [-1,28,28,1])
# apply convolution layer 1
hiddenconv1 = tf.add(convolution(reshaped1, Wconvol1), bconvol1)
# apply max pooling 1
pooling1 = pooling(hiddenconv1)
# convolution layer 2 w/b
Wconvol2 = create_weights([3, 3, 16, 16])
bconvol2 = create_biases([16])
# apply convolution layer 2
hiddenconv2 = tf.add(convolution(pooling1, Wconvol2), bconvol2)
# apply max pooling 2
pooling2 = pooling(hiddenconv2)
# FLATTEN
reshapeflat =  tf.reshape(pooling2, [-1, 7*7*16])
# hidden layer
W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="W_h1")
b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
# add ReLu non-linearity to hidden layer 1
hidden1 = tf.nn.relu(tf.matmul(reshapeflat, W_h1) + b_h1)
# linear layer
W = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_classes]), name="W")
b = tf.Variable(tf.zeros([n_classes]), name="b")
#  put together linear layer
z = tf.add(tf.matmul(hidden1, W), b)
# softmax performed
y_pred = tf.nn.softmax(z)
# term definitions
# one-hot encoding -> predicted numerical label (class), index of largest element in row
y_pred_klass = tf.argmax(y_pred, dimension=1)
# model is trained using cross-entropy loss function
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=y_pred))
# run optimiser
optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(cost)
# define the accuracy - percentage of times correct prediction
correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# error is 1-accuracy
error = 1 - accuracy
```

```python

# save final  weights, biases for train.Saver
tf.add_to_collection('vars', W)
tf.add_to_collection('vars', b)

saver = tf.train.Saver()

# NEURAL NETWORK TRAINING
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # operation to initialise all variables
    tf.global_variables_initializer().run()
    print("==================================")
    print("Training started........")
    # initialisation of arrays keeping track of loss/error history
    train_error_history = []
    test_error_history = []
    selected = []
    tot_confmat = np.zeros((n_classes, n_classes))
    # loop over each training iteration
    for iteration in range(tot_training_iterations):
        # load 'minibatch_size' examples in each training iteration
        xbatch, ybatch = mnist.train.next_batch(minibatch_size)

        if iteration % print_update_every == 0:
            # error/loss for test and train calculated for each iteration
            train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
            train_error_history.append(train_error)
            test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels})
            test_error_history.append(test_error)
            selected.append(iteration)
            print("iteration %d, training error %g" % (iteration, train_error))
            print("iteration %d, testing error %g" % (iteration, test_error))
            # saves error/loss history to directory
            np.savez(os.getcwd() + "/traininghistory1d",
                     train_error_history=train_error_history,
                     test_error_history=test_error_history)
            # print error/loss values every 'print_update_every' iteration to track progress
            # groups test set images, one-hot encoding, true numerical labels of test set
            feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_klass: mnist.test.klass}
            # test set true numerical classes (labels)
            true_klass = mnist.test.klass
            # test set predicted numerical classes (labels)
            pred_klass = sess.run(y_pred_klass, feed_dict=feed_dict_test)
            # sci-kit learn functionality to generate a confusion matrix
            confmat = confusion_matrix(y_true=true_klass, y_pred=pred_klass)
            tot_confmat = np.add(tot_confmat, confmat)
        # run optimiser for each batch - update gradients
        optimiser.run(feed_dict={x: xbatch, y: ybatch})
    print("Training finished.")
    print("==================================\n")
    # save model
    saver.save(sess, model_filename)
    # in final confusion matrix, the diagonal is set to zeros.
```

```python
    # correct classifications hugely outweigh errors and are ignored in the confusion matrix plot
    for i in range(n_classes):
        tot_confmat[i, i] = 0
    ############
    # confusion matrix plots
    ############
    # text print of confusion matrix
    print(tot_confmat)
    # image print of confusion matrix
    plt.figure()
    plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
    # make plot nice
    plt.colorbar()
    tick_marks = np.arange(n_classes)
    plt.xticks(tick_marks, range(n_classes))
    plt.yticks(tick_marks, range(n_classes))
    plt.xlabel('Predicted Class')
    plt.ylabel('True Class')
    plt.savefig('confmat1d.png')
    ############
    # learning curve plots for loss/error
    ############
    plt.figure()
    model_history = np.load(os.getcwd() + "/traininghistory1d.npz")
    train_error = model_history["train_error_history"]
    test_error = model_history["test_error_history"]
    x_axis = np.arange(0,tot_training_iterations,print_update_every)
    plt.plot(x_axis, train_error, "g-", linewidth=0.8, label="training")
    plt.plot(x_axis, test_error, "r-", linewidth=0.8, label="testing")
    plt.grid()
    plt.legend()
    plt.xlabel("iteration")
    plt.ylabel("error")
    plt.xlim(0, tot_training_iterations)
    plt.show()
    plt.savefig('1derrorlearningcurve.png')
    ###
    ###
    ###


"""


### RESTORE MODEL AND OBTAIN FINAL ACCURACY

print("Restoring and testing model")
with tf.Session() as sess:
    new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "model_1d.ckpt.meta")
    new_saver.restore(sess, tf.train.latest_checkpoint(model_folder + subdirectory + './'))
    all_vars = tf.get_collection('vars')
    weights = all_vars[0]
    biases = all_vars[1]
    train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train.labels})
    test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.
```

```
        labels})
    print("The final train error is %g" % (train_error))
    print("The final test error is %g" % (test_error))

"""
```