

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.

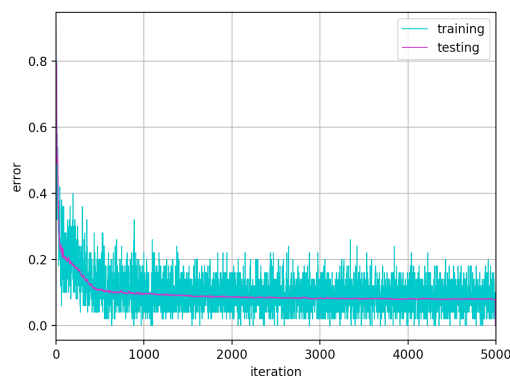


Figure 1: Plot of training/testing errors as a function of iterations during training for **(1a)**.

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

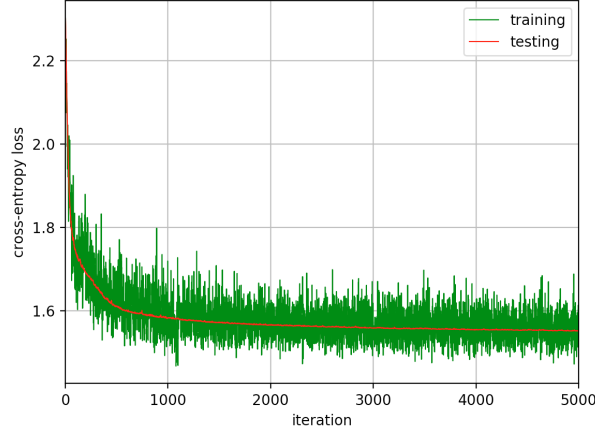


Figure 2: Plot of training/testing cross-entropy loss as a function of training iterations for **(1a)**.

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).

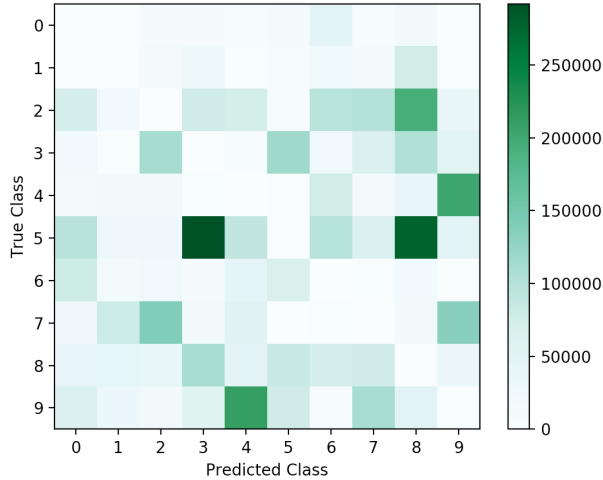


Figure 3: Confusion matrix over all iterations for the test set for **(1a)**.

(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.

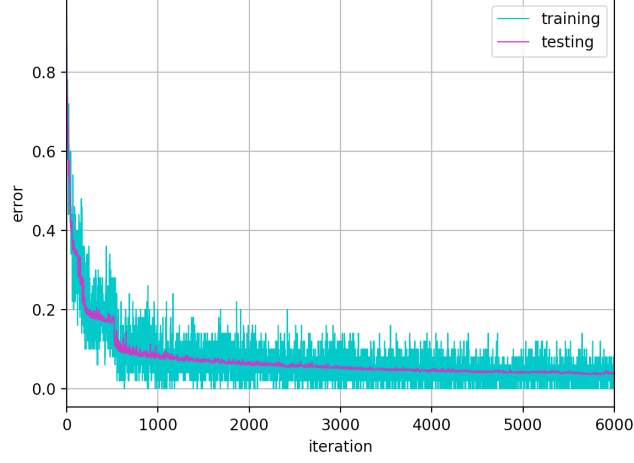


Figure 4: Plot of training/testing errors as a function of training iterations for **(1b)**.

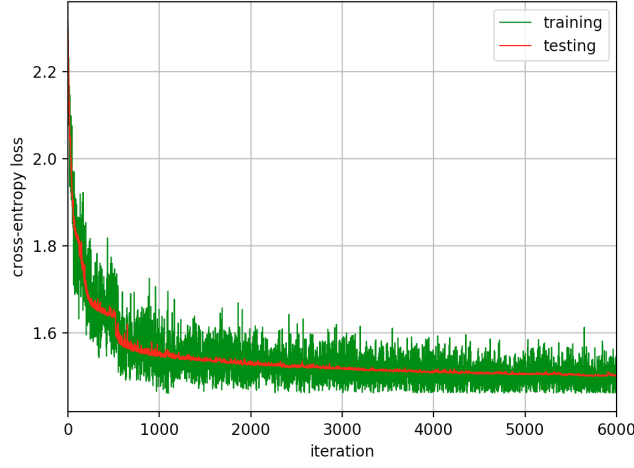


Figure 5: Plot of training/testing cross-entropy loss as a function of iterations during training for **(1b)**.

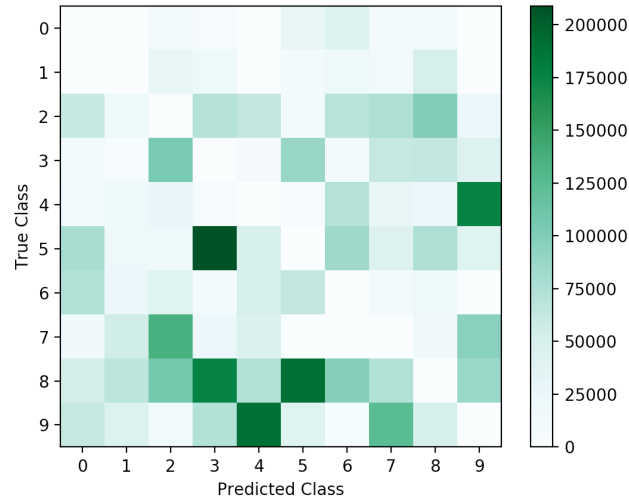


Figure 6: Confusion matrix over all iterations for the test set for **(1b)**.

(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.

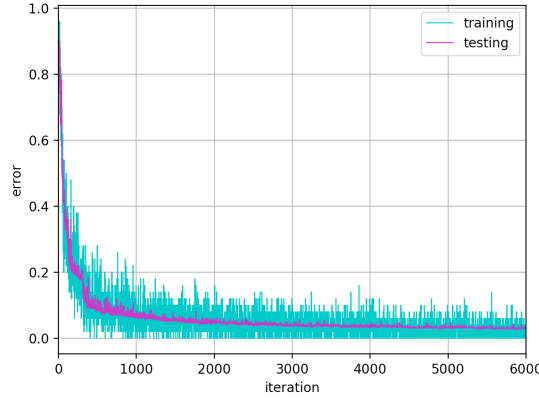


Figure 7: Plot of training/testing errors as a function of iterations during training for (1c).

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

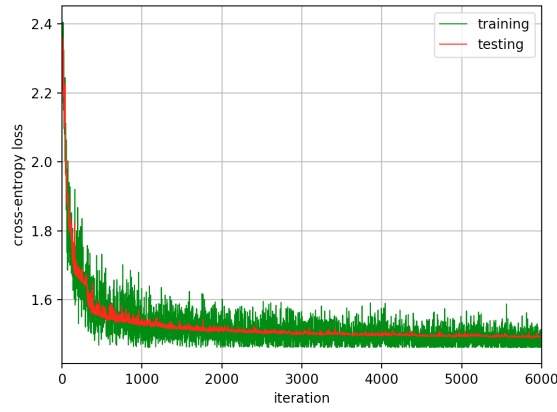


Figure 8: Plot of cross-entropy loss as a function of training iterations for (1c).

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’.

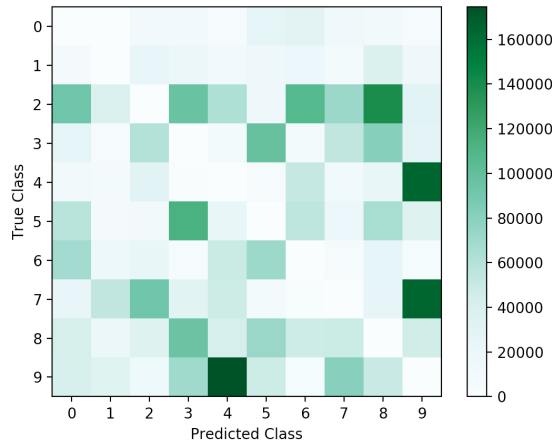


Figure 9: Confusion matrix over all iterations for the test set for (1c).

(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.

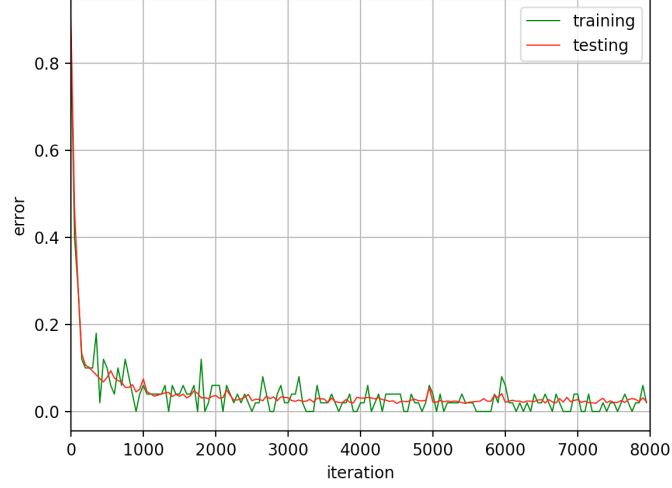


Figure 10: Plot of training/testing error as a function of training iterations for **(1d)**.

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.

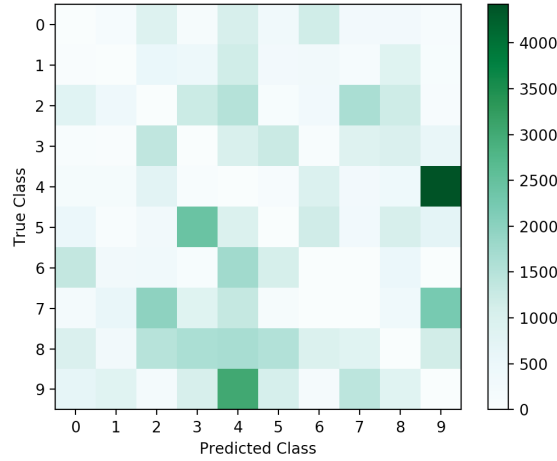


Figure 11: Confusion matrix over selected iterations for the test set for **(1d)**

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**).

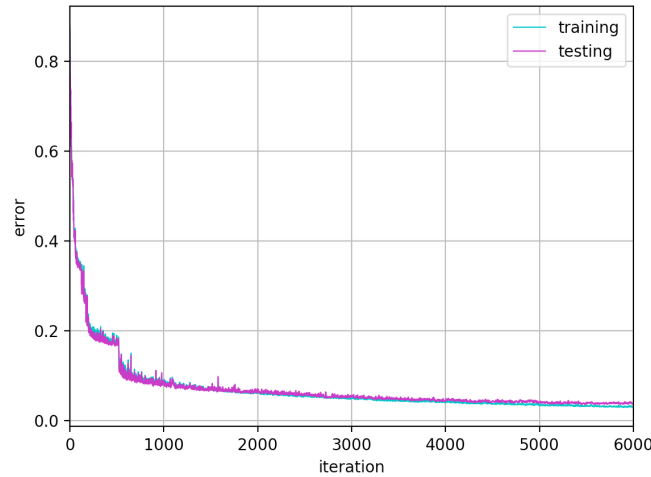


Figure 12: Plot of training (over entire set)/testing error as a function of training iterations for (**1b**).

2 MNIST without TensorFlow

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

$$\text{loss}_i = -\log \left(\frac{\exp(z_i[y_i])}{\sum_{c=1}^{10} \exp(z_i[c])} \right),$$

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,

$$\text{loss}_i = -\log(o_i[y_i]).$$

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

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

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

$$\begin{aligned} \frac{\partial o_j}{\partial z_i} &= o_i(1 - o_i), & i = j, \\ \frac{\partial o_j}{\partial z_i} &= -o_i o_j, & i \neq j, \end{aligned}$$

Then for the original derivative,

$$\begin{aligned} \frac{\partial \text{loss}}{\partial z_i} &= \sum_j \frac{\partial \text{loss}}{\partial o_j} \frac{\partial o_j}{\partial z_i} = - \sum_j \frac{\partial \log o_j}{\partial z_i} = - \sum_j \frac{1}{o_j} \frac{o_j}{z_i}, \\ &= -(1 - o_i) - \sum_{j \neq i} \frac{(-o_j o_i)}{o_j} = -(1 - o_i) + \sum_{j \neq i} o_i = o_i - \vec{1} \quad (i = j). \end{aligned}$$

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

(2aii) The derivative of the loss function with respect to the inputs x , $\frac{\partial \text{loss}}{\partial x}$ 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,

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

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 $\frac{\partial \text{loss}}{\partial z_k} = \sum_j \frac{\partial \text{loss}}{\partial o_j} \frac{\partial o_j}{\partial z_k} = o_k - \vec{1}$. Hence,

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

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

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

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

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

(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,

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

This expression can be presented as,

$$\frac{\partial o_{i'j'}^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial}{\partial W_{ij}^{(l)}} \left(\sum_{i''} \sum_{j''} W_{i''j''}^{(l)} A^{(l)}(o_{(i'-i'')(j'-j'')})^{(l-1)} + b^{(l)} \right),$$

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,

$$\frac{\partial o_{i'j'}^{(l)}}{\partial W_{ij}^{(l)}} = \frac{\partial}{\partial W_{ij}^{(l)}} \left(W_{ij}^{(l)} A^{(l)}(o_{(i'-i)(j'-j)})^{(l-1)} \right) = A^{(l)}(o_{(i'-i)(j'-j)})^{(l-1)},$$

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

$$\frac{\partial L}{\partial W_{ij}^{(l)}} = \sum_{i'} \sum_{j'} \delta_{i'j'}^{(l)} A^{(l)}(o_{(i'-i)(j'-j)})^{(l-1)} = \delta_{ij}^{(l)} * A^{(l)}(o_{(-i)(-j)})^{(l-1)}$$

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

$$\frac{\partial L}{\partial W_{ij}^{(l)}} = \delta_{ij}^{(l)} * A^{(l)}(rot_{180} o_{(ij)}^{(l-1)}).$$

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,

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

From the previous question, and since $\frac{\partial o_{i'j'}^{(l)}}{\partial x_{ij}^{(l)}} = W_{i'j'}^{(l)}$,

$$\frac{\partial L}{\partial x_{ij}^{(l)}} = \left(\delta_{ij}^{(l)} * A^{(l)}(rot_{180} o_{(ij)}^{(l-1)}) \right) \sum_{i'} \sum_{j'} W_{i'j'}^{(l)}.$$

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

Appendix

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

`exercisel1a.py`

```

1  """
2  Neural network model to classify MNIST digits consisting of 1 linear layer,
   followed by a
3  softmax.
4  """
5
6
7  # for compatibility
8  from __future__ import absolute_import
9  from __future__ import division
10 from __future__ import print_function
11
12 # data directory
13 data_dir = "../MNIST_data/"
14
15 # for saving the model

```



```

16 model_folder = "savedmodels/"
17 # question
18 subdirectory = "1a/"
19 model_filename = model_folder + subdirectory + "model_1a.ckpt"
20
21 # load and read MNIST data, import tensorflow
22 from tensorflow.examples.tutorials.mnist import input_data
23 # import dataset with one-hot class encoding
24 print("Loading the data.....")
25 mnist = input_data.read_data_sets(data_dir, one_hot=True)
26 print("Data has been loaded. ")
27 import tensorflow as tf
28
29 # import sklearn, matplotlib for confusion matrix generation functionality
30 import matplotlib.pyplot as plt
31 from sklearn.metrics import confusion_matrix
32
33 # import numpy for train/test accuracy/loss tracking
34 import numpy as np
35
36 # operating system interface to save cost/accuracy history.
37 import os
38
39 # set a random seed for reproducibility
40 tf.set_random_seed(555)
41
42 # define hyper-parameter values
43 n_attributes = 784
44 n_classes = 10
45 optimiser_step_size = 0.5
46 tot_training_iterations = 5000
47 minibatch_size = 50
48 print_update_every = 200
49
50 # one-hot encoding converted to single number class by storing
51 # index of highest element.
52 mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])
53
54 # placeholder for data x, (784 pixels/attributes)
55 # placeholder for label y
56 x = tf.placeholder(tf.float32, [None, n_attributes])
57 y = tf.placeholder(tf.float32, [None, n_classes])
58 # placeholder for true single number class (label)
59 y_klass = tf.placeholder(tf.int64, [None])
60
61 # neural network model building
62 # define variables - initialise weights, biases as tensors full of zeros
63 weights = tf.Variable(tf.zeros([n_attributes, n_classes]))
64 biases = tf.Variable(tf.zeros([n_classes]))
65 # linear regression layer
66 y_temp = tf.add(tf.matmul(x, weights), biases)
67 # softmax performed
68 y_pred = tf.nn.softmax(y_temp)
69
70 # one-hot encoding -> predicted numerical label (class), index of largest
    element in row
71 y_pred_klass = tf.argmax(y_pred, dimension = 1)
72 # model is trained using cross-entropy loss function

```

```

73 cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,
    logits=y_pred))
74 # stochastic gradient descent for minimising objective
75 optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(
    cost)
76 # define the accuracy – percentage of times correct prediction
77 correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
78 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
79 # error is 1-accuracy
80 error = 1 - accuracy
81
82 # save weights, biases for train.Saver
83 tf.add_to_collection('vars', weights)
84 tf.add_to_collection('vars', biases)
85
86 saver = tf.train.Saver()
87
88 with tf.Session() as sess:
89     # operation to initialise all variables
90     tf.global_variables_initializer().run()
91     print("=====")
92     print("Training started.....")
93     # initialisation of arrays keeping track of loss/error history
94     train_loss_history = np.zeros(tot_training_iterations)
95     test_loss_history = np.zeros(tot_training_iterations)
96     train_error_history = np.zeros(tot_training_iterations)
97     test_error_history = np.zeros(tot_training_iterations)
98     tot_confmat = np.zeros((n_classes, n_classes))
99     # loop over each training iteration
100    for iteration in range(tot_training_iterations):
101        # load 'minibatch_size' examples in each training iteration
102        xbatch, ybatch = mnist.train.next_batch(minibatch_size)
103        # error/loss for test and train calculated for each iteration
104        train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
105        train_error_history[iteration] = train_error
106        train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
107        train_loss_history[iteration] = train_loss
108        test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.
            test.labels})
109        test_error_history[iteration] = test_error
110        test_loss = cost.eval(feed_dict={x: mnist.test.images, y: mnist.test.
            labels})
111        test_loss_history[iteration] = test_loss
112        # print error/loss values every 'print_update_every' iteration to
            track progress
113        if iteration % print_update_every == 0:
114            print("iteration %d, training error %g" % (iteration, train_error
                ))
115            print("iteration %d, training loss %g" % (iteration, train_loss))
116            print("iteration %d, testing error %g" % (iteration, test_error))
117            print("iteration %d, testing loss %g" % (iteration, test_loss))
118        # saves error/loss history to directory
119        np.savez(os.getcwd() + "/traininghistory1a", train_loss_history=
            train_loss_history,
120                test_loss_history=test_loss_history,
121                train_error_history = train_error_history,
122                test_error_history = test_error_history)
123        # run optimiser for each batch – update gradients
124        optimiser.run(feed_dict={x: xbatch, y: ybatch})

```

```

125     # groups test set images, one-hot encoding, true numerical labels of
        test set
126     feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_class
        : mnist.test.class}
127     # test set true numerical classes (labels)
128     true_class = mnist.test.class
129     # test set predicted numerical classes (labels)
130     pred_class = sess.run(y_pred_class, feed_dict=feed_dict_test)
131     # sci-kit learn functionality to generate a confusion matrix
132     confmat = confusion_matrix(y_true=true_class, y_pred=pred_class)
133     tot_confmat = np.add(tot_confmat, confmat)
134     saver.save(sess, model_filename)
135     print("Training finished.")
136     print("=====\n")
137     # in final confusion matrix, the diagonal is set to zeros.
138     # correct classifications hugely outweigh errors and are ignored in the
        confusion matrix plot
139     for i in range(n_classes):
140         tot_confmat[i,i] = 0
141     #####
142     # confusion matrix plots
143     #####
144     # text print of confusion matrix
145     print(tot_confmat)
146     # image print of confusion matrix
147     plt.figure()
148     plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
149     # make plot nice
150     plt.colorbar()
151     tick_marks = np.arange(n_classes)
152     plt.xticks(tick_marks, range(n_classes))
153     plt.yticks(tick_marks, range(n_classes))
154     plt.xlabel('Predicted Class')
155     plt.ylabel('True Class')
156     plt.savefig('1aconfusionmatrix.png')
157     #####
158     # learning curve plots for cross entropy loss
159     #####
160     plt.figure()
161     traininghistory = np.load(os.getcwd() + "/traininghistory1a.npz")
162     trainingloss = traininghistory["train_loss_history"]
163     testingloss = traininghistory["test_loss_history"]
164     axisx = np.arange(tot_training_iterations)
165     plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
166     plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
167     plt.grid()
168     plt.legend()
169     plt.xlabel("iteration")
170     plt.ylabel("cross-entropy loss")
171     plt.xlim(0, tot_training_iterations)
172     plt.show()
173     plt.savefig('1alosslearningcurve.png')
174     ###
175     ### learning curve for error
176     ###
177     plt.figure()
178     traininghistory2 = np.load(os.getcwd() + "/traininghistory1a.npz")
179     trainingerror = traininghistory2["train_error_history"]
180     testingerror = traininghistory2["test_error_history"]

```

```

181     axisx2 = np.arange(tot_training_iterations)
182     plt.plot(axisx2, trainingerror, "c-", linewidth=0.8, label="training")
183     plt.plot(axisx2, testingerror, "m-", linewidth=0.8, label="testing")
184     plt.grid()
185     plt.legend()
186     plt.xlabel("iteration")
187     plt.ylabel("error")
188     plt.xlim(0, tot_training_iterations)
189     plt.show()
190     plt.savefig('1aerrorlearningcurve.png')
191
192
193 """
194 ### RESTORE MODEL AND OBTAIN FINAL ACCURACY
195
196 print("Restoring and testing model")
197 with tf.Session() as sess:
198     new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "
199     model_1a.ckpt.meta")
200     new_saver.restore(sess, tf.train.latest_checkpoint(model_folder +
201     subdirectory + './'))
202     all_vars = tf.get_collection('vars')
203     weights = all_vars[0]
204     biases = all_vars[1]
205     test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.
206     labels})
207     print("The final test error is %g" % (test_error))
208
209 """

```

exercise1b.py

```

1 """
2 Neural network model to classify MNIST digits consisting of 1 hidden layer
3 (128
4 units) with a ReLu non-linearity, followed by a softmax.
5 """
6 # for compatibility
7 from __future__ import absolute_import
8 from __future__ import division
9 from __future__ import print_function
10
11 # data directory
12 data_dir = "./MNIST_data/"
13
14 # for saving the model
15 model_folder = "savedmodels/"
16 # question
17 subdirectory = "1b/"
18 model_filename = model_folder + subdirectory + "model_1b.ckpt"
19
20 # load and read MNIST data, import tensorflow
21 from tensorflow.examples.tutorials.mnist import input_data
22 # import dataset with one-hot class encoding
23 print("Loading the data.....")
24 mnist = input_data.read_data_sets(data_dir, one_hot=True)
25 print("Data has been loaded. ")
26 import tensorflow as tf
27
28 # import sklearn, matplotlib for confusion matrix generation functionality

```

```

28 import matplotlib.pyplot as plt
29 from sklearn.metrics import confusion_matrix
30
31 # import numpy for train/test accuracy/loss tracking
32 import numpy as np
33
34 # operating system interface to save cost/accuracy history.
35 import os
36
37 # set a random seed for reproducibility
38 tf.set_random_seed(555)
39
40 # define hyper-parameter values
41 n_attributes = 784
42 n_classes = 10
43 optimiser_step_size = 0.5
44 tot_training_iterations = 6000
45 minibatch_size = 50
46 print_update_every = 200
47 # number of units in hidden layer 1
48 n_units_h1 = 128
49
50 # one-hot encoding converted to single number class by storing
51 # index of highest element.
52 mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])
53
54 # placeholder for data x, (784 pixels/attributes)
55 # placeholder for label y
56 x = tf.placeholder(tf.float32, [None, n_attributes])
57 y = tf.placeholder(tf.float32, [None, n_classes])
58 # placeholder for true single number class (label)
59 y_klass = tf.placeholder(tf.int64, [None])
60
61
62 # neural network model building
63 # define variables - initialise weights, biases
64 # hidden layer 1
65 W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="
    W_h1")
66 b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
67 # add ReLu non-linearity to hidden layer 1
68 hidden1 = tf.nn.relu(tf.matmul(x, W_h1) + b_h1)
69 # linear layer
70 W = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_classes]), name="W")
71 b = tf.Variable(tf.zeros([n_classes]), name="b")
72 # put together linear layer
73 z = tf.matmul(hidden1, W) + b
74 # softmax performed
75 y_pred = tf.nn.softmax(z)
76 # one-hot encoding -> predicted numerical label (class), index of largest
    element in row
77 y_pred_klass = tf.argmax(y_pred, dimension=1)
78 # model is trained using cross-entropy loss function
79 cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,
    logits=y_pred))
80 # 0.61 optimal
81 optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(
    cost)
82 # define the accuracy - percentage of times correct prediction

```

```

83 correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
84 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
85 # error is 1-accuracy
86 error = 1 - accuracy
87
88 # save final weights, biases for train.Saver
89 tf.add_to_collection('vars', W)
90 tf.add_to_collection('vars', b)
91
92 saver = tf.train.Saver()
93
94 # NEURAL NETWORK TRAINING COMMENCES
95 with tf.Session() as sess:
96     # operation to initialise all variables
97     tf.global_variables_initializer().run()
98     print("=====")
99     print("Training started .....")
100    # initialisation of arrays keeping track of loss/error history
101    train_loss_history = np.zeros(tot_training_iterations)
102    test_loss_history = np.zeros(tot_training_iterations)
103    train_error_history = np.zeros(tot_training_iterations)
104    test_error_history = np.zeros(tot_training_iterations)
105    tot_confmat = np.zeros((n_classes, n_classes))
106    # loop over each training iteration
107    for iteration in range(tot_training_iterations):
108        # load 'minibatch_size' examples in each training iteration
109        xbatch, ybatch = mnist.train.next_batch(minibatch_size)
110        # error/loss for test and train calculated for each iteration
111        train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
112        train_error_history[iteration] = train_error
113        train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
114        train_loss_history[iteration] = train_loss
115        test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.
            test.labels})
116        test_error_history[iteration] = test_error
117        test_loss = cost.eval(feed_dict={x: mnist.test.images, y: mnist.test.
            labels})
118        test_loss_history[iteration] = test_loss
119        # print error/loss values every 'print_update_every' iteration to
            track progress
120        if iteration % print_update_every == 0:
121            print("iteration %d, training error %g" % (iteration, train_error
                ))
122            print("iteration %d, training loss %g" % (iteration, train_loss))
123            print("iteration %d, testing error %g" % (iteration, test_error))
124            print("iteration %d, testing loss %g" % (iteration, test_loss))
125        # saves error/loss history to directory
126        np.savez(os.getcwd() + "/traininghistory1b", train_loss_history=
            train_loss_history,
127                test_loss_history=test_loss_history,
128                train_error_history=train_error_history,
129                test_error_history=test_error_history)
130        # run optimiser for each batch - update gradients
131        optimiser.run(feed_dict={x: xbatch, y: ybatch})
132        # groups test set images, one-hot encoding, true numerical labels of
            test set
133        feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_class
            : mnist.test.klass}
134        # test set true numerical classes (labels)

```

```

135     true_class = mnist.test.class
136     # test set predicted numerical classes (labels)
137     pred_class = sess.run(y_pred_class, feed_dict=feed_dict_test)
138     # sci-kit learn functionality to generate a confusion matrix
139     confmat = confusion_matrix(y_true=true_class, y_pred=pred_class)
140     tot_confmat = np.add(tot_confmat, confmat)
141 # saving
142 saver.save(sess, model_filename)
143 print("Training finished.")
144 print("=====\n")
145 # in final confusion matrix, the diagonal is set to zeros.
146 # correct classifications hugely outweigh errors and are ignored in the
    confusion matrix plot
147 for i in range(n_classes):
148     tot_confmat[i, i] = 0
149 #####
150 # confusion matrix plots
151 #####
152 # text print of confusion matrix
153 print(tot_confmat)
154 # image print of confusion matrix
155 plt.figure()
156 plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
157 # make plot nice
158 plt.colorbar()
159 tick_marks = np.arange(n_classes)
160 plt.xticks(tick_marks, range(n_classes))
161 plt.yticks(tick_marks, range(n_classes))
162 plt.xlabel('Predicted Class')
163 plt.ylabel('True Class')
164 plt.savefig('1bconfusionmatrix.png')
165 #####
166 # learning curve plots for loss/error
167 #####
168 plt.figure()
169 traininghistory = np.load(os.getcwd() + "/traininghistory1b.npz")
170 trainingloss = traininghistory["train_loss_history"]
171 testingloss = traininghistory["test_loss_history"]
172 axisx = np.arange(tot_training_iterations)
173 plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
174 plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
175 plt.grid()
176 plt.legend()
177 plt.xlabel("iteration")
178 plt.ylabel("cross-entropy loss")
179 plt.xlim(0, tot_training_iterations)
180 plt.show()
181 plt.savefig('1blosslearningcurve.png')
182 ###
183 ### learning curve for error
184 ###
185 plt.figure()
186 traininghistory2 = np.load(os.getcwd() + "/traininghistory1b.npz")
187 trainingerror = traininghistory2["train_error_history"]
188 testingerror = traininghistory2["test_error_history"]
189 axisx2 = np.arange(tot_training_iterations)
190 plt.plot(axisx2, trainingerror, "c-", linewidth=0.8, label="training")
191 plt.plot(axisx2, testingerror, "m-", linewidth=0.8, label="testing")
192 plt.grid()

```

```

193     plt.legend()
194     plt.xlabel("iteration")
195     plt.ylabel("error")
196     plt.xlim(0, tot_training_iterations)
197     plt.show()
198     plt.savefig('1berrorlearningcurve.png')
199
200 """
201
202 ### RESTORE MODEL AND OBTAIN FINAL ACCURACY
203
204 print("Restoring and testing model")
205 with tf.Session() as sess:
206     new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "
207     model_1b.ckpt.meta")
208     new_saver.restore(sess, tf.train.latest_checkpoint(model_folder +
209     subdirectory + './'))
210     all_vars = tf.get_collection('vars')
211     weights = all_vars[0]
212     biases = all_vars[1]
213     train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train
214     .labels})
215     test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.
216     labels})
217     print("The final train error is %g" % (train_error))
218     print("The final test error is %g" % (test_error))
219
220 """

```

exercise1c.py

```

1 """
2 Neural network model to classify MNIST digits consisting of 2 hidden layers
3 (256
4 units each) with a ReLu non-linearity, followed by a softmax.
5 """
6
7 # for compatibility
8 from __future__ import absolute_import
9 from __future__ import division
10 from __future__ import print_function
11
12 # data directory
13 data_dir = "./MNIST_data/"
14
15 # for saving the model
16 model_folder = "savedmodels/"
17 # question
18 subdirectory = "1c/"
19 model_filename = model_folder + subdirectory + "model_1c.ckpt"
20
21 # load and read MNIST data, import tensorflow
22 from tensorflow.examples.tutorials.mnist import input_data
23 # import dataset with one-hot class encoding
24 print("Loading the data.....")
25 mnist = input_data.read_data_sets(data_dir, one_hot=True)
26 print("Data has been loaded. ")
27 import tensorflow as tf
28
29 # import sklearn, matplotlib for confusion matrix generation functionality

```



```

28 import matplotlib.pyplot as plt
29 from sklearn.metrics import confusion_matrix
30
31 # import numpy for train/test accuracy/loss tracking
32 import numpy as np
33
34 # operating system interface to save cost/accuracy history.
35 import os
36
37 # set a random seed for reproducibility
38 tf.set_random_seed(555)
39
40 # define hyper-parameter values
41 n_attributes = 784
42 n_classes = 10
43 optimiser_step_size = 0.5
44 tot_training_iterations = 6000
45 minibatch_size = 50
46 print_update_every = 200
47 # number of units in hidden layer 1
48 n_units_h1 = 256
49 # number of units in hidden layer 2
50 n_units_h2 = 256
51
52 # one-hot encoding converted to single number class by storing
53 # index of highest element.
54 mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])
55
56 # placeholder for data x, (784 pixels/attributes)
57 # placeholder for label y
58 x = tf.placeholder(tf.float32, [None, n_attributes])
59 y = tf.placeholder(tf.float32, [None, n_classes])
60 # placeholder for true single number class (label)
61 y_klass = tf.placeholder(tf.int64, [None])
62
63 # neural network model building
64 # define variables - initialise weights, biases
65 # hidden layer 1
66 W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="
    W_h1")
67 b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
68 # add ReLu non-linearity to hidden layer 1
69 hidden1 = tf.nn.relu(tf.matmul(x, W_h1) + b_h1)
70 # hidden layer 2
71 W_h2 = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_units_h2]), name="
    W_h2")
72 b_h2 = tf.Variable(tf.random_normal([n_units_h2]), name="b_h2")
73 # add ReLu non-linearity to hidden layer 2
74 hidden2 = tf.nn.relu(tf.matmul(hidden1, W_h2) + b_h2)
75 # linear layer
76 W = tf.Variable(0.1 * tf.random_normal([n_units_h2, n_classes]), name="W")
77 b = tf.Variable(tf.zeros([n_classes]), name="b")
78 # put together linear layer
79 z = tf.matmul(hidden2, W) + b
80 # softmax performed
81 y_pred = tf.nn.softmax(z)
82 # one-hot encoding -> predicted numerical label (class), index of largest
    element in row
83 y_pred_klass = tf.argmax(y_pred, dimension=1)

```

```

84 # model is trained using cross-entropy loss function
85 cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,
    logits=y_pred))
86 # 0.61 optimal
87 optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(
    cost)
88 # define the accuracy - percentage of times correct prediction
89 correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
90 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
91 # error is 1-accuracy
92 error = 1 - accuracy
93
94 # save final weights, biases for train.Saver
95 tf.add_to_collection('vars', W)
96 tf.add_to_collection('vars', b)
97
98 saver = tf.train.Saver()
99
100 # NEURAL NETWORK TRAINING COMMENCES
101
102 with tf.Session() as sess:
103     # operation to initialise all variables
104     tf.global_variables_initializer().run()
105     print("=====")
106     print("Training started .....")
107     # initialisation of arrays keeping track of loss/error history
108     train_loss_history = np.zeros(tot_training_iterations)
109     test_loss_history = np.zeros(tot_training_iterations)
110     train_error_history = np.zeros(tot_training_iterations)
111     test_error_history = np.zeros(tot_training_iterations)
112     tot_confmat = np.zeros((n_classes, n_classes))
113     # loop over each training iteration
114     for iteration in range(tot_training_iterations):
115         # load 'minibatch_size' examples in each training iteration
116         xbatch, ybatch = mnist.train.next_batch(minibatch_size)
117         # error/loss for test and train calculated for each iteration
118         train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
119         train_error_history[iteration] = train_error
120         train_loss = cost.eval(feed_dict={x: xbatch, y: ybatch})
121         train_loss_history[iteration] = train_loss
122         test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.
            test.labels})
123         test_error_history[iteration] = test_error
124         test_loss = cost.eval(feed_dict={x: mnist.test.images, y: mnist.test.
            labels})
125         test_loss_history[iteration] = test_loss
126         # print error/loss values every 'print_update_every' iteration to
            track progress
127         if iteration % print_update_every == 0:
128             print("iteration %d, training error %g" % (iteration, train_error)
                )
129             print("iteration %d, training loss %g" % (iteration, train_loss))
130             print("iteration %d, testing error %g" % (iteration, test_error))
131             print("iteration %d, testing loss %g" % (iteration, test_loss))
132         # saves error/loss history to directory
133         np.savez(os.getcwd() + "/traininghistory1C", train_loss_history=
            train_loss_history,
134                 test_loss_history=test_loss_history,
135                 train_error_history=train_error_history,

```

```

136         test_error_history=test_error_history)
137     # run optimiser for each batch – update gradients
138     optimiser.run(feed_dict={x: xbatch, y: ybatch})
139     # groups test set images, one-hot encoding, true numerical labels of
        test set
140     feed_dict_test = {x: mnist.test.images, y: mnist.test.labels, y_class
        : mnist.test.class}
141     # test set true numerical classes (labels)
142     true_class = mnist.test.class
143     # test set predicted numerical classes (labels)
144     pred_class = sess.run(y_pred_class, feed_dict=feed_dict_test)
145     # sci-kit learn functionality to generate a confusion matrix
146     confmat = confusion_matrix(y_true=true_class, y_pred=pred_class)
147     tot_confmat = np.add(tot_confmat, confmat)
148     # saving
149     print("Training finished.")
150     print("=====\n")
151     saver.save(sess, model_filename)
152     # in final confusion matrix, the diagonal is set to zeros.
153     # correct classifications hugely outweigh errors and are ignored in the
        confusion matrix plot
154     for i in range(n_classes):
155         tot_confmat[i, i] = 0
156     #####
157     # confusion matrix plots
158     #####
159     # text print of confusion matrix
160     print(tot_confmat)
161     # image print of confusion matrix
162     plt.figure()
163     plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
164     # make plot nice
165     plt.colorbar()
166     tick_marks = np.arange(n_classes)
167     plt.xticks(tick_marks, range(n_classes))
168     plt.yticks(tick_marks, range(n_classes))
169     plt.xlabel('Predicted Class')
170     plt.ylabel('True Class')
171     plt.savefig('confmat1c.png')
172     #####
173     # learning curve plots for loss/error
174     #####
175     plt.figure()
176     traininghistory = np.load(os.getcwd() + "/traininghistory1c.npz")
177     trainingloss = traininghistory["train_loss_history"]
178     testingloss = traininghistory["test_loss_history"]
179     axisx = np.arange(tot_training_iterations)
180     plt.plot(axisx, trainingloss, "g-", linewidth=0.8, label="training")
181     plt.plot(axisx, testingloss, "r-", linewidth=0.8, label="testing")
182     plt.grid()
183     plt.legend()
184     plt.xlabel("iteration")
185     plt.ylabel("cross-entropy loss")
186     plt.xlim(0, tot_training_iterations)
187     plt.show()
188     plt.savefig('1closslearningcurve.png')
189     ###
190     ### learning curve for error
191     ###

```

```

192 plt.figure()
193 traininghistory2 = np.load(os.getcwd() + "/traininghistory1c.npz")
194 trainingerror = traininghistory2["train_error_history"]
195 testingerror = traininghistory2["test_error_history"]
196 axisx2 = np.arange(tot_training_iterations)
197 plt.plot(axisx2, trainingerror, "c-", linewidth=0.8, label="training")
198 plt.plot(axisx2, testingerror, "m-", linewidth=0.8, label="testing")
199 plt.grid()
200 plt.legend()
201 plt.xlabel("iteration")
202 plt.ylabel("error")
203 plt.xlim(0, tot_training_iterations)
204 plt.show()
205 plt.savefig('1cerrorlearningcurve.png')
206
207 """
208
209
210 ### RESTORE MODEL AND OBTAIN FINAL ACCURACY
211
212 print("Restoring and testing model")
213 with tf.Session() as sess:
214     new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "
215         model_1c.ckpt.meta")
216     new_saver.restore(sess, tf.train.latest_checkpoint(model_folder +
217         subdirectory + './'))
218     all_vars = tf.get_collection('vars')
219     weights = all_vars[0]
220     biases = all_vars[1]
221     train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train
222         .labels})
223     test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.
224         labels})
225     print("The final train error is %g" % (train_error))
226     print("The final test error is %g" % (test_error))
227
228 """

```

exerciseld.py

```

1 """
2 Neural network model to classify MNIST digits consisting of a 3 layer
3 convolutional
4 model (2 convolutional layers followed by max pooling), followed by one non-
5 linear
6 layer (256 units) followed by a softmax.
7 """
8
9 # for compatibility
10 from __future__ import absolute_import
11 from __future__ import division
12 from __future__ import print_function
13
14 # data directory
15 data_dir = "./MNIST_data/"
16
17 # for saving the model
18 model_folder = "savedmodels/"
19
20 # question
21 subdirectory = "1d/"
22 model_filename = model_folder + subdirectory + "model_1d.ckpt"

```

```

19
20 # load and read MNIST data, import tensorflow
21 from tensorflow.examples.tutorials.mnist import input_data
22 # import dataset with one-hot class encoding
23 print("Loading the data.....")
24 mnist = input_data.read_data_sets(data_dir, one_hot=True)
25 print("Data has been loaded. ")
26 import tensorflow as tf
27
28 # import sklearn, matplotlib for confusion matrix generation functionality
29 import matplotlib.pyplot as plt
30 from sklearn.metrics import confusion_matrix
31
32 # import numpy for train/test accuracy/loss tracking
33 import numpy as np
34
35 # operating system interface to save cost/accuracy history.
36 import os
37
38 # set a random seed for reproducibility
39 tf.set_random_seed(555)
40
41 # define hyper-parameter values
42 n_attributes = 784
43 n_classes = 10
44 optimiser_step_size = 0.5
45 tot_training_iterations = 8000
46 minibatch_size = 50
47 print_update_every = 50
48 # number of units in hidden layer 1
49 n_units_h1 = 256
50
51 # one-hot encoding converted to single number class by storing
52 # index of highest element.
53 mnist.test.klass = np.array([lbl.argmax() for lbl in mnist.test.labels])
54
55 # placeholder for data x, (784 pixels/attributes)
56 # placeholder for label y
57 x = tf.placeholder(tf.float32, [None, n_attributes])
58 y = tf.placeholder(tf.float32, [None, n_classes])
59 # placeholder for true single number class (label)
60 y_klass = tf.placeholder(tf.int64, [None])
61
62 # function which initialises (and returns) bias variables,
63 # it intakes the specified shape 'config' of the variables and creates
64 # a tensor of bias values populated with 0.1s (FOR LINEAR LAYER)
65 def create_biases(config):
66     init = tf.constant(0.1, shape=config)
67     return tf.Variable(init)
68
69 # function which initialises (and returns) weight variables.
70 # it intakes the specified shape 'config' of the variables and creates
71 # a tensor of weights from a truncated normal distribution (SD 0.1)
72 def create_weights(config):
73     init = tf.truncated_normal(config, stddev=0.1)
74     return tf.Variable(init)
75
76 # function which intakes input x and filter W and returns and computes
77 # their 2D convolution, which is returned - 'SAME' padding and plain (no

```

```

        strides)
78 def convolution(x, weights):
79     return tf.nn.conv2d(x, weights, strides=[1, 1, 1, 1], padding='SAME')
80
81 # function which intakes input x and performs max pooling on it with 2x2
    sliding
82 # windows – no overlapping pixels.
83 def pooling(x):
84     return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
        padding='SAME')
85
86 # neural network model building
87 # define variables – initialise weights, biases
88 # convolution layer 1 w/b
89 Wconvol1 = create_weights([3, 3, 1, 16])
90 bconvol1 = create_biases([16])
91 # before applying layer 1, reshape x to 4d tensor
92 reshaped1 = tf.reshape(x, [-1, 28, 28, 1])
93 # apply convolution layer 1
94 hiddenconv1 = tf.add(convolution(reshaped1, Wconvol1), bconvol1)
95 # apply max pooling 1
96 pooling1 = pooling(hiddenconv1)
97 # convolution layer 2 w/b
98 Wconvol2 = create_weights([3, 3, 16, 16])
99 bconvol2 = create_biases([16])
100 # apply convolution layer 2
101 hiddenconv2 = tf.add(convolution(pooling1, Wconvol2), bconvol2)
102 # apply max pooling 2
103 pooling2 = pooling(hiddenconv2)
104 # FLATTEN
105 reshapeflat = tf.reshape(pooling2, [-1, 7*7*16])
106 # hidden layer
107 W_h1 = tf.Variable(0.1 * tf.random_normal([n_attributes, n_units_h1]), name="
    W_h1")
108 b_h1 = tf.Variable(tf.random_normal([n_units_h1]), name="b_h1")
109 # add ReLu non-linearity to hidden layer 1
110 hidden1 = tf.nn.relu(tf.matmul(reshapeflat, W_h1) + b_h1)
111 # linear layer
112 W = tf.Variable(0.1 * tf.random_normal([n_units_h1, n_classes]), name="W")
113 b = tf.Variable(tf.zeros([n_classes]), name="b")
114 # put together linear layer
115 z = tf.add(tf.matmul(hidden1, W), b)
116 # softmax performed
117 y_pred = tf.nn.softmax(z)
118 # term definitions
119 # one-hot encoding -> predicted numerical label (class), index of largest
    element in row
120 y_pred_klass = tf.argmax(y_pred, dimension=1)
121 # model is trained using cross-entropy loss function
122 cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,
    logits=y_pred))
123 # run optimiser
124 optimiser = tf.train.GradientDescentOptimizer(optimiser_step_size).minimize(
    cost)
125 # define the accuracy – percentage of times correct prediction
126 correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
127 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
128 # error is 1-accuracy
129 error = 1 - accuracy

```

```

130
131 # save final weights, biases for train.Saver
132 tf.add_to_collection('vars', W)
133 tf.add_to_collection('vars', b)
134
135 saver = tf.train.Saver()
136
137 # NEURAL NETWORK TRAINING
138 with tf.Session() as sess:
139     sess.run(tf.global_variables_initializer())
140     # operation to initialise all variables
141     tf.global_variables_initializer().run()
142     print("=====")
143     print("Training started .....")
144     # initialisation of arrays keeping track of loss/error history
145     train_error_history = []
146     test_error_history = []
147     selected = []
148     tot_confmat = np.zeros((n_classes, n_classes))
149     # loop over each training iteration
150     for iteration in range(tot_training_iterations):
151         # load 'minibatch_size' examples in each training iteration
152         xbatch, ybatch = mnist.train.next_batch(minibatch_size)
153
154         if iteration % print_update_every == 0:
155             # error/loss for test and train calculated for each iteration
156             train_error = error.eval(feed_dict={x: xbatch, y: ybatch})
157             train_error_history.append(train_error)
158             test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist
159                                     .test.labels})
160             test_error_history.append(test_error)
161             selected.append(iteration)
162             print("iteration %d, training error %g" % (iteration, train_error))
163             print("iteration %d, testing error %g" % (iteration, test_error))
164             # saves error/loss history to directory
165             np.savez(os.getcwd() + "/traininghistory1d",
166                     train_error_history=train_error_history,
167                     test_error_history=test_error_history)
168             # print error/loss values every 'print_update_every' iteration
169             # to track progress
170             # groups test set images, one-hot encoding, true numerical labels
171             # of test set
172             feed_dict_test = {x: mnist.test.images, y: mnist.test.labels,
173                              y_klass: mnist.test.klass}
174             # test set true numerical classes (labels)
175             true_klass = mnist.test.klass
176             # test set predicted numerical classes (labels)
177             pred_klass = sess.run(y_pred_klass, feed_dict=feed_dict_test)
178             # sci-kit learn functionality to generate a confusion matrix
179             confmat = confusion_matrix(y_true=true_klass, y_pred=pred_klass)
180             tot_confmat = np.add(tot_confmat, confmat)
181             # run optimiser for each batch - update gradients
182             optimiser.run(feed_dict={x: xbatch, y: ybatch})
183             print("Training finished.")
184             print("=====\n")
185             # save model
186             saver.save(sess, model_filename)
187             # in final confusion matrix, the diagonal is set to zeros.

```

```

184     # correct classifications hugely outweigh errors and are ignored in the
        confusion matrix plot
185     for i in range(n_classes):
186         tot_confmat[i, i] = 0
187     #####
188     # confusion matrix plots
189     #####
190     # text print of confusion matrix
191     print(tot_confmat)
192     # image print of confusion matrix
193     plt.figure()
194     plt.imshow(tot_confmat, interpolation='none', cmap=plt.cm.BuGn)
195     # make plot nice
196     plt.colorbar()
197     tick_marks = np.arange(n_classes)
198     plt.xticks(tick_marks, range(n_classes))
199     plt.yticks(tick_marks, range(n_classes))
200     plt.xlabel('Predicted Class')
201     plt.ylabel('True Class')
202     plt.savefig('confmat1d.png')
203     #####
204     # learning curve plots for loss/error
205     #####
206     plt.figure()
207     model_history = np.load(os.getcwd() + "/traininghistory1d.npz")
208     train_error = model_history["train_error_history"]
209     test_error = model_history["test_error_history"]
210     x_axis = np.arange(0, tot_training_iterations, print_update_every)
211     plt.plot(x_axis, train_error, "g-", linewidth=0.8, label="training")
212     plt.plot(x_axis, test_error, "r-", linewidth=0.8, label="testing")
213     plt.grid()
214     plt.legend()
215     plt.xlabel("iteration")
216     plt.ylabel("error")
217     plt.xlim(0, tot_training_iterations)
218     plt.show()
219     plt.savefig('1derrorlearningcurve.png')
220     ###
221     ###
222     ###
223
224
225     """
226
227
228     ### RESTORE MODEL AND OBTAIN FINAL ACCURACY
229
230     print("Restoring and testing model")
231     with tf.Session() as sess:
232         new_saver = tf.train.import_meta_graph(model_folder + subdirectory + "
            model_1d.ckpt.meta")
233         new_saver.restore(sess, tf.train.latest_checkpoint(model_folder +
            subdirectory + './'))
234         all_vars = tf.get_collection('vars')
235         weights = all_vars[0]
236         biases = all_vars[1]
237         train_error = error.eval(feed_dict={x: mnist.train.images, y: mnist.train
            .labels})
238         test_error = error.eval(feed_dict={x: mnist.test.images, y: mnist.test.

```



```
        labels})
239     print("The final train error is %g" % (train_error))
240     print("The final test error is %g" % (test_error))
241
242     """
```