University of Toronto Mississauga
Department of Mathematical and Computational Sciences
CSC 411 - Machine Learning and Data Mining, Fall 2017

Assignment 3

Due date: Monday December 4, 11:59pm.
No late assignments will be accepted.

As in all work in this course, 20% of your grade is for quality of presentation, including the use of good English, properly commented and easy-to-understand programs, and clear proofs. In general, short, simple answers are worth more than long, complicated ones. Unless stated otherwise, all answers should be justified. The TA has a limited amount of time to devote to each assignment, so what you hand in should be legible (either typed or *neatly* hand-written), well-organized and easy to evaluate. (An employer would demand no less.) All computer problems are to be done in Python with the NumPy, SciPy and scikit-learn libraries.

Hand in four files: the source code of all your Python programs, a pdf file of all the requested program output, answers to all the non-programming questions (scanned handwriting is fine), and a scanned, signed copy of the cover sheet at the end of the assignment. Be sure to indicate clearly which question(s) each program and piece of output refers to. All the Python code (functions and script) for a given question should appear in one location in your source file, along with a comment giving the question number. All material in all files should appear in order; *i.e.*, material for Question 1 before Question 2 before Question 3, etc. It should be easy for the TA to identify the material for each question. In particular, all figures should be titled, and all printed output should be identified with the Question number. The four files should be submitted electronically as described on the course web page. In addition, if we run your source file, it should not produce any errors, it should produce all the output that you hand in (figures and print outs), and it should be clear which question each piece of output refers to. *Output that is not labled with the Question number will not be graded.*

Style: Use the solutions to Assignments 1 and 2 as a guide/model for how to present your solutions to Assignment 3.

I don't know policy: If you do not know the answer to a question (or part), and you write "I don't know", you will receive 20% of the marks of that question (or part). If you just leave a question blank with no such statement, you get 0 marks for that question.

No more questions will be added.

Notes on Scientific Programming in Python: For generating plots, you may find the SciPy functions `plot` and `figure` in `matplotlib.pyplot` useful. Also, if `A` and `B` are two-dimensional numpy arrays, then `A*B` performs *element-wise* multiplication, *not* matrix multiplication. To perform matrix multiplication, you can use `numpy.matmul(A,B)`. Also, whenever possible, *do not use loops*, which are very slow in Python. Instead, use Numpy's vector and matrix operations, which are much faster and can be executed in parallel. If you haven't already done so, please read the NumPy tutorial on the course web page. Be sure to read about slicing and indexing Numpy arrays (*e.g.*, if A is a matrix, then A[:,5] returns the 5th column).

1. (10 points total) *Neural Networks: decision boundaries.*

   This warm-up exercise introduces the process of defining and training a neural network, and illustrates the non-linear decision boundaries they produce. For this purpose, you will use one of the many synthetic data sets that Scikit-learn provides. In particular, you will train and test a simple neural net on Moons data, which has two-dimensional input points belonging to two classes.

   You will need to import the following Python modules:

   ```
   from sklearn import datasets
   from sklearn.neural_network import MLPClassifier
   import matplotlib.pyplot as plt
   import numpy as np
   import bonnerlib2
   ```

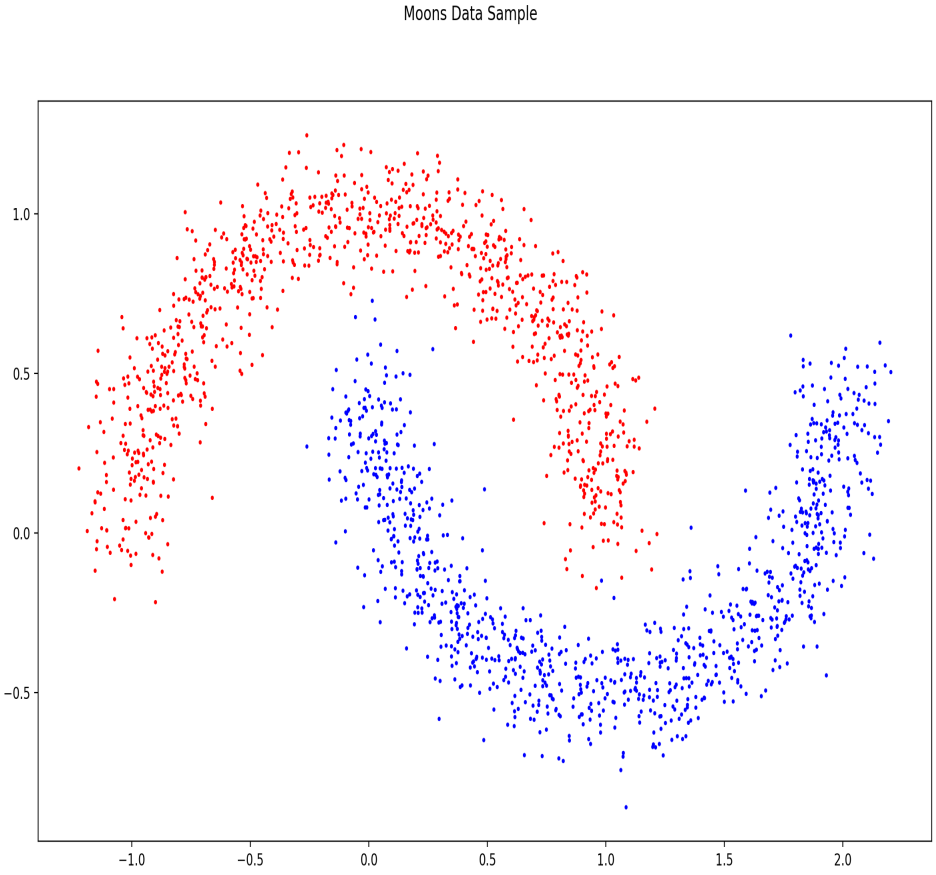
   The command below then creates a random sample of Moons data. Here, `X` is a matrix of 2-dimensional data points, and `t` is a vector of corresponding class labels. Each label is either 0 or 1. Figure 1 shows a scatter plot of the 2000 data points created by this command.

   ```
   X,t = datasets.make_moons(n_samples=2000, noise=0.1)
   ```

   The following Python command defines a simple neural net with one hidden layer containing 3 neurons (hidden units). (MLP stands for Multi-Layer Perceptron, which is another name for a neural net.) The hidden layer uses a tanh activation function. During training, stochastic gradient descent (sgd) will be used as the optimizer with a fixed learning rate of 0.01. Training will stop when the loss function changes by less than 10^{-20} or after 10,000 iterations, whichever comes first.

   ```
   clf = MLPClassifier(hidden_layer_sizes=[3],
                       activation='tanh',
                       solver='sgd',
                       learning_rate_init=0.01,
                       tol=10.0**(-20),
                       max_iter=10000)
   ```

Figure 1:



If you want to monitor the learning, add the keyword argument `verbose=True`. The command `clf.fit(X,t)` fits the neural net to the data set `X,t`.

What to do:

(a) (3 points) Generate a training set of 200 Moons data points, and a test set of 10,000 Moons data points. Use `noise=0.2` for both of them. Generate a scatter plot of each data set, and title the two plots, "Figure 1, Question 1(a): Moons training data" and "Figure 2, Question 1(a): Moons test data", respectively. Use a dot size of `s=2` in both scatter plots.

(b) (7 points) Write a Python program `fitMoons()` that does the following:

- Defines a neural net using the code fragment above.
- Fits the neural net to your Moons training set.
- Computes and prints out the error of the fitted net on your Moons test set. The error is the proportion of misclassified data samples. (This is one minus the accuracy, as computed by `clf.score`.)
- Displays a contour plot of the decision function of the neural net superimposed on a scatter plot of the training data (as in Assignment 2). Note that the black contour is the decision boundary.

Your program should repeat the above items nine times. Each time, it defines and trains a new neural net. Note that each training session starts from a different, random set of weights, and produces a different set of final weights, with a different decision boundary and a different test error. So, the nine neural nets are all different. Often the differences will be slight, but sometimes the optimizer will converge to a local minimum that is not near the global minimum, and the differences will be great. (The Moons data set results in some egregious local minima.)

All nine contour plots should be arranged on a square grid in a single figure. The figure should be titled, "Figure 3, Question 1(b): contour plots for various training sessions." Your program should also keep track of which neural net produces the smallest test error. Print this test error separately, and display the corresponding contour plot (with scatter plot) in a separate figure. Title the figure, "Figure 4, Question 1(b): contour plot for best training session."

Execute your program. Some of the printed test errors should be greater than 0.1 and some should be less than 0.04. If not, execute the program again. (If you never get such errors, you may have an unlucky training set. Try generating a new one.) Hand in the figures and printed values.

2. (31 points total) *Neural Networks: Theory.*

Suppose we wish to use a neural network for multi-class classification. In this case, the problem is to decide which class, $C_1, C_2, ..., C_K$, a given data point belongs to. Let x be the data point, let t be a vector representing the true/target class of x, and let o

be a vector representing the predicted class of x. Thus, $t = (t_1, t_2, ..., t_K)$ where $t_k = 1$ if C_k is the true/target class of x, and $t_k = 0$ otherwise. Likewise, $o = (o_1, o_2, ..., o_K)$ where $o_k = P(C_k = 1|x)$ is the predicted probability that point x belongs to class C_k. The input to the neural net is x, and the output is o. More specifically, the output consists of K neurons that implement a softmax function, and o_k is the value of the k^{th} output neuron:

$$o_k \;=\; e^{z_k} / \sum_i e^{z_i} \qquad \text{where} \;\; z_k = \sum_{j=1}^{J} w_{kj} h_j + w_{k0}$$

Here, h_j is the value of the j^{th} hidden unit in the top hidden layer of the net, and w_{kj} is the weight on the edge from hidden unit j to output unit k. w_{k0} is the bias term for output unit k.

We use cross entropy as the loss function. That is, for a single training instance,

$$l(o, t) \;=\; -\sum_k t_k \log(o_k)$$

and for an entire training set,

$$\mathcal{L}(w, w_0) \;=\; \sum_n l(o^{(n)}, t^{(n)}) \tag{1}$$

where the sum is over training samples. Here, $t^{(n)}$ is the true/target output for training point $x^{(n)}$, and $o^{(n)}$ is the output of the neural net on input $x^{(n)}$, that is, $o_k^{(n)} = P(C_k = 1|x^{(n)})$. The goal of training is to find w and w_0 to minimize $\mathcal{L}(w, w_0)$. In general, w includes all the weights in all the layers of the neural net, and w_0 includes all the bias terms. However, for the purpose of this question, w and w_0 will consist of just the weights and biases between the top hidden layer and the output layer. Thus, w is a matrix and w_0 is a vector. We will also let $h^{(n)} = [h_1^{(n)}, h_2^{(n)}, ..., h_J^{(n)}]$ be the vector of hidden values in the top hidden layer when the input to the network is $x^{(n)}$.

What to do:

(a) (4 points) Prove that $\partial o_k / \partial z_j = -o_k o_j$ for $k \neq j$.

(b) (5 points) Prove that $\partial o_k / \partial z_k = (1 - o_k) o_k$.

(c) (3 points) Derive simple formulas for $\partial z_i / \partial w_{kj}$ for all i.

(d) (10 points) Prove that for $j \geq 1$,

$$\frac{\partial \mathcal{L}(w, w_0)}{w_{kj}} \;=\; \sum_n [o_k^{(n)} - t_k^{(n)}] h_j^{(n)}$$

Hint: Recall that $t = (t_1, t_2, ..., t_K)$ where one of the $t_k = 1$ and the rest are 0.

(e) (3 points) Without going through a separate proof, argue that

$$\frac{\partial \mathcal{L}(w, w_0)}{\partial w_{k0}} \;=\; \sum_n [o_k^{(n)} - t_k^{(n)}]$$

(f) (4 points) Let H be a $N \times J$ matrix of hidden values whose n^{th} row is the vector $h^{(n)}$. Also, let T and O be $N \times K$ matrices, where the n^{th} row of T is the vector $t^{(n)}$, and the n^{th} row of O is the vector $o^{(n)}$. Using the results from part (d), prove that

$$\frac{\partial \mathcal{L}(w, w_0)}{\partial w} \;=\; (O - T)^T H$$

Here $\partial \mathcal{L}(w, w_0)/\partial w$ is a matrix with the same dimensions as w, where the kj^{th} entry is $\partial \mathcal{L}(w, w_0)/\partial w_{kj}$.

(g) (2 points) Using the results from part (e), derive a simple formula for $\partial \mathcal{L}(w, w_0)/\partial w_0$.

3. (20 points total) *Neural Networks: real data.*

In Assignment 2, you trained logistic-regression on the MNIST data and achieved a test error of about 8% (i.e., an accuracy of 92%). In this question, you will train a neural net on the same data with the goal of achieving a test error of less than 2.5%. To achieve this, you will have to choose hyperparameters (such as learning rate and momentum) that cause the optimizer to converge to a good minimum in a reasonable amount of time without over or underfitting the data. This will involve some experimentation on your part with different hyperparameter values. In addition, to monitor the training, you will compute and print out the training and test error every so often. This will require you to stop and restart the training periodically.

To begin, define a neural net with a single hidden layer of 100 neurons that use a tanh activation function. Use stochastic gradient descent as the optimizer. In addition, add the following key-word arguments to the definition of your neural net:

```
batch_size=200,
max_iter=5,
warm_start=True,
tol=0.0,
learning_rate_init=??,
momentum=??,
alpha=??
```

The line `batch_size=200` means that each step of gradient descent (gradient computions and weight updates) uses only 200 training points. (These points are called a *mini-batch*.) It thus takes 300 steps of gradient descent to go through all 60,000 MNIST training points. (Each such pass through the training data is called an *epoch*.) The line `max_iter=5` means that each time `clf.fit` is called, training will stop after (at most) 5 epochs. (It could stop sooner if the optimizer decides that the minimum has been found.) You can then compute and print out the training and test error. To restart training, simply call `clf.fit` again. Normally, this would cause training to begin from scratch with newly initialized, random weights. However, the line `warm_start=True` prevents this reinitialization.

Part of your job is to choose good values for the last three lines: learning rate, momentum and alpha. Recall that the learning rate is non-negative, and the momentum is between 0 and 1 (with 0 giving ordinary gradient descent). Alpha is the weight-decay coefficient. It is a non-negative number that encourages weights to be small. (Larger alpha encourages smaller weights.) Small values of alpha can lead to overfitting, and large values to underfitting.

To speed up learning, it often helps to preprocess the data so that each image pixel has a mean of 0 and a variance of 1. This is called *normalization*. This is done by applying a linear transformation to the data. In particular, if $x = (x_1, x_2, ..., x_D)$ is a data point, then x_i is transformed to $(x_i - \mu_i)/\sigma_i$, where μ_i is the mean value of feature i over all the training data, and σ_i is the standard deviation of feature i over all the training data. This operation is so common that `sklearn` provides special facilities for it. Here is how to generate a scaling operation from a given data matrix `X1` and how to apply it to another data matrix `X2` to produce a transformed data matrix `X3`:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X1)
X3 = scaler.transform(X2)
```

To normalize the training and test data, you generate a scaling operation from the training data, and then apply it to the training data and to the testing data. (Note that the same scaling operation is applied to both data sets. In particular, do not generate a separate scaling operation from the test data.) Also, notice that the target values, `t`, remain unchanged.

In the questions below, you will be using some of your code from Assignment 2. Hand it in with your code from this assignment (so we can run your programs), but clearly identify which code comes from Assignment 2.

What to do:

(a) (2 points) Read in the MNIST training and test data and flatten it as in Assignment 2. Randomly shuffle the flattened training data. This will ensure that each mini-batch is a random sample of training points. Note that you will have to shuffle the images and the labels in the same way. You may find the function `shuffle` in `sklearn.utils` useful. (You do not need to shuffle the test data since it is not used for training.)

(b) (2 points) Normalize the flattened (and shuffled) training and test data as described above. Use this normalized data to train neural nets in the rest of this assignment.

(c) (1 point) Use your `displaySample` function from Assignment 2 to display a random sample of 16 normalized MNIST test digits. (They should look more ghostly than unnormalized digits, but still be recognizable.) Title the figure, "Question 3(c): some normalized MNIST digits."

(d) (5 points) Choose values for the hyperparameters learning rate, momentum and alpha. Use these values to define a neural net as described above. Train the neural net on the (normalized) MNIST data by repeatedly doing the following at most 50 times.

- Call `clf.fit` (for 5 epochs of training).
- Compute the training and test errors and print them out as a percent.

You should notice that the test error goes up and down, though it will tend in a downward direction and then tend to flatten out. For bad choices of the hyperparameters, the test error may tend to go up at some point.

(e) (5 points) Repeat part (d) for different choices of the three hyperparameters. Keep trying different choices until the test error goes below 2.5% at least twice during a given training run. Hand in the values printed out by your program for this run. If you can't get the test error to go below 2.5%, hand in your best run. In any case, you should find that the final training error is considerably less than the test error. Hand in the program code as well.

(f) (5 points) This question is intended to demonstrate the advantage of using small mini batches in stochastic gradient descent. Using the hyperparameters you found in part (e), run your program again, but in batch mode. That is, use `batch_size=60000`, the number of MNIST training points. Also, let the program go for 1000 iterations, instead of 50. (This may take several hours, so you might want to run it over night. This is very common in machine learning.) You should find that the test error never gets close to your best run in part (e). Plot the history of test error (in blue) and training error (in orange) on a single pair of axes. Label the vertical axis, "error", and the horizontal axis, "training iterations". The curves should drop on the left and flatten out on the right. Label the figure, "Figure 5, Question 3: training and test error in batch mode." In a separate figure, plot the test error of the last 500 iterations of training. You should find that it tends to decrease, but very slowly. Label this figure, "Figure 6, Question 3: test error during last 500 iterations of batch training". (Of course, depending on the precise values of your hyperparameters, you may find something a bit different. For example, the test error may eventually increase.) Hand in the two plots and your code.

4. (42 points total) *Stochastic Gradient Descent.*

In this question, you will use the theory developed in Question 2 to implement (part of) a program for neural-net training. There are three main parts to the program: forward inference, back propagation, and weight updating using (batch and stochastic) gradient descent. In this question, you will implement full forward inference, but a limited form of back propagation. In particular, you will only update the weights in the top layer of the neural net. Also, to simplify the problem, we will consider a net with just one hidden layer. During back propagation, you will compute gradients of the mean loss function, $\mathcal{L}/N$, where $\mathcal{L}$ is the sum defined in Equation (1), and N is the number of

terms in the sum. To test your programs, use the normalized (and shuffled) MNIST training and test data from Question 3. Except where otherwise specified, you should only use `numpy` functions in this question, and not any functions from `sklearn`.

(a) (5 points) *Forward Inference.*
Write a Python function `predict(X,W1,W2,b1,b2)` that computes the hidden and output values of a neural net given a set of input vectors. Here, `X` is a data matrix, where each row is an input vector. The neural net itself has a single hidden layer and a softmax output layer. The hidden layer uses a tanh activation function. `W1` and `b1` are the weight matrix and bias terms, respectively, for the first layer (inputs to hidden units), and `W2` and `b2` are the weight matrix and bias terms of the second layer (hidden units to outputs). The function should return two matrices, one for the hidden values, and one for the output values. Note that each output value is a probability.

(b) (3 points) *Testing.*
Use your `predict` function to compute the output of a neural net on the (normalized) MNIST test data. Use the weights and bias terms from the neural net that you learned in Question 3(e) in your best training run. You can retrieve the weights of the neural net using `clf.coefs_` and, the bias terms using `clf.intercepts_`. Using the MNIST test data as input, compare the values returned by your `predict` function to the output values of the neural net that you trained in Question 3(e) (given by `clf.predict_proba`). Specifically, compute $\sum_{ij}(O1_{ij} - O2_{ij})^2$, where $O1$ is the output matrix returned by `predict`, and $O2$ is the output matrix returned by `clf.predict_proba`. The result should be very small. Print out the sum and hand it in.

(c) (4 points) *Back Propagation.*
Write a Python function `gradient(H,Y,T)` that computes the gradient of the mean loss function wrt the weights and biases in the top layer of the neural net. Here, `H`, `Y` and `T` are matrices, in which each row corresponds to a particular input of the neural net. Each row of `H` is a vector of hidden values for the top-layer of the neural net, each row of `Y` is a vector of computed output values for the neural net, and each row of `T` is a vector of target values. The function should return a matrix `DW` and a vector `Db`, where `DW` is the gradient of the mean loss function with repect to `W2`, and `Db` is the gradient wrt `b2`.

(d) (10 points) *Batch Gradient Descent.*
Define a Python function `bgd(W1,b1,lrate,sigma,K)` that uses batch gradient descent to learn the weights and bias terms in the top layer of a neural net. Here, `W1` is the matrix of weights for the first layer of the neural net, and `b1` is the vector of bias terms for the first layer. These arguments partially specify the neural net. The remaining arguments are hyperparameters that specify the learning procecdure: `lrate` is the learning rate (a positive number), `sigma` is the standard deviation of the initial weights in the second layer, and K is the maximum number of steps of gradient descent to perform. Note that because this is batch training, each step of gradient descent (gradient computations and

weight updates) uses all the training data, so it is one epoch of training.

Call the weights and biases in the top layer of the neural net `W2` and `b2`, respectively. Your job is to learn them. You will need the number of output neurons, `M`. This can be figured out from the training data. Your program should do the following:

i. Initialize `W2` using random numbers from a Gaussian distribution with a mean of 0 and a standard deviation of `sigma`. You may find the function `randn` in `numpy.random` useful. Initialize `b2` to be all zeros.

ii. Repeatedly update `W2` and `b2` using `K` steps (epochs) of batch gradient descent.

iii. Every five epochs, compute and print out the iteration number, the mean training loss, and the training and test errors as percents.

iv. After the `K` epochs have finished, plot the history of training and test errors in a single graph. Use blue for the test error and orange for the training error. Label the axes appropriately, and label the figure, "Figure 7, Question 4(d): training and test error for batch gradient descent."

v. In a separate figure, plot the history of the mean loss. Label the axes appropriately, and label the figure, "Figure 8, Question 4(d): mean training loss for batch gradient descent."

vi. If `K` > 500, then plot the last 100 training and test errors in a single graph. Use blue for the test error and orange for the training error. Label the axes appropriately, and label the figure, "Figure 9, Question 4(d): training and test error for last 500 epochs of bgd."

vii. If `K` > 500, then plot the last 100 mean losses in a separate figure. Label the axes appropriately, and label the figure, "Figure 10, Question 4(d): mean training loss for last 500 epochs of bgd."

(e) (5 points) *Testing.*

Use your `bgd` program to train and test a neural net on the (normalized) MNIST data. Use the optimal learning rate that you found in Question 3(e), and use `sigma` $= 0.01$. Train the neural net for 1000 epochs. Hand in the four figures produced by the program. In the first two figures (showing the entire training history), the curves should drop sharply on the left, and then flatten out for the rest of the training history. In the last two figures (showing a close-up of the last half of the training history), the curves should trend downwards slowly, before possibly flattening out or even rising slowly. Also hand in the last ten lines printed out by your program (giving loss and errors). The final training error should be close to 3%.

(f) (5 points) *Weight Decay.*

Using the notation of Question 2, let $\mathcal{C} = \mathcal{L}/N + \alpha \sum_{kj \geq 1} w_{kj}^2/2$, where $\mathcal{L}$ is the sum defined in Equation (1) and N is the number of terms in the sum. Thus, we have added a weight-decay (or regularization) term to the mean loss. Derive simple formulas for the gradients $\partial \mathcal{C}/\partial w$ and $\partial \mathcal{C}/\partial w_0$.

(g) (5 points) *Stochastic Gradient Descent.*

Modify your function `bgd` in part (d) to peform stochastic gradient descent with

mini-batches, momentum and weight decay. Call the new function `sgd(W1,b1,lrate,alpha,sigma,K,batchSize,mom)`, where `mom` is the momentum (a number between 0 and 1), `batchSize` is the number of training points in a mini-batch, and `alpha` is the weight-decay coefficient, α. Note that to implement weight decay, you must minimize (and compute gradients of) $\mathcal{C}$ from part (f), instead of $\mathcal{L}/N$. Hint: add one feature at a time to `bgd`, and get it working before adding another. State clearly if you do not get all the features working.

Since the training data has been randomly shuffled, mini-batches can be chosen by sweeping across the training data from start to finish. That is, the first mini-batch is the first `batchSize` points in the training set. The second mini-batch is the next `batchSize` points. The third mini-batch is the next `batchSize` points, etc. (If the number of training points is not a multiple of `batchSize`, then the last mini-batch in a sweep will have fewer than `batchSize` points in it.) Each such sweep of the training data is an epoch. Your program should do `K` epochs of training.

Your `sgd` program should print out all the same values (losses and errors every five epochs) that `bgd` did. It should also generate the same first-two figures (the last two are not needed). The two figures should be labeled, "Figure 11, Question 4(g): training and test error for stochastic gradient descent," and, "Figure 12, Question 4(g): mean training loss for stochastic gradient descent." In addition, `sgd` should print out the minimum test error achieved during the `K` epochs of training.

(h) (5 points) *Testing.*
Use your `sgd` program to train and test a neural net on the (normalized) MNIST data. Use `alpha = 0.0001`, and use the same learning rate and value for `sigma` as in part (e). Do 50 epochs of training. Choose the mini-batch size and momentum so that the minimm test error is as low as possible. (You should be able to get close to 2.2%.) You should find that the curves of loss and error decrease on the left, and then tend to flatten out, though not as dramatically as in part (e). You should also find that the final training error is considerably less than the test error. Hand in the two figures produced by the program, and everything printed out by it. Also, report the batch size and momentum that you used to produce this output.

University of Toronto Mississauga
CSC 411 - Machine Learning and Data Mining

Cover sheet for Assignment 3

Complete this page and hand it in with your assignment.

Name: Sayantan Chattopadhyay
(Underline your last name)

Student number: 1000818145

I declare that the solutions to Assignment 3 that I have handed in are solely my own work, and they are in accordance with the University of Toronto Code of Behavior on Academic Matters.

Signature: