Assignment 3: Questions and Solutions

Due date: Wednesday 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 five files: The source code of all your programs (functions and script) in a single Python file, a pdf file of figures generated by the programs, a pdf file of all printed output, a pdf file of answers to all the non-programming questions (scanned hand-writing is fine, but not photographs), 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 five 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 labeled with the Question number will not be graded. Programs that do not produce output 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.
Tips on Scientific Programming in Python

If you haven’t already done so, please read the NumPy tutorial on the course web page. Be sure to read about indexing and slicing Numpy arrays. First, indexing begins at 0, not 1. Thus, if A is a matrix, then A[7,0] is the element in row 7 and column 0. Likewise, A[0,4] is the element in row 0 and column 4. Slicing allows large segments of an array to be referenced. For example, A[:,5] returns column 5 of matrix A, and A[7, [3,6,8]] returns elements 3, 6 and 8 of row 7. Similarly, if v is a vector, then the statement A[:,6]=v copies v into row 6 of matrix A. Note that if A and B are two-dimensional Numpy arrays, then A*B performs element-wise multiplication, not matrix multiplication. To perform matrix multiplication, you should use numpy.matmul(A,B). Whenever possible, do not use loops, which are very slow in Python. In particular, avoid iterating over the elements of a large vector or matrix. Instead, use numpy’s vector and matrix operations, which are much faster and can be executed in parallel. For example, if A is a matrix and v is a column vector, the A+v will add v to every column of A. Likewise for rows and row vectors. Also, the functions sum and mean in numpy are useful for summing or averaging over all or part of an array. Many NumPy functions that are defined for single numbers can be passed lists, vectors and matrices instead. For example, f([x1, x2, ..., xn]) returns the list [f(x1), f(x2), ..., f(xn)]. The same is true for many user-defined functions. The term numpy.inf represents infinity. It results from dividing by 0 in numpy. It can also result from overflow (i.e., from numbers that are too large to represent in the computer, like 10^{1000}). The term numpy.nan stands for “not a number”, and it results from doing 0/0, inf/inf or inf-inf in numpy. For generating and labelling plots, the following SciPy functions in matplotlib.pyplot are needed: plot, xlabel, ylabel, title, suptitle and figure. You can use Google to conveniently look up SciPy functions. e.g., you can google “numpy matmul” and “pyplot suptitle”.

Because they are very slow, you should avoid the use of loops and iteration in your Python programs, replacing them by Numpy matrix and vector operations wherever possible. For the same reason, you should not use recursion or higher-order functions (such as the python map function or any numpy function listed under “functional programming”, such as apply-along-axis, which are just just loops in disguise), unless otherwise specified.

In addition, if a part of a question prints any output, you should precede all code for that part with lines like the following:

```python
print(‘\n’)  
print(‘Question 1(d).’)  
print(‘----------’)  
```

You do not have to include these lines in your line-counts of code.
1. (30 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. You will use two-dimensional data similar to that of Question 1 of Assignment 2. To train a neural net, you will use the class `MLPClassifier` in `sklearn.neural_network`. As in Assignment 2, to plot the decision boundary, you should use the function `dfContour` from `bonnerlib2`, which you can download from the course web site.

To keep things simple, this question will only use neural nets with a single hidden layer. You should set your neural nets to use stochastic gradient descent (sgd) as the optimization method, and logistic for the hidden-layer activation function. Set the initial learning rate to 0.01, the tolerance for optimization to $10^{-8}$, and the maximum number of iterations to 1,000. When calling `MLPClassifier` it may be convenient to put each argument on a separate line, for readability. In such cases, the call to `MLPClassifier` will still count as only a single line of code.

In answering the questions below, you will need to use the methods and attributes of `MLPClassifier`. You should not use any loops in your programs, except a single loop when generating an array of subplots. You may borrow any code you like from Assignments 1 or 2, but you must include it in your line counts of code.

(a) (0 points) Using the function `gen_data` that you wrote in Assignment 2, generate a training set like that of Question 4(c) of Assignment 2. That is, `mu0 = (1,1), mu1 = (2,2), cov0 = 0, cov1 = 0.9, N0 = 1,000 and N1 = 500`. Generate a test set with `N0 = 10,000 and N2 = 5,000`. Use this data set in the rest of this question.

(b) (8 points total) Train a neural net with one unit in the hidden layer. Plot the training data, and draw the decision boundary on top of the training data using `dfContour`\(^1\). The x and y axes should both extend from -3 to 6. (Be sure to plot the training data and set the axes before you call `dfContour`.) Title the figure, “Question 1(b): Neural net with 1 hidden unit.” The decision boundary should be a straight line. In addition, print the accuracy, precision and recall of the neural net on the test data. You should write your own code to compute the precision and recall using only Numpy functions. You can do this question in at most 26 lines of code. (5 points)

*Explain why the decision boundary is a straight line* (3 points).

**ANSWER:** A neural net with one hidden unit is simply a logistic-regression classifier followed by a sigmoid unit. The sigmoid unit does not change the direction or shape of the decision boundary, just the decision threshold. Thus, since logistic regression has a linear decision boundary, so does the neural net.

\(^1\) `dfContour` plots the decision boundary as a black line and plots several other contours of $P(C = 1|x)$ for a classifier. The colour of a region represents the predicted probability that a point in the region is in class 1 or 0. The darker the blue, the greater the probability that $C = 1$. The darker the red, the greater the probability that $C = 0$. 

3
(c) (4 points) Train a neural net with two units in the hidden layer, plot the training data, and draw the decision boundary on top of the training data. Do this twelve times. You should observe twelve different decision boundaries. Arrange the twelve plots in a $4 \times 3$ grid in a single figure. Title the figure, “Question 1(c): Neural nets with 2 hidden units.” In a separate figure, plot the training data and decision boundary of the neural net with the highest test accuracy. Title the figure, “Question 1(c): Best neural net with 2 hidden units.” Print the accuracy, precision and recall of this net on the test data. The accuracy should be greater than in part (b). You can do this question in at most 26 lines of code.

(d) (4 points) Repeat part (c) for neural nets with three hidden units. The best test accuracy here should be greater than in part (c). (If not, run your procedures again.) Title the figures appropriately. You can do this question in at most 4 lines of code by reusing code from part (c), i.e., by calling functions defined in part (c). (Each such function call counts as one line of code.)

(e) (4 points) Repeat part (c) for neural nets with four hidden units. The best test accuracy here should be about the same as in part (d). (If not, run your procedures again.) Title the figures appropriately. You can do this in at most 4 lines of code by reusing code from part (c).

(f) (5 points) The best test accuracies in parts (c), (d) and (e) should all be less than in Question 4(c) of Assignment 2. Explain why we should expect this.

ANSWER: The learning method in Question 4(c) of Assignment 2 (Gaussian Discriminant Analysis) assumes the data is Gaussian and exploits this. Since the data is indeed Gaussian, it does very well. Moreover, since there is lots of data, it is the optimal method. In contrast, the neural net makes no such assumptions about the data and must effectively learn that the data is Gaussian. It therefore has a more difficult learning problem.

(g) (5 points) Explain why you get so many different decision boundaries in parts (c), (d) and (e).

ANSWER: The loss function for a neural net has many local minima, which gradient descent may converge to. Moreover, since the initial weights are random, each training run may converge to a different local minimum, thus yielding different weights for the neural net, giving rise to different decision boundaries (and accuracies).

2. (50 points total) Neural Networks: theory

In this question, you will develop matrix equations needed to implement a neural net in Question 3. The net has a single hidden layer and multiple output units, where the hidden units use a sigmoid activation function. We will use the network for multi-class classification, so the activation function at the output is a softmax. The operation of
the neural net can be specified as follows:

\[ o = \text{softmax}(z) \quad z = hW + w_0 \quad h = \sigma(u) \quad u = xV + v_0 \quad (1) \]

Here \( x, h \) and \( o \) are row vectors representing the input, the hidden units, and the output, respectively; \( W \) and \( V \) are weight matrices; \( w_0 \) and \( v_0 \) are row vectors representing bias terms; and \( \sigma \) is the sigmoid function.

Recall that the softmax function is given by

\[ o_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \]

and the sigmoid function is given by \( \sigma(y) = 1/(1+e^{-y}) \), for any real number, \( y \). In this question, you may use the following properties of the softmax and sigmoid functions, respectively, without proving them:

\[
\frac{\partial o_i}{\partial z_j} = o_i \delta_{ij} - o_i o_j \quad (2)
\]

\[
\frac{\partial \sigma(y)}{\partial y} = \sigma(y)[1 - \sigma(y)] \quad (3)
\]

where \( \delta_{ij} \) is the Kronecker delta (i.e., \( \delta_{ij} = 0 \) if \( i \neq j \), and 1 otherwise). You may not use any other results from the lecture slides, although you may use their proofs if you find them useful. (Note however that the derivatives on slide 38 of Lecture 10 are wrong.)

Since we are using the neural network for classification, the loss function used during training is the cross entropy:

\[
C = \sum_n c(t^{(n)}, o^{(n)}) \quad \text{where} \quad c(t, o) = -\sum_i t_i \log o_i \quad (4)
\]

where the left sum is over training points, and the right sum is over output units. Here, \( o^{(n)} \) is the output of the neural net on input \( x^{(n)} \), and \( t^{(n)} \) is a binary vector representing the target class of \( x^{(n)} \) using a 1-of-K encoding (see slide 10 of Lecture 7).

Below, you will derive gradient equations for the neural net. Since the inputs and outputs of each layer of the net are vectors, we will need a generalization of the chain rule from real functions to vector functions. For example, let \( z_j \) be a component of vector \( z \). As explained in class, \( z_j \) affects the loss, \( c \), through each of the output units, \( o_i \). The derivative of \( c \) thus has components due to each of the \( o_i \). In particular,

\[
\frac{\partial c}{\partial z_j} = \sum_i \frac{\partial c}{\partial o_i} \frac{\partial o_i}{\partial z_j} \quad (5)
\]

Likewise, hidden unit \( h_k \) affects \( c \) through each component, \( z_j \), of vector \( z \). Thus,

\[
\frac{\partial c}{\partial h_k} = \sum_i \frac{\partial c}{\partial z_j} \frac{\partial z_j}{\partial h_k} \quad (6)
\]
In the questions below, $X$, $Z$, $H$, $U$, $O$ and $T$ are data matrices whose $n^{th}$ rows are $x^{(n)}$, $z^{(n)}$, $h^{(n)}$, $u^{(n)}$, $o^{(n)}$ and $t^{(n)}$, respectively. $\mathbf{1}$ is a row vector of 1's. To make your proofs easier to mark, use the index $n$ to range over training instances, use $i$ to range over output units, $j$ to range over components of $z$, $k$ to range over hidden units, and $m$ to range over features of an input vector, $x$.

Prove each of the matrix equations below from scratch, using only the results stated above, along with basic results from calculus and linear algebra.

**ANSWER:** First note the following three straightforward results:

$$
\begin{align*}
\left[ \frac{\partial C}{\partial Z} \right]_{nj} &= \frac{\partial c(t^{(n)}, o^{(n)})}{\partial z_j^{(n)}} \quad (7) \\
\left[ \frac{\partial C}{\partial H} \right]_{nk} &= \frac{\partial c(t^{(n)}, o^{(n)})}{\partial h_k^{(n)}} \quad (8) \\
\left[ \frac{\partial C}{\partial U} \right]_{nk} &= \frac{\partial c(t^{(n)}, o^{(n)})}{\partial u_k^{(n)}} \quad (9)
\end{align*}
$$

The first result is proved as follows:

$$
\begin{align*}
\left[ \frac{\partial C}{\partial Z} \right]_{nj} &= \frac{\partial C}{\partial z_{nj}} \quad \text{by definition} \\
&= \frac{\partial C}{\partial z_j^{(n)}} \quad \text{by the definition of } Z \\
&= \frac{\partial \sum_m c(t^{(m)}, o^{(m)})}{\partial z_j^{(n)}} \quad \text{by equation (4)} \\
&= \frac{\partial c(t^{(n)}, o^{(n)})}{\partial z_j^{(n)}} \quad \text{since only } o^{(n)} \text{ depends on } z_j^{(n)}
\end{align*}
$$

The other two results are proved similarly. These results will be used in the proofs below.

(a) (12 points)

$$
\frac{\partial C}{\partial Z} = O - T
$$

**ANSWER:** Note that $\sum_i t_i = 1$, since $t$ is a 1-of-K encoding. With this in mind, we first prove the following result:

$$
\frac{\partial c(t, o)}{\partial z_j} = -\frac{\partial \sum_i t_i \log o_i}{\partial z_j} \quad \text{by equation (4)}
$$
\[ \begin{align*}
&= - \sum_i t_i \frac{\partial \log o_i}{\partial z_j} \quad \text{since } t_i \text{ is a constant} \\
&= - \sum_i t_i \frac{\partial o_i}{o_i} \frac{\partial o_i}{\partial z_j} \quad \text{by the chain rule} \\
&= - \sum_i t_i \frac{\partial}{o_i} [o_i \delta_{ij} - o_j o_i] \quad \text{by equation (2)} \\
&= - \sum_i t_i [\delta_{ij} - o_j] \\
&= - \sum_i t_i \delta_{ij} + \sum_i t_i o_j \\
&= - t_j + o_j \sum_i t_i \\
&= - t_j + o_j \quad \text{as noted above} \\
&= o_j - t_j
\end{align*} \]

Thus,
\[ \left[ \frac{\partial C}{\partial Z} \right]_{nj} = \frac{\partial c(t^{(n)}, o^{(n)})}{\partial z_j^{(n)}} \quad \text{by equation (7)} \]
\[ = o_j^{(n)} - t_j^{(n)} \quad \text{from the result above} \]
\[ = O_{nj} - T_{nj} \quad \text{by the definition of } T \text{ and } O \]
\[ = [O - T]_{nj} \]

Thus \( \frac{\partial C}{\partial Z} = O - T \) since all their components are equal.

(b) (7 points)
\[ \frac{\partial C}{\partial W} = H^T \frac{\partial C}{\partial Z} \]

ANSWER:
First note that \( z_j = \sum_k h_k W_{kj} + w_{0j} \), by Equation (1), and thus \( \frac{\partial z_j}{\partial W_{kj}} = h_k \).
\[ \left[ \frac{\partial C}{\partial W} \right]_{kj} = \frac{\partial C}{\partial W_{kj}} \quad \text{by definition} \]
\[ = \frac{\partial}{\partial W_{kj}} \sum_n c(t^{(n)}, o^{(n)}) \quad \text{by equation (4)} \]
\[ = \sum_n \frac{\partial c(t^{(n)}, o^{(n)})}{\partial W_{kj}} \]
\[ = \sum_n \frac{\partial c(t^{(n)}, o^{(n)})}{\partial z_j^{(n)}} \frac{\partial z_j^{(n)}}{\partial W_{kj}} \quad \text{by the chain rule} \]
\[ = \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} \frac{\partial z_j^{(n)}}{\partial W_{kj}} \quad \text{by equation (7)} \]
\[
\begin{align*}
&= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} h_k^{(n)} \quad \text{as noted above} \\
&= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} H_{nk} \quad \text{by the definition of } H \\
&= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} [H^T]_{kn} \quad \text{by the definition of matrix transpose} \\
&= \sum_n [H^T]_{kn} \left[ \frac{\partial C}{\partial Z} \right]_{nj} \\
&= \left[ H^T \frac{\partial C}{\partial Z} \right]_{kj} \quad \text{by the definition of matrix multiplication}
\end{align*}
\]

Thus $\frac{\partial C}{\partial W} = H^T(\frac{\partial C}{\partial Z})$ since all of their components are equal.

(c) (3 points)

\[
\frac{\partial C}{\partial w_0} = \mathbf{1} \frac{\partial C}{\partial Z}
\]

ANSWER: First note that $z_j = \sum_k h_k w_{kj} + w_{0j}$, by Equation (1), and thus $\frac{\partial z_j}{\partial w_{0j}} = 1$.

\[
\begin{align*}
\left[ \frac{\partial C}{\partial w_{0j}} \right]_j &= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} \frac{\partial z_j^{(n)}}{\partial w_{0j}} \quad \text{as in the proof of part (b)} \\
&= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} 1 \quad \text{as noted above} \\
&= \sum_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} [\mathbf{1}]_n \quad \text{by the definition of } \mathbf{1} \\
&= \sum_n [\mathbf{1}]_n \left[ \frac{\partial C}{\partial Z} \right]_{nj} \\
&= \left[ \mathbf{1} \frac{\partial C}{\partial Z} \right]_j \quad \text{by the definition of matrix-vector multiplication}
\end{align*}
\]

Thus $\frac{\partial C}{\partial w_0} = [\mathbf{1}] (\frac{\partial C}{\partial Z})$, since all their components are equal.

(d) (9 points)

\[
\frac{\partial C}{\partial H} = \frac{\partial C}{\partial Z} W^T
\]
ANSWER: First note that \( z_j = \sum_k h_k W_{kj} + w_{0j} \), by Equation (1), and thus \( \frac{\partial z_j}{\partial h_k} = W_{kj} \).

\[
\left[ \frac{\partial C^*}{\partial H} \right]_{nk} = \frac{\partial c(\ell^{(n)}, o^{(n)})}{\partial h_k^{(n)}} \quad \text{by equation (8)}
\]

\[
= \sum_j \frac{\partial c(\ell^{(n)}, o^{(n)})}{\partial z_j^{(n)}} \frac{\partial z_j^{(n)}}{\partial h_k^{(n)}} \quad \text{by equation (6)}
\]

\[
= \sum_j \left[ \frac{\partial C}{\partial Z} \right]_{nj} \frac{\partial z_j^{(n)}}{\partial h_k^{(n)}} \quad \text{by equation (7)}
\]

\[
= \sum_j \left[ \frac{\partial C}{\partial Z} \right]_{nj} W_{kj} \quad \text{by the note above}
\]

\[
= \sum_j \left[ \frac{\partial C}{\partial Z} \right]_{nj} [W^T]_{jk} \quad \text{by the definition of matrix transpose}
\]

\[
= \left[ \frac{\partial C^*}{\partial Z} W^T \right]_{nk} \quad \text{by the definition of matrix multiplication}
\]

Thus \( \frac{\partial C}{\partial H} = (\frac{\partial C}{\partial Z})W^T \) since all their components are equal.

(e) (7 points)

\[
\left[ \frac{\partial C}{\partial U} \right]_{nk} = H_{nk}(1 - H_{nk}) \left[ \frac{\partial C}{\partial H} \right]_{nk}
\]

ANSWER: Note that \( h_k = \sigma(u_k) \), by Equation (1).

\[
\left[ \frac{\partial C}{\partial U} \right]_{nk} = \frac{\partial C}{\partial U_{nk}} \quad \text{by definition}
\]

\[
= \frac{\partial C}{\partial u_k^{(n)}} \quad \text{by the definition of } U
\]

\[
= \frac{\partial C}{\partial h_k^{(n)}} \frac{\partial h_k^{(n)}}{\partial u_k^{(n)}} \quad \text{by the chain rule}
\]

\[
= \frac{\partial C}{\partial h_k^{(n)}} \sigma(u_k^{(n)}) \quad \text{as noted above}
\]

\[
= \frac{\partial C}{\partial h_k^{(n)}} \sigma(u_k^{(n)})[1 - \sigma(u_k^{(n)})] \quad \text{by equation (3)}
\]

\[
= \frac{\partial C}{\partial h_k^{(n)}} h_k^{(n)}[1 - h_k^{(n)}] \quad \text{as noted above}
\]

\[
= \frac{\partial C}{\partial H_{nk}} H_{nk}(1 - H_{nk}) \quad \text{by the definition of } H
\]
\[ \frac{\partial C}{\partial V} = X^T \frac{\partial C}{\partial U} \]

**ANSWER:** The proof is similar to part (b). First note that \( u_k = \sum_m x_m V_{mk} + v_{0k} \), by Equation (1), and thus \( \partial u_k / \partial V_{mk} = x_m \).

\[
\begin{bmatrix} \frac{\partial C}{\partial V} \end{bmatrix}_{mn} = \frac{\partial C}{\partial V_{mk}} \quad \text{by definition}
= \frac{\partial}{\partial V_{mk}} \sum_n c(t^{(n)}, o^{(n)}) \quad \text{by equation (4)}
= \sum_n \frac{\partial c(t^{(n)}, o^{(n)})}{\partial V_{mk}} \quad \text{by the chain rule}
= \sum_n \frac{\partial c(t^{(n)}, o^{(n)})}{\partial u_k^{(n)}} \frac{\partial u_k^{(n)}}{\partial V_{mk}} \quad \text{by equation (9)}
= \sum_n \frac{\partial C}{\partial U} \begin{bmatrix} \frac{\partial C}{\partial V_{mn}} \end{bmatrix}_{nk} \quad x_m^{(n)} \quad \text{as noted above}
= \sum_n \frac{\partial C}{\partial U} \begin{bmatrix} \frac{\partial C}{\partial U} \end{bmatrix}_{nk} X_{mn} \quad \text{by the definition of X}
= \sum_n \frac{\partial C}{\partial U} \begin{bmatrix} \partial C \end{bmatrix}_{nk} \begin{bmatrix} X^T \end{bmatrix}_{mn} \quad \text{by the definition of matrix transpose}
= \sum_n \begin{bmatrix} X^T \end{bmatrix}_{mn} \frac{\partial C}{\partial V} \begin{bmatrix} \partial C \end{bmatrix}_{nk}
= \begin{bmatrix} X^T \cdot \frac{\partial C}{\partial U} \end{bmatrix}_{mnk} \quad \text{by the definition of matrix multiplication}
\]

Thus \( \partial C / \partial V = X^T (\partial C / \partial U) \) since all of their components are equal.

\( (g) \) (3 points)

\[ \frac{\partial C}{\partial v_0} = I \frac{\partial C}{\partial U} \]
ANSWER: The proof is similar to part (c). First note that $u_k = \sum_m x_m V_{mk} + v_{0k}$, by Equation (1), and thus $\partial u_k / \partial v_{0k} = 1$.

\[
\left[ \frac{\partial C}{\partial v_0} \right]_k = \sum_n \left[ \frac{\partial C}{\partial U} \right]_{nk}^{(n)} \frac{\partial u_k^{(n)}}{\partial v_{0k}} \text{ as in the proof of part (f)} \\
= \sum_n \left[ \frac{\partial C}{\partial U} \right]_{nk}^{(n)} 1 \text{ as noted above} \\
= \sum_n \left[ \frac{\partial C}{\partial U} \right]_{nk}^{(n)} \left[ \mathbf{1} \right]_n \text{ by the definition of } \mathbf{1} \\
= \sum_n \left[ \mathbf{1} \right]_n \left[ \frac{\partial C}{\partial U} \right]_{nk}^{(n)} \text{ by the definition of matrix-vector multiplication} \\
= \left[ \mathbf{1} \frac{\partial C}{\partial U} \right]_k \text{ by the definition of matrix-vector multiplication}
\]

Thus $\partial C / \partial v_0 = \mathbf{1} \left( \partial C / \partial U \right)$, since all their components are equal.

(h) (2 points) Let $\mathbf{dCdw}_0$ and $\mathbf{dCdZ}$ be Numpy arrays representing $\partial C / \partial w_0$ and $\partial C / \partial Z$, respectively. Write a single line of Python code for computing $\partial C / \partial w_0$ using the equation in part (c) without constructing a vector of 1's or using any multiplication. Do not execute this code. Hand it in in the same file as your proofs.

ANSWER: $\mathbf{dCdw}_0 = \text{np.sum}(\mathbf{dCdZ}, \text{axis}=0)$

3. (60 points total) Neural Networks: implementation

In this question, you will use the theory developed in Question 2 to write a Python program that trains neural networks on the MNIST data from Assignment 2. As in Question (2), we will only consider neural nets with one hidden layer and a sigmoid (logistic) activation function. You will implement both batch and stochastic gradient descent. The batch implementation is more straightforward, but as you will see, it takes much longer to converge. For comparison, and to provide some quick results, you will first train neural networks on MNIST using MLPclassifier. Do not use any functions in sklearn other than MLPclassifier and its methods.

In parts (a) to (e), you will use a reduced training set, which will reduce accuracy, but will speed up training and program development. Use the first 10,000 points of the MNIST training data as validation data, and use the next 10,000 points as the reduced training data. (So 40,000 of the 60,000 MNIST training points will be unused.) Use all of the MNIST test data for testing. Once your programs are working, you will use the entire MNIST data set in parts (f) and (g). When you are asked to compute the cross entropy, this refers to the training cross entropy, i.e., cross entropy computed wrt the data the neural net was trained on (whether reduced data or not).

(a) (5 points) Stochastic Gradient Descent.
Using **MLPClassifier** in **sklearn**, train a neural net with 30 hidden units on the reduced MNIST data. Do 10 iterations of training. Use stochastic gradient descent as the solver with a batch size of 100. Set the tolerance for optimization to $10^{-8}$, a very low value to ensure all 10 iterations are carried out. Do not use 1-of-K encodings of class labels with **MLPClassifier**, as it expects integer labels for multi-class classification.

You will have to experiment to find a good learning rate. At this stage, you may want to set the keyword argument **verbose** to True, to print out the training error at each iteration. If the training error increases as learning progresses, or bounces around erratically, then the learning rate is too high. If the training error decreases very slowly, then the learning rate is too small. Find a learning rate that maximizes the test accuracy after 10 iterations. Only consider learning rates that are powers of 10, like 100, 10, 1, 0.1, 0.01, etc. You should be able to achieve a test accuracy of about 94% after 10 iterations. Do not hand in any of these experiments. The point is simply to find a good learning rate.

Using the learning rate you have just found, train the neural net 10 times (with **verbose=False**) on the reduced MNIST data set. Compute and print out the validation accuracy of each trained net. Choose the trained net that has the maximum validation accuracy. Print out its validation accuracy, test accuracy and cross entropy. The cross entropy should be between 300 and 500. (If not, try running your program again.) You can use the **score** method to compute the accuracies, but you will have to compute the cross entropy yourself. Finally, print out the learning rate that you used. You may use one loop in your program.

(b) (10 points total) **Batch Gradient Descent.**

Repeat part (a) with a batch size equal to the size of the training set. This effectively carries out batch gradient descent. You should find that the same learning rate is still about the best, but the accuracy after 10 iterations is lower than in part (a) and is 75%-80%. (5 points) **Explain why this is** (5 points).

**ANSWER:** Stochastic gradient descent (sgd) performs more weight updates per epoch than batch gradient descent (bgd). In this case, sgd performs $10,000/100 = 100$ updates per epoch, whereas bgd performs only 1. While the gradient computed by sgd may not be in the direction of steepest descent, like bgd, it will still be in a good downhill direction, since 100 samples provides a lot of information about what MNIST digits look like. In effect, 100 good updates (sgd) gets one much closer to the minimum than 1 very good update (bgd).

(c) (5 points) Here, we will see how long it takes batch gradient descent to achieve the same level of test accuracy as stochastic gradient descent. Use **MLPClassifier** with the same arguments as in part (b) to define a neural net and train it for 50 iterations. Train the net just once, not 10 times. Print the final training and

---

**MLPClassifier** assumes that class labels encoded as binary arrays are meant for multi-label classification (something we have not studied), which is different from multi-class classification.
test accuracies and cross entropy. You should find that the test accuracy is about 92%. Do not use any loops.

Repeat again using 200 iterations. The final test accuracy should now be closer to 94%, i.e., an accuracy that was achieved in part (a) after just 10 iterations.

(d) (20 points total) Batch Gradient Descent: implementation.

Use the theory developed in Question 2 to write a Python program that trains a neural net with one hidden layer for multi-class classification using batch gradient descent. Your program may use one loop. Using program comments, clearly indicate what portion of your program implements the forward pass of training, and what portion implements back propagation. Initialize the weight matrices randomly using a standard Gaussian distribution (i.e., mean 0 and variance 1), and initialize the bias terms to 0.

When doing weight updates, use the average gradient, not the gradient itself. For instance, to update the weight matrix $W$, use the command

$$W = W - \lambda \frac{\partial C}{\partial W} / N$$

where $\lambda$ is the learning rate and $N$ is the number of terms in the sum in equation (4). In batch gradient descent, $N$ is the number of points in the training set, but would be the batch size in stochastic gradient descent. Using the average gradient means that the optimal learning rate does not change much when the size of the training set changes (which is why the same learning rate worked in parts (a) and (b)). Explain why this is (5 points). Notice that using the average gradient is equivalent to dividing the learning rate by $N$.

**ANSWER:** According to Equation (4), weight gradients are given by equations like the following:

$$\frac{\partial C}{\partial W} = \sum_{n=1}^{N} \frac{\partial c(t^{(n)}, o^{(n)})}{\partial W}$$

As $N$, the number of points in the sum, increases, we do not expect the direction of the gradient to change much, as explained in the answer to part (b). Thus, the main effect of the sum is to increase the magnitude of the gradient, so that it is proportional to $N$. Normally, one would have to counteract this effect by decreasing the learning rate as $N$ increases. Using the average gradient, instead of the sum achieves this automatically, so that the learning rate can remain roughly constant as $N$ increases.

Run your program on the reduced MNIST data set using 30 hidden units and 100 iterations of gradient descent. Again, you will have to experiment to find a good learning rate. You should be able to achieve a test accuracy of around 85% after 100 iterations. Do not hand in any of these experiments. You will probably find that the optimal learning rate here is different than in part (b), since MLPclassifier uses momentum, while your program does not.
As in part (a), use the learning rate you have just found to train a neural net 10 times. (You may use one additional loop for this purpose.) Compute and print out the validation accuracy of each trained net. Choose the trained net that has the maximum validation accuracy. Print out its validation accuracy, test accuracy and cross entropy. Finally, print out the learning rate that you used. (15 points)

(e) (10 points) **Stochastic Gradient Descent: implementation.**

Modify your program in part (d) to perform stochastic gradient descent with mini-batches. That is, instead of computing the gradient of the loss function on the entire training set at once, compute the gradient on a small, random subset of the training data (called a mini-batch), perform weight updates, and then move on to the next mini-batch, and so on. As you saw in parts (a), (b) and (c), this can lead to much faster convergence. As in part (d), your implementation of the weight-update step should use the average gradient (averaged over the current mini-batch).

To produce random mini-batches of the training data, first shuffle the training data randomly, then sweep across the shuffled data from start to finish. For example, if we want mini-batches of size 100, then the first mini-batch is the first 100 points in the training set. The second mini-batch is the next 100 points. The third mini-batch is the next 100 points, etc. (If the number of training points is not a multiple of 100 then the last mini-batch in a sweep will have fewer than 100 points in it.) Each such sweep of the training data is called an epoch. Before each epoch begins, you should reshuffle the entire training set randomly, so that each epoch produces a different, random set of mini-batches.

Your program may use two loops, one nested inside the other. Program comments should clearly indicate where an epoch begins and where mini-batches are created. Now, repeat the rest of part (d), performing 100 epochs of training with mini-batches of size 100, as in part (a). Because you are using the average gradient to perform weight updates, you should find that the optimal learning rate here is about the same as in part (d). You should be able to achieve a test accuracy of over 90% after 100 epochs.

(f) (5 points) In the rest of this question, you will use the entire MNIST data set, so you may want to reload it from a file. With more data, we can achieve higher test accuracy. To fully take advantage of this possibility, we will increase the number of hidden units in the neural net, to increase the net’s capacity to represent complex functions and decision boundaries. With more data and more hidden units, the neural net will now take much longer to train, so we will forgo the use of validation data, and will train the net just once, not 10 times.

Modify your program for stochastic gradient descent so that it prints out the test accuracy after every ten epochs (with the first print-out being after the first epoch), so we can monitor the progress of training.

Use the modified program to train a neural net with 100 hidden units on the full MNIST data set using mini-batches of size 100. Perform 100 epochs of training. Print out the final training accuracy, test accuracy and cross entropy. You should
be able to achieve a final test accuracy of over 96%. In fact, a test accuracy of over 90% can be achieved after just one epoch, and an accuracy of over 96% after 50 epochs. You should also find that the final training accuracy is almost 100%.

(g) (5 points) Repeat part (f) with your program for batch gradient descent. You should find that the test accuracy increases much more slowly during training, and that the final test accuracy is below 90%.

140 points total
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
import bonnerlib2
import sklearn.utils as utils
import pickle

##### Question 1 #####

# from assignment 2.

# generate cluster data
def gen_data(mu0,mu1,cov0,cov1,N0,N1):
    Sigma0 = [[1, cov0], [cov0, 1]]
    Sigma1 = [[1, cov1], [cov1, 1]]
    data0 = rnd.multivariate_normal(mu0,Sigma0,N0)  # data matrix for class 0
    data1 = rnd.multivariate_normal(mu1,Sigma1,N1)  # data matrix for class 1
    # combine the two data matrices into one
    X = np.concatenate([data0, data1])
    # construct a vector of labels
    t0 = np.zeros([N0], dtype=np.int32)  # labels for cluster 0
    t1 = np.ones([N1], dtype=np.int32)  # labels for cluster 1
    # combine the two vectors into one
    t = np.concatenate([t0, t1])
    return utils.shuffle(X, t)  # shuffle and return the data

# Question 1(a)

# generate training data
mu0 = [1.0, 1.0]
mu1 = [2.0, 2.0]
cov0 = 0
cov1 = 0.9
N0 = 1000
N1 = 500
Xtrain, Ttrain = gen_data(mu0, mu1, cov0, cov1, N0, N1)

# generate test data
N0 = 10000
N1 = 5000
Xtest, Ttest = gen_data(mu0, mu1, cov0, cov1, N0, N1)

# Question 1(b)
print('
')
print('Question 1(b).')
print('-------------')

# display cluster data.
xmin = -3
xmax = 6
```python
def plot_data(X,T):
    colors = np.array(['r', 'b'])  # red for class 0, blue for class 1
    plt.scatter(X[:, 0], X[:, 1], color=colors[T], s=2)
    plt.xlim(xmin, xmax)
    plt.ylim(xmin, xmax)

# compute the precision and recall of classifier clf on data set X,T.
def precision_recall(clf, X, T):
    # classify each test point as positive, predicted positive, etc.
    Pos = T  # positives
    PP = clf.predict(X)  # predicted positives
    TP = Pos & PP  # true positive predictions
    numPP = np.sum(P)
    numTP = np.sum(TP)
    numTP = np.float(numPP)
    Recall = numTP / np.float(numP)
    return Precision, Recall

# Fit a neural net with 1 hidden unit to the training data
nn1 = MLPClassifier(hidden_layer_sizes=[1],
                        activation='logistic',
                        max_iter=1000,
                        solver='sgd',
                        tol=10.0**(8),
                        learning_rate_init=0.01)

nn1.fit(Xtrain, Ttrain)  # train the neural net

plt.figure()
plot_data(Xtrain, Ttrain)
plt.suptitle('Question 1(b): Neural net with 1 hidden unit')

accuracy = nn1.score(Xtest, Ttest)
precision, recall = precision_recall(nn1, Xtest, Ttest)
print('Accuracy = ', accuracy)
print('Precision = ', precision)
print('Recall = ', recall)

# Question 1(c)
print('
')
print('Question 1(c).')
print('----------

# fit 12 neural nets with K hidden units to the training data
for i in range(12):
    clf = MLPClassifier(hidden_layer_sizes=[K],
                        activation='logistic',
                        max_iter=1000,
                        solver='sgd',
                        tol=10.0**(8),
                        learning_rate_init=0.01)
    clf.fit(Xtrain, Ttrain)  # train a neural net
    accuracy = clf.score(Xtest, Ttest)
```

2
```python
# plot the decision boundary
plt.subplot(4,3,i+1)
plot data(Xtrain,Ttrain)
plt.axis('off')
bonnerlib2.dfContour(clf)
# keep track of the best neural net so far
if accuracy > accBest:
    accBest = accuracy  # the best accuracy so far
    clfBest = clf  # the best classifier so far
precision,recall = precision recall(clfBest,Xtest,Ttest)
print 'Accuracy of best classifier = ', accBest
print 'Precision of best classifier = ', precision
print 'Recall of best classifier = ', recall

# plot the decision boundary of the best classifier
fig2 = plt.figure()
plot data(Xtrain,Ttrain)
bonnerlib2.dfContour(clfBest)
return clfBest,fig1,fig2
K = 2  # 2 hidden units
nn2,fig1,fig2 = fitNN(K)
fig1.suptitle('Question 1(c): Neural nets with 2 hidden units')
fig2.suptitle('Question 1(c): Best neural net with 2 hidden units')

# Question 1(d)
print('n')
print('Question 1(d).')
print('---------')
K = 3  # 3 hidden units
nn3,fig1,fig2 = fitNN(K)
fig1.suptitle('Question 1(d): Neural nets with 3 hidden units')
fig2.suptitle('Question 1(d): Best neural net with 3 hidden units')

# Question 1(e)
print('n')
print('Question 1(e).')
print('---------')
K = 4  # 4 hidden units
nn4,fig1,fig2 = fitNN(K)
fig1.suptitle('Question 1(e): Neural nets with 4 hidden units')
fig2.suptitle('Question 1(e): Best neural net with 4 hidden units')

########## Question 3 ##########

# read MNIST data from a file
with open('mnist.pickle','r') as f:
    mnist = pickle.load(f)
Xtrain,Ttrain,Xtest,Ttest = mnist
```
N = 10000
Xval = Xtrain[N:2*N]
Tval = Ttrain[N:2*N]
Xtrain = Xtrain[:N]
Ttrain = Ttrain[:N]
Ntrain = N

# Question 3(a)
print('
')
print('Question 3(a).')
print('--------------')

# convert T from an integer encoding of class labels to a one_hot encoding
def one_hot(T):
    N = np.shape(T)[0]
    J = np.max(T) + 1
    hotT = range(J)
    hotT = np.reshape(hotT,[1,J])
    hotT = np.tile(hotT,[N,1])
    T = np.reshape(T,[N,1])
    hotT = (hotT==T)
    return hotT.astype(int)

# loss function for the neural net.
# class labels, T, have an integer encoding.
def cross_entropy(0,T):
    return -np.sum(one_hot(T)*np.log(0))  # mean cross entropy

# test a learning rate
# K = 30
# lrate = 1
# nn = MLPClassifier(hidden_layer_sizes=[K],
#                     activation='logistic',
#                     max_iter=10,
#                     solver='sgd',
#                     tol=10.0**(-8),
#                     learning_rate_init=lrate,
#                     batch_size=100,
#                     verbose=True)
# nn.fit(Xtrain,Ttrain)
# print 'Validation accuracy =', nn.score(Xval,Tval)
# print 'Training accuracy =', nn.score(Xtrain,Ttrain)
# print 'Test accuracy =', nn.score(Xtest,Ttest)
# prob = nn.predict_proba(Xtrain)
# print 'Cross entropy =', cross_entropy(prob,Ttrain)

K = 30    # number of hidden units
lrate = 1  # initial learning rate
nn = MLPClassifier(hidden_layer_sizes=[K],
                    activation='logistic',
                    max_iter=10,
                    solver='sgd',

4
\[ \text{tol}=10.0 \times 10^{-8}, \]
\[ \text{learning rate init}=\text{lrate}, \]
\[ \text{batch size}=100 \]

\[ I = 10 \quad \# \text{number of training sessions} \]
\[ \text{accBest} = 0 \quad \# \text{best accuracy so far} \]

\text{for} \ i \ \text{in} \ \text{range}(I): \]
\text{nn.fit(Xtrain,Ttrain)}
\text{accVal} = \text{nn.score(Xval,Tval)}
\text{print 'Validation accuracy =', accVal}
\text{if accVal} > \text{accBest}:
\text{accBest} = \text{accVal}
\text{nnBest} = \text{nn}

\text{print '}'
\text{print 'Maximum validation accuracy =', accBest}
\text{accTest} = \text{nn.score(Xtest,Ttest)}
\text{print 'Test accuracy =', accTest}
\text{prob} = \text{nn.predict_proba(Xtrain)}
\text{print 'Cross entropy =', cross_entropy(prob,Ttrain)}
\text{print 'Learning rate =', lrate}

\# Question 3(b)
\text{print('
')}
\text{print('Question 3(b).')}
\text{print('---------')}

\[ K = 30 \quad \# \text{number of hidden units} \]
\[ \text{lrate} = 1 \quad \# \text{initial learning rate} \]
\text{nn = MLPClassifier(hidden_layer_sizes=[K],}
\text{activation='logistic',}
\text{max iter=10,}
\text{solver='sqd',}
\text{tol=10.0 \times 10^{-8},}
\text{learning rate init}=\text{lrate},
\text{batch size=Ntrain)}

\[ I = 10 \quad \# \text{number of training sessions} \]
\[ \text{accBest} = 0 \quad \# \text{best accuracy so far} \]

\text{for} \ i \ \text{in} \ \text{range}(I): \]
\text{nn.fit(Xtrain,Ttrain)}
\text{accVal} = \text{nn.score(Xval,Tval)}
\text{print 'Validation accuracy =', accVal}
\text{if accVal} > \text{accBest}:
\text{accBest} = \text{accVal}
\text{nnBest} = \text{nn}

\text{print '}
\text{print 'Maximum validation accuracy =', accBest}
\text{accTest} = \text{nn.score(Xtest,Ttest)}
\text{print 'Test accuracy =', accTest}
\text{prob} = \text{nn.predict_proba(Xtrain)}
\text{print 'Cross entropy =', cross_entropy(prob,Ttrain)}
\text{print 'Learning rate =', lrate}
# Question 3(c)
print('
')
print('Question 3(c).')
print('----------

K = 30       # number of hidden units
lrate = 1    # initial learning rate

# train for 50 iterations
nn = MLPClassifier(hidden_layer_sizes=[K],
                    activation='logistic',
                    max_iter=50,
                    solver='sqd',
                    tol=10.0**(-8),
                    learning_rate_init=lrate,
                    batch_size=Ntrain)
nn.fit(Xtrain,Ttrain)
print 'Results for 50 iterations:'
print 'Validation accuracy =', nn.score(Xval,Tval)
print 'Test accuracy =', nn.score(Xtest,Ttest)
prob = nn.predict_proba(Xtrain)
print 'Cross entropy =', cross_entropy(prob,Ttrain)
print 'Learning rate =', lrate

# train for 200 iterations
nn = MLPClassifier(hidden_layer_sizes=[K],
                    activation='logistic',
                    max_iter=200,
                    solver='sqd',
                    tol=10.0**(-8),
                    learning_rate_init=lrate,
                    batch_size=Ntrain)
nn.fit(Xtrain,Ttrain)
print '
print 'Results for 200 iterations:'
print 'Validation accuracy =', nn.score(Xval,Tval)
print 'Test accuracy =', nn.score(Xtest,Ttest)
prob = nn.predict_proba(Xtrain)
print 'Cross entropy =', cross_entropy(prob,Ttrain)
print 'Learning rate =', lrate

# Question 3(d)
print('
')
print('Question 3(d).')
print('----------

# convert all class labels to a one-hot representation
Ttrain = one_hot(Ttrain)
Tval = one_hot(Tval)
Ttest = one_hot(Ttest)
# loss function for the neural net.
# class labels, T, have one-hot encoding.

def cross_entropy(O, T):
    return -np.sum(T*np.log(O))  # mean cross entropy

# accuracy of the neural net

def accuracy(O, T):
    Predictions = np.argmax(O, axis=1)  # predictions
    Truth = np.argmax(T, axis=1)
    correct = (Predictions == Truth)  # correct predictions
    return np.mean(correct)

def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def softmax(Z):
    expZ = np.exp(Z)
    sumExpZ = np.sum(expZ, axis=1, keepdims=True)
    return expZ/sumExpZ

# do a forward pass through a neural net.
# return the output and hidden values

def forward(X, V, v0, W, w0):
    U = np.matmul(X, V) + v0
    H = sigmoid(U)
    Z = np.matmul(H, W) + w0
    O = softmax(Z)
    return O, H

# do N epochs of batch gradient descent with K hidden units
# where lrate is the learning rate.
# return the weights and bias terms

def bgd(K, N, lrate):
    Ntrain = np.shape(Xtrain)[0]
    lrate = lrate/float(Ntrain)
    # initialise the weights and biases
    M = np.shape(Xtrain)[1]  # number of input units
    J = np.shape(Ttrain)[1]  # number of output units
    V = rnd.randn(M, K)  # random initial weights for hidden layer
    W = rnd.randn(K, J)  # random initial weights for output layer
    v0 = np.zeros([1, K])  # initial bias terms for hidden layer
    w0 = np.zeros([1, J])  # initial bias term for output layer

    for n in range(N):
        # forward pass
        O, H = forward(Xtrain, V, v0, W, w0)

        # backward pass (compute gradients of C)
        Delta = O - Ttrain  # gradient wrt Z
        gW = np.matmul(H, T, Delta)  # gradient wrt W
        gW0 = np.sum(Delta, axis=0)  # gradient wrt w0
        Delta = np.matmul(Delta, W)  # gradient wrt H
        Delta = H*(1-H)*Delta  # gradient wrt U
        gV = np.matmul(Xtrain.T, Delta)  # gradient wrt V
gv0 = np.sum(Delta, axis=0)  # gradient w.r.t v0

# update weights and biases
V -= lrate*gV
v0 -= lrate*gv0
W -= lrate*gW
w0 -= lrate*gw0

# every 10 epochs, compute and print the test accuracy
if np.mod(n, 10) == 0:
    # compute and print test accuracy
    Otest = forward(Xtest, V, v0, W, w0)[0]
    accTest = accuracy(Otest, Ttest)
    print 'Epoch', n+1, '  Test accuracy=', accTest

return V, v0, W, w0

## test a learning rate
## execute batch gradient descent
K = 30  # number of hidden units
N = 100  # number of epochs of training
lrate = 10  # learning rate
V, v0, W, w0 = bgd(K, N, lrate)

# O = forward(Xval, V, v0, W, w0)[0]
# print 'Validation accuracy=', accuracy(O, Tval)
# O = forward(Xtest, V, v0, W, w0)[0]
# print 'Test accuracy=', accuracy(O, Ttest)
# print 'Cross entropy=', cross_entropy(O, Ttrain)

K = 30  # number of hidden units
N = 100  # number of epochs of training
lrate = 10  # learning rate
I = 10  # number of training sessions
accBest = 0
for i in range(I):
    V, v0, W, w0 = bgd(K, N, lrate)
    O = forward(Xval, V, v0, W, w0)[0]
    accVal = accuracy(O, Tval)
    print 'Validation accuracy=', accVal
    if accVal > accBest:
        accBest = accVal
        weightsBest = [V, v0, W, w0]

print ''
print 'Maximum validation accuracy=', accBest
V, v0, W, w0 = weightsBest
Otest = forward(Xtest, V, v0, W, w0)[0]
Otrain = forward(Xtrain, V, v0, W, w0)[0]
print 'Test accuracy=', accuracy(Otest, Ttest)
print 'Cross entropy=', cross_entropy(Otrain, Ttrain)
print 'Learning rate=', lrate
# Question 3(e)
print('n')
print('Question 3(e).')
print('----------'

# do N epochs of stochastic gradient descent with the given mini-batch size
# and K hidden units, where lrate is the learning rate.
# return the validation accuracy and a list of the neural-net weights.
def sgd(K,N,lrate,batch_size):
    # initialise the weights and biases
    M = np.shape(Xtrain)[1] # number of input units
    J = np.shape(Ttrain)[1] # number of output units
    V = rnd.randn(M,K) # random initial weights for hidden layer
    W = rnd.randn(K,J) # random initial weights for output layer
    v0 = np.zeros([1,K]) # initial bias terms for hidden layer
    w0 = np.zeros([1,J]) # initial bias term for output layer

    Ntrain = np.shape(Xtrain)[0] # number of training points
    idx = range(Ntrain)
    for n in range(N):
        # shuffle the training data randomly
        rnd.shuffle(idx)
        Xtrain = Xtrain[idx]
        Ttrain = Ttrain[idx]
        # perform one epoch of stochastic gradient descent
        ptr1 = 0 # pointer to start of mini-batch
        while ptr1 < Ntrain:
            # get next mini-batch of data
            ptr2 = np.min([ptr1+batch_size,Ntrain]) # pointer to end of mini-batch
            X = Xtrain[ptr1:ptr2]
            T = Ttrain[ptr1:ptr2]
            ptr1 = ptr2 # update mini-batch pointer

            # forward pass
            O,H = forward(X,V,v0,W,w0)

            # backward pass (compute gradients of C)
            Delta = O - T # gradient wrt Z
            gW = np.matmul(H.T,Delta) # gradient wrt W
            gw0 = np.sum(Delta,axis=0) # gradient wrt w0
            Delta = np.matmul(Delta,W.T) # gradient wrt H
            gV = np.matmul(X.T,Delta) # gradient wrt V
            gv0 = np.sum(Delta,axis=0) # gradient wrt v0

            # update weights and biases
            V -= lrate*gV
            v0 -= lrate*gv0
            W -= lrate*gW
            w0 -= lrate*gw0

        # every 10 epochs, compute and and print the test accuracy
        if np.mod(n,10) == 0:
            # compute and print test accuracy
            Otest = forward(Xtest,V,v0,W,w0)[0]
            accTest = accuracy(Otest,Ttest)
            print 'Epoch', n+1, ' Test accuracy =', accTest
return V,v0,W,w0

K = 30    # number of hidden units
N = 100   # number of epochs of training
lrate = 10    # learning rate
batch size = 100
I = 10    # number of training sessions
accBest = 0
for i in range(I):
    V,v0,W,w0 = sgd(K,N,lrate,batch_size)
    O = forward(Xval,V,v0,W,w0)[0]
    accVal = accuracy(O,Tval)
    print 'Validation accuracy =', accVal
    if accVal > accBest:
        accBest = accVal
        weightsBest = [V,v0,W,w0]

print ''
print 'Maximum validation accuracy =', accBest
V,v0,W,w0 = weightsBest
Otest = forward(Xtest,V,v0,W,w0)[0]
Otrain = forward(Xtrain,V,v0,W,w0)[0]
print 'Test accuracy =', accuracy(Otest,Ttest)
print 'Cross entropy =', cross_entropy(Otrain,Ttrain)
print 'Learning rate =', lrate

# Question 3(f)
print('\n')
print('Question 3(f).')
print('---------

# reload MNIST data from a file
with open('mnist.pickle','r') as f:
    mnist = pickle.load(f)
Xtrain,Ttrain,Xtest,Ttest = mnist

# convert the target values from an integer encoding of class labels to a one_hot
Ttrain = one_hot(Ttrain)
Ttest = one_hot(Ttest)

# execute stochastic gradient descent
K = 100    # number of hidden units
N = 100    # number of epochs of training
lrate = 10    # learning rate
batch size = 100
V,v0,W,w0 = sgd(K,N,lrate,batch_size)

print ''
Otest = forward(Xtest,V,v0,W,w0)[0]
Otrain = forward(Xtrain,V,v0,W,w0)[0]
print 'Training accuracy =', accuracy(Otrain,Ttrain)
print 'Test accuracy =', accuracy(Otest,Ttest)
print 'Cross entropy =', cross_entropy(Otrain,Ttrain)

# Question 3(g)
print('
')
print('Question 3(g).')
print('-------------

# execute batch gradient descent
K = 100    # number of hidden units
N = 100    # number of epochs of training
lrate = 10  # learning rate
V,v0,W,w0 = bgd(K,N,lrate)

print ''
Otest = forward(Xtest,V,v0,W,w0)[0]
Otrain = forward(Xtrain,V,v0,W,w0)[0]
print 'Training accuracy =', accuracy(Otrain,Ttrain)
print 'Test accuracy =', accuracy(Otest,Ttest)
print 'Cross entropy =', cross_entropy(Otrain,Ttrain)
Question 1(b).

---------
Accuracy = 0.78326666666667
Precision = 0.7642792384406165
Recall = 0.5058

Question 1(c).

---------
Accuracy of best classifier = 0.825133333333334
Precision of best classifier = 0.7594411700502074
Recall of best classifier = 0.6958

Question 1(d).

---------
Accuracy of best classifier = 0.8278666666666666
Precision of best classifier = 0.7888198757763976
Recall of best classifier = 0.6604

Question 1(e).

---------
Accuracy of best classifier = 0.827733333333333
Precision of best classifier = 0.7903846153846154
Recall of best classifier = 0.6576

Question 3(a).

---------
Validation Accuracy = 0.9309
Validation Accuracy = 0.9366
Validation Accuracy = 0.9407
Validation Accuracy = 0.932
Validation Accuracy = 0.9382
Validation Accuracy = 0.9397
Validation Accuracy = 0.9371
Validation Accuracy = 0.9374
Validation Accuracy = 0.9356
Validation Accuracy = 0.935

Maximum validation accuracy = 0.9407
Test accuracy = 0.9359
Cross entropy = 360.41561295220527
Learning rate = 1

Question 3(b).

Validation Accuracy = 0.7698
Validation Accuracy = 0.7909
Validation Accuracy = 0.7923
Validation Accuracy = 0.7924
Validation Accuracy = 0.7903
Validation Accuracy = 0.7854
Validation Accuracy = 0.7892
Validation Accuracy = 0.7946
Validation Accuracy = 0.7962
Validation Accuracy = 0.7894

Maximum validation accuracy = 0.7962
Test accuracy = 0.8028
Cross entropy = 8356.912220228845
Learning rate = 1

Question 3(c).

Results for 50 iterations:
Validation accuracy = 0.9172
Test accuracy = 0.921
Cross entropy = 2293.476702768422
Learning rate = 1

Results for 200 iterations:
Validation accuracy = 0.9371
Test accuracy = 0.9382
Cross entropy = 814.5757217094275
Learning rate = 1

Question 3(d).

Validation accuracy = 0.8398
Validation accuracy = 0.8596
Validation accuracy = 0.8449
Validation accuracy = 0.8551
Validation accuracy = 0.8598
Validation accuracy = 0.8303
Validation accuracy = 0.8481
Validation accuracy = 0.8691
Validation accuracy = 0.8593
Validation accuracy = 0.8517

Maximum validation accuracy = 0.8691
Test accuracy = 0.8671
Cross entropy = 3608.273001991528
Learning rate = 10

Question 3(e).
---------
Validation accuracy = 0.9171
Validation accuracy = 0.9064
Validation accuracy = 0.9168
Validation accuracy = 0.919
Validation accuracy = 0.9181
Validation accuracy = 0.9187
Validation accuracy = 0.9203
Validation accuracy = 0.9156
Validation accuracy = 0.9124
Validation accuracy = 0.9159

Maximum validation accuracy = 0.9203
Test accuracy = 0.9275
Cross entropy = 160.46474211138283
Learning rate = 10

Question 3(f).
---------
Epoch 1  Test accuracy = 0.9138
Epoch 11  Test accuracy = 0.955
Epoch 21  Test accuracy = 0.9628
Epoch 31  Test accuracy = 0.9652
Epoch 41  Test accuracy = 0.9633
Epoch 51  Test accuracy = 0.9664
Epoch 61  Test accuracy = 0.9654
Epoch 71  Test accuracy = 0.9648
Epoch 81  Test accuracy = 0.9645
Epoch 91  Test accuracy = 0.9666

Training accuracy = 0.9991666666666666
Test accuracy = 0.9651
Cross entropy = 258.83367353267425
Question 3(g).

Epoch 1   Test accuracy = 0.1109
Epoch 11  Test accuracy = 0.3199
Epoch 21  Test accuracy = 0.5929
Epoch 31  Test accuracy = 0.6989
Epoch 41  Test accuracy = 0.7186
Epoch 51  Test accuracy = 0.7735
Epoch 61  Test accuracy = 0.8092
Epoch 71  Test accuracy = 0.8384
Epoch 81  Test accuracy = 0.8521
Epoch 91  Test accuracy = 0.8611

Training accuracy = 0.8620166666666667
Test accuracy = 0.8667
Cross entropy = 28226.361088769703
Question 1(b): Neural net with 1 hidden unit
Question 1(c): Neural nets with 2 hidden units
Question 1(c): Best neural net with 2 hidden units
Question 1(d): Neural nets with 3 hidden units
Question 1(d): Best neural net with 3 hidden units
Question 1(e): Neural nets with 4 hidden units
Question 1(e): Best neural net with 4 hidden units