# Programming Assignment 2

## CSE 253: Neural Networks

## Winter 2017

# Instructions

**Due on Friday, February 3rd, 2017**

1. Please submit your assignment on Vocareum. There are two components to this assignment: written homework (Problems 1 & 2a-c), and a programming part. You will be writing a report in a conference paper format for this assignment, reporting your findings. While we won't enforce this, We prefer the report to be written using LaTeX or Word in NIPS format (NIPS is the top machine learning conference, and it is now dominated by deep nets - it will be good practice for you to write in that format!). You are free to choose an alternate format, but NIPS format is strongly recommended. The templates, both in **Word** and LaTeX are available from the 2015 NIPS format site.

2. You may use a language of your choice (Python and MATLAB are recommended). You also need to submit all of the source codes files and a *readme.txt* file that includes detailed instructions on how to run your code.

   You should write clean code with consistent format, as well as explanatory comments, as this code may be reused in the future.

3. Using the MATLAB neural network toolbox or any off-the-shelf code is strictly prohibited.

4. Any form of copying, plagiarizing, grabbing code from the web, having someone else write your code for you, etc., is cheating. We expect you all to do your own work, and when you are on a team, to pull your weight. Team members who do not contribute will not receive the same scores as those who do. Discussions of course materials and homework solutions are encouraged, but you should write the final solutions to the written part alone. Books, notes, and Internet resources can be consulted, but not copied from. Working together on homework must follow the spirit of the **Gilligan's Island Rule** (Dymond, 1986): No notes can be made (or recording of any kind) during a discussion, and you must watch one hour of Gilligan's Island or something equally insipid before writing anything down. Suspected cheating has been and will be reported to the UCSD Academic Integrity office.

## Part I

# Homework problems to be solved individually, and turned in individually

# Multi-layer Neural Networks

In this problem, we will continue classifying handwritten digits from Yann LeCun's MNIST Database. In Assignment 1, we classified the digits using a single-layer neural network with different output activation functions. (Logistic and Softmax regression). In this assignment, we are going to classify the MNIST dataset using multi-layer neural networks with softmax outputs.

**Problem**

1. In class we discussed two different error functions: sum-of-squared error (SSE) and cross-entropy error. We learned that SSE is appropriate for linear regression problems where we try to fit data generated from:

$$t = h(x) + \epsilon \tag{1}$$

Here $x$ is a $K$-dimensional vector, $h(x)$ is a deterministic function of $x$, where $x$ includes the bias $x_0$, and $\epsilon$ is random noise that has a Gaussian probability distribution with zero mean and variance $\sigma^2$, i.e. $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Suppose we want to model this data with a linear function approximation with parameter vector $w$:

$$y = \sum_{i=0}^{K} w_i x_i \tag{2}$$

Prove that finding the optimal parameter $w$ for the above linear regression problem on the dataset $D = \{(x^1, t^1), ..., (x^N, t^N)\}$ is equal to finding the $w^*$ that minimizes the SSE:

$$w^* = \text{argmin}_w \sum_{n=1}^{N} (t^n - y^n)^2 \tag{3}$$

*Hint:* This problem is solved in Bishop, Chapter 6.

2. For multiclass classification on the MNIST dataset, we previously used softmax regression with cross-entropy error as the objective function, and learned the weights of a single-layer network to classify the digits. In this assignment, we will add a hidden layer between the input and output, that consists of $J$ units with the sigmoid activation function. So this network has three layers: an input layer, a hidden layer and a softmax output layer.

*Notation:* We use index $k$ to represent a node in output layer and index $j$ to represent a node in hidden layer and index $i$ to represent a node in the input layer. Additionally, the weight from node $i$ in the input layer to node $j$ in the hidden layer is $w_{ij}$. Similarly, the weight from node j in the hidden layer to node k in the output layer is $w_{jk}$.

(a) **Derivation** Derive the expression for $\delta$ for both the units of output layer ($\delta_k$) and the hidden layer ($\delta_j$). For this problem, use the definition of $\delta$ as $\delta_i = -\frac{\partial E}{\partial a_i}$, where $a_i$ is the weighted sum of the inputs to unit $i$.

(b) **Update rule.** Derive the update rule for $w_{ij}$ and $w_{jk}$ using learning rate $\eta$, starting with the gradient descent rule:

$$w_{jk} = w_{jk} - \eta \frac{\partial E}{\partial w_{jk}} \tag{4}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial w_{jk}} \tag{5}$$

The derivative should take into account all of the outputs, so:

$$\frac{\partial E^n}{\partial a_k^n} = \sum_{k'} \frac{\partial E^n}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k} \tag{6}$$

(c) **Vectorize computation.** The computation is much faster when you update all $w_{ij}$s and $w_{jk}$s at the same time, using matrix multiplications rather than **for** loops. Please show the update rule for the weight matrix from the hidden layer to output layer and the matrix from input layer to hidden layer, using matrix/vector notation.

# Part II

# Team Programming Assignment

3. **Classification.** Classification on MNIST datatbase. Refer to your derivations from Problem 2.

(a) Read in the data from the MNIST database. You can use a loader function to read in the data. Loader functions for matlab can be found in http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip. Python helper functions can be found in https://gist.github.com/akesling/5358964.

(b) The pixel values in the digit images are in the range [0..255]. Divide by 255 so that they are in the range [0..1], and then subtract the mean over all of the pixels in each image. I.e., if the mean pixel value in an image of a "4" is 0.3, you would subtract 0.3 from each pixel value.

(c) While you should use the softmax activation function at the output level, for the hidden layer, you can use the logistic, i.e., $y_j = 1/(1 + exp(-a_j))$. This has the nice feature that $dy/da = y(1 - y)$.

(d) Check your code for computing the gradient using a small subset of data - so small, you can use one input-output pattern! You can compute the slope with respect to one weight using the numerical approximation:

$$\frac{d}{dw}E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon}$$

where $\epsilon$ is a small constant, e.g., $10^{-2}$. Compare the gradient computed using numerical approximation with the one computed as in backpropagation. The difference of the gradients should be within big-O of $\epsilon^2$, so if you used $10^{-2}$, your gradients should agree within $10^{-4}$. (See section 4.8.4 in Bishop for more details). Note that $w$ here is *one* weight in the network, so this must be repeated for *every* weight and bias. Report your results.

(e) Using the update rule you obtained from 2(c), perform gradient descent to learn a classifier that maps each input data to one of the labels $t \in \{0, ..., 9\}$ (but using a one-hot encoding). Use a hold-out set to decide when to stop training. (*Hint*: You may choose to split the training set of 60000 images to two subsets: one training set with 50000 training images and one validation set with 10000 images.) Stop training when the error on the validation set goes up, or better yet, save the weights as you go, keeping the ones from when the validation set error was at a minimum. Report your training procedure and plot your training and testing accuracy vs. number of training iterations of gradient descent. Again, by *accuracy*, we mean the percent correct on the training and testing patterns.

4. **Adding the "Tricks of the Trade."** Read the first paper on the resources page under readings, "lecun98efficient.pdf", sections 4.1-4.7. Implement the following ideas from that paper (you can do these all at once, or incrementally - no need to try all possible combinations, however).

(a) Shuffle the examples, and then use stochastic gradient descent. For the purposes of this class, use "minibatches", in which you compute the total gradient over, say, 128 patterns, take the average weight change (i.e., divide by 128), and then change the weights by that amount.

(b) Stick with the normalization you have already done in the previous part, which is a little different than what he describes in Section 4.3 - that can be expensive to compute over a whole data set.

(c) For the output layer, use the usual softmax activation function. However, for the hidden layer(s), use the sigmoid in Section 4.4. Note that you will need to derive the slope of that sigmoid to use when computing deltas.

(d) Initialize the input weights to each unit using a distribution with 0 mean and standard deviation 1/sqrt(fan-in), where the fan-in is the number of inputs to the unit.

(e) Use momentum.

(f) Comment on the change in performance, which has hopefully improved, at least in terms of learning speed.

5. **Experiment with Network Topology.** Start with the network of 4. Now, we will consider how the topology of the neural network changes the performance.

(a) Try halving and doubling the number of hidden units. What do you observe if the number of hidden units is too small? What if the number is too large?

(b) Change the number of hidden layers. Use two hidden layers instead of one. Create a new architecture that uses two hidden layers of equal size and has approximately the same number of parameters, as the previous network with one hidden layer. By that, we mean it should have roughly the same total number of weights and biases. Report training and testing accuracy vs. number of training iterations of gradient descent.