Programming Assignment 2: Multilayer Backpropagation Neural Networks

CSE 253: Neural Networks

Winter 2018

In this programming assignment, 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.

Instructions

Due on Thursday, February 1st, 2018, at 11:59 PM.

1. This assignment can be done in groups of 2 or 3 people. We expect three person teams to do exceptionally nice work.
2. Please submit your assignment on Gradescope. 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. As in the first assignment, we prefer the report to be written using LaTeX or Word in NIPS format. The templates, both in **Word** and LaTeX are available from the 2015 NIPS format site. As in PA 1, have an abstract describing what you did (briefly!), noting your accuracy results. Give an introduction, methods, results, discussion and conclusions. Don't forget the paragraphs at the end describing individual contributions and learning outcomes.
3. You may use a language of your choice (Python (using NumPy), MATLAB, or Octave are recommended). You should write clean code with consistent format, as well as explanatory comments. 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.
4. Using the MATLAB neural network toolbox or any off-the-shelf code is strictly prohibited.
5. 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 written work (Part I), and for the programming assignment (Part II), to fully contribute. Team members who do not contribute will not receive the same scores as those who do. Again, as always, please include a *separate paragraph at the end for each team member* explaining what your contribution was, and what you learned from the project. 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 (20 pts)

Problem

1. (5 pts) 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 d+1-dimensional vector, $h(x)$ is a deterministic function of x, where x includes the bias $x_0 = 1$, and ϵ is random noise that has a Gaussian probability distribution with zero mean and variance σ^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:

$$y = w^T x \tag{2}$$

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. (15 pts) 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, index j to represent a node in the hidden layer, and index i to represent an input, i.e., x_i. Hence, 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}. Using Bishop's notation, we would write the activation of hidden unit j as $z_j = g(a_j)$, where $a_j = \sum_{i=0}^{d} w_{ij} x_i$, and g represents the hidden unit activation function (e.g., the logistic), and the activation of output unit k as $y_k = g(a_k)$, where again, g represents the output unit activation function (e.g., softmax). Which g we mean should be clear from the context. This notation allows us to write the slope of the hidden unit activation function as $g'(a_j)$.

(a) **Update rule.** (10 pts) Derive the update rule for w_{ij} (the weights to the hidden layer) using learning rate η, starting with the gradient descent rule. The rule for the output units is the same as you found in Programming Assignment 1; we won't make you derive that again. To avoid too many superscripts, just assume there is only one pattern, i.e., $N = 1$. For w_{jk}, the rule is:

$$w_{jk} := w_{jk} - \eta \frac{\partial E}{\partial w_{jk}} = w_{jk} + \eta \delta_k z_j \tag{4}$$

where $\delta_k = -\frac{\partial E}{\partial a_k}$ by definition, and ":=" means assignment. Note here we are not assuming any particular form for the error, or any particular activation function.

So, again, your job is to start from this:

$$w_{ij} := w_{ij} - \eta \frac{\partial E}{\partial w_{ij}} \tag{5}$$

and the definition of $\delta_j = -\frac{\partial E}{\partial a_j}$, and arrive at this:

$$-\eta \frac{\partial E}{\partial w_{ij}} = \eta \delta_j x_i \tag{6}$$

where

$$\delta_j = g'(a_j) \sum_k w_{jk} \delta_k \tag{7}$$

(b) **Vectorize computation.** (5 pts). 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 (60 pts)

3. **Classification (20 pts)** 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 127.5 so that they are in the range [0..2], and then subtract 1 so they are in the range [-1..1]. This is not z-scoring but it's close enough for government work (an elephant is a mouse built to government specifications).

   (c) While you should use the softmax activation function at the output level, for the hidden layer, you can use the logistic, i.e., $y_j = g(a_j) = 1/(1 + exp(-a_j))$. This has the nice feature that $dy/da = y(1 - y)$. Start with 64 hidden units.

   (d) Check your code for computing the gradient using a small subset of data - e.g., just a few examples. You can compute the slope with respect to one weight using the numerical approximation:

$$\frac{\partial E^n}{\partial w_{ij}} \approx \frac{E^n(w_{ij} + \epsilon) - E^n(w_{ij} - \epsilon)}{2\epsilon} \tag{8}$$

   where ϵ is a small constant, e.g., 10^{-2}, and $E^n(w_{ij} + \epsilon)$ refers to the error on example x^n when weight w_{ij} is set to $w_{ij} + \epsilon$. Compare the gradient computed using numerical approximation with the one computed by backpropagation. The difference between the gradients should be within big-O of ϵ^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_{ij} here is *one* weight in the network, so to check your gradient computation you really should repeat this calculation for *every* weight and bias and every example! However, we'll be happy if you do it for some output biases, input biases, and a few weights from input to hidden and hidden to output for a few examples. Report your results.

   (e) Using the update rule you obtained from 2(a), perform mini-batch gradient descent to learn a classifier that maps each input data to one of the labels $t \in \{0, ..., 9\}$ (using a one-hot encoding). Start with initially random weights and a mini-batch size of 128. 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 50,000 training images and one validation set with 10,000 images.) Check the error on the hold-out set after each complete run through the 50,000 patterns (holding the weights fixed!). 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. Describe your training procedure so someone reading your report could replicate what you've done. Plot your training, holdout, 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. Also plot the loss.

4. **Adding the "Tricks of the Trade." (20 pts)** Read "lecun98efficient.pdf", sections 4.1-4.7. The paper can be found on the resources page under readings. Implement the following ideas from that paper. Do these incrementally, i.e., report your results, then add another trick, and report your results again, so that you can see how much each change improves the learning. 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. For the output layer, use the usual softmax activation function, and initially use the logistic activation function for the hidden units.

   (a) If you didn't shuffle the examples in the first part after each epoch, try that now. Again, for the purposes of this class, we 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) Now, for the hidden layer, use the sigmoid in Section 4.4. Note that you will need to derive the slope of that sigmoid to use when computing deltas.

   (c) 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.

   (d) Use momentum, with an α of 0.9.

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

5. **Experiment with Network Topology (20 pts)** Start with the final network of 4. Now, we will consider how the topology of the neural network changes the performance.

   (a) (10 pts) 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) (10 pts) Increase the number of hidden layers, e.g., 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. Plot training, holdout, and testing accuracy vs. number of training iterations of gradient descent. Repeat this plot for the loss.

   (c) (up to 5 extra pts) At this point, you are welcome to have a little fun and try different things of your own choosing to improve accuracy. (Hint: Among the things to try, do *not* try to implement convnets on your own!!!). E.g., you could try different topologies, you could google Nesterov momentum and implement it, or use ReLU units, etc. This is up to you.