CS189: Introduction to Machine Learning

Homework 6

Due: 11:59 pm on Thursday, April 23, 2015

Neural Networks for MNIST Digit Recognition

In this homework, you will implement neural networks to classify handwritten digits using raw pixels as features. You will be using the MNIST digits dataset that you used in previous homework assignments.

The state-of-the-art error rate on this dataset using deep convolutional neural networks is around 0.5%. For this assignment, you should, with appropriate parameter settings, get approximately or better than 6% error using a neural network with one hidden layer.

You should expect around an hour of full training time for neural networks. Please start this assignment early!

Multi-Layer Feed Forward Neural Network

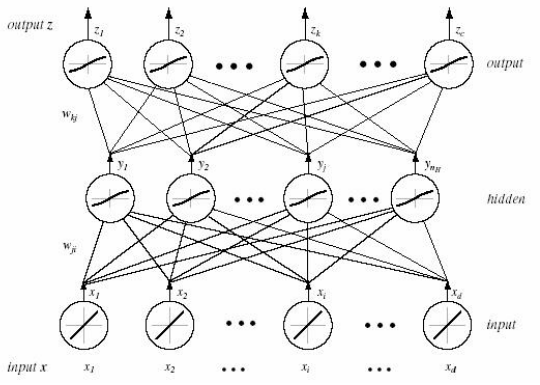

Figure 1: Example of a multiple layer neural network with one hidden layer.

In this assignment, you are asked to implement a neural network with one hidden layer. Fig 1 is an example of multi-layer feed forward neural network with one hidden layer. You will have a total of three layers: the input layer, the hidden layer, and the output layer. Pay careful attention to the following implementation details presented.

1. You will be using a hidden layer of size 200. Let $n_{in} = 784$, the number of features for the digits class. Let $n_{hid} = 200$, the size of the hidden layer. Finally, let $n_{out} = 10$, the number of classes. Then, you will have $n_{in} + 1$ units in the input layer, $n_{hid} + 1$ units in the hidden layer, and n_{out} units in the output layer.

   The input and hidden layers have one additional unit which always takes a value of 1 to represent bias. The output layer size is set to the number of classes. Each label will have to be transformed to a vector of length 10 which has a single 1 in the position of the true class and 0 everywhere else.

2. The parameters of this model are the following:

   - $\mathbf{W^{(1)}}$, a $(n_{in} + 1)$-by-n_{hid} matrix where the (i, j)-entry represents the weight connecting the i-th unit in the input layer to the j-th unit in the hidden layer. Note: there is an additional row for weights connecting the bias term to each unit in the hidden layer.
   - $\mathbf{W^{(2)}}$, a $(n_{hid} + 1)$-by-n_{out} matrix where the (i, j)-entry represents the weight connecting the i-th unit in the hidden layer to the j-th unit in the output layer. Note: again there is an additional row for weights connecting the bias term to each unit in the output layer.

3. You will be expected to train and run your neural network using both mean-squared error and cross-entropy error as your loss function (one network for each loss function). Suppose y is the ground truth label (using the same 1-of-n_{out} encoding as stated previously in point 1) and $h(x)$ is a vector containing each value of the units in the output layer given the feature vector x. Then, the mean-squared error is

$$J = \frac{1}{2} \sum_{k=1}^{n_{out}} (y_k - h_k(x))^2$$

   The cross-entropy error is given as

$$J = -\sum_{k=1}^{n_{out}} [y_k \ln h_k(x) + (1 - y_k) \ln(1 - h_k(x))]$$

4. All hidden units should use the tanh activation function as the choice of non-linear function and the output units should use the sigmoid function as its choice. Remember the sigmoid function is given as

$$g(z) = \frac{1}{1 + e^{-z}}$$

5. You will be using stochastic gradient descent to update your weights.

Problems

1. Derive the stochastic gradient descent updates for all parameters ($\mathbf{W^{(1)}}$ and $\mathbf{W^{(2)}}$) for both mean-squared error and cross-entropy error as your loss function given a single data point (x, y). Use tanh activation function for the hidden layer units and the sigmoid function for the output layer units. To do this, you must compute the partial derivative of J with respect to every $\mathbf{W_{ij}^{(1)}}$ and $\mathbf{W_{ij}^{(2)}}$. Use the notation provided above. Please be clear when adding new notation used in the derivation. The following information might be helpful to you.

   - The derivative of $\tanh(z)$ is $1 - \tanh^2(z)$.
   - The derivative of the sigmoid function $g(z)$ is $g(z)(1 - g(z))$.

2. Train this multi-layer neural network on full training data using stochastic gradient descent. Predict the labels to the test data and submit your results to Kaggle. Please also report the following:

   - Parameters that you tuned including learning rate, when you stopped training, how you initialized the weights
   - Training accuracy and validation accuracy
   - Running-time (Total training time)
   - Plots of total training error and classification accuracy on training set vs. iteration. If you find that evaluating error takes a long time, you may compute the error or accuracy every 1000 or so iterations.
   - Comment on the differences in using the two loss functions. Which performs better?

3. **(Optional)** After you have implemented this basic multi-layer neural network and have reported all results, you may do some of the following or anything else to improve your neural network for your Kaggle submission. Please include any of these extra features you have implemented in your report.

   - Use more than one hidden layer or change the number of hidden layer units. When using more layers (deeper networks) you may want to look into initializing your weights using auto-encoders and using ReLU (see next bullet point) for faster convergence and better performance.
   - Use ReLU (rectified linear units) as your non-linear activation function for your hidden layer units. The rectifier is given as

     $$f(x) = \max(0, x)$$

     You will have to re-derive stochastic gradient descent updates. It is largely observed that using ReLU as the non-linear activation function allows the neural network to converge faster. Try to see if that is true for this case.
   - Rather than using stochastic gradient descent, implement using mini-batch or batch gradient descent. If you implement mini-batch, try batches of sizes between 200-500.

Submission

For this homework, you need to submit the code, a README describing how to run your code, and a write-up that includes all answers to questions (derivations, explanations, and plots) and references to all external sources you used (articles, papers, books). Submissions are through **bCourses** and **Kaggle**.

Implementation Tips

Suggested Specifications

These are just guidelines on one way you could implement neural networks. Your implementation should consist of two main methods.

def trainNeuralNetwork(images, labels, params*)
images: training set (features)
labels: training set (labels)
params: hyper-parameters, i.e. learning rate η, choice of cost function, etc.

1. Initialize all weights, $\mathbf{W^{(1)}}, \mathbf{W^{(2)}}$ at random
2. while (some stopping criteria):
3. pick one data point (x, y) at random from the training set
4. perform forward pass (computing necessary values for gradient descent update)
5. perform backward pass (again computing necessary values)
6. perform stochastic gradient descent update
7. return $\mathbf{W^{(1)}}, \mathbf{W^{(2)}}$

There are several options for the stopping criteria. You may choose to have a maximum number of iterations or look at the change in the cost function over the full training set and stop when this change is smaller than some small constant ϵ. Note that since we are using stochastic gradient descent, your change in the value of the cost function may be positive so be careful how you set up your stopping criteria. One suggestion for this problem is to find the average change in the value of the cost function over the last k iterations. You can also look at the magnitude of the gradient change.

def predictNeuralNetwork(weights, images)
weights: trained weights, i.e. $\mathbf{W^{(1)}}, \mathbf{W^{(2)}}$
images: test set (features)

1. Compute labels of all images using the weights, $\mathbf{W^{(1)}}, \mathbf{W^{(2)}}$
2. return labels

You may also want to make separate functions for the sigmoid function, each loss function, and the gradients of the loss functions or implement other helper functions. We

will leave it up to your preference! Feel free to implement however you would like but please make sure to include instructions on how to run your code in your README.

Initialization of Weights

Make sure you initialize your weights with random values. This allows us to break symmetry that occurs when all weights are initialized to 0. Some ways to do this are to initialize by drawing values from a uniform distribution from $[-\epsilon, \epsilon]$ or from a Gaussian distribution with mean 0 and variance ϵ^2 where ϵ is some small fixed constant.

Preprocessing Data

Preprocessing your data may be helpful. One way to preprocess your data is to normalize all your features. You are encouraged to try other preprocessing methods such as ones presented in previous homework assignments.

Step Size and Convergence

Inappropriate step sizes (learning rate) will cause you a lot of trouble for neural networks. With step sizes that are too small, stochastic gradient descent on neural networks can easily get trapped in bad local minima. With step sizes that are too big, stochastic gradient descent on neural networks will diverge to a really poor classifier. To help you avoid these situations, we encourage you to examine either the magnitude of the gradient or the current cost on the whole training set every some fixed amount of iterations.

Also, you may want to make the step size be inversely proportional to some function of the iteration count, i.e. $\eta \propto \frac{1}{f(t)}$ where t is the iteration count you are currently at.

Vectorization

Please vectorize your code when possible (to save you hours of time). Although you must be used to vectorizing all your implementation by now, this is another reminder to make sure you vectorize your matrix and vector computations such as matrix-matrix multiplication or element-wise vector-vector multiplication. Neural networks will take longer than other assignments to train. Therefore, it is **necessary** that you vectorize your computations in order to take advantage of the highly optimized operations provided to you by Matlab or Numpy. A simple rule-of-thumb is to avoid explicit for-loops when possible.

Gradient Checking

Back-propagation is quite tricky to implement correctly. We strongly encourage you to verify the gradients computed by your back-propagation code by comparing its output to gradients computed by finite differences. Let $f : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary function (for example, the loss of a neural network with d weights and biases), and let a be a

d-dimensional vector. You can approximately evaluate the partial derivatives of f as:

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\dots, a_k + \epsilon, \dots) - f(\dots, a_k - \epsilon, \dots)}{2\epsilon}$$

where ϵ is a small constant. As ϵ approaches 0, this approximation theoretically becomes exact, but becomes prone to numerical precision issues. The choice $\epsilon = 10^{-5}$ usually works well.

This technique is extremely slow, so don't forget to remove it from your code after you've used it to verify your back-propagation implementation!

Checkpointing

Since training these neural networks takes lots of time, we encourage you to write your code so that it saves its progress every some fixed amount of iterations. This way, if your program somehow gets terminated, you can resume computation where it left off.