CS189: Introduction to Machine Learning

Homework 4

Due: 12:00 p.m. Thursday, November 3, 2016. **NOON. NOT MIDNIGHT**.

Submission: **Gradescope and Kaggle**

Submission Instructions

You will submit this assignment to **Gradescope**. There will also be a **Kaggle** competition.

In your submission to **Gradescope**, include separately:

1. A pdf writeup with answers to all the parts of the problem and your plots. Include in the pdf a copy of your code.
2. A zip of all your code.

Submit to **Kaggle**

3. A csv file with your best predictions for the examples in the test set, just like in previous homeworks.

Note: The Kaggle invite links and more instructional details will be posted on Piazza.

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.
You can expect up to an hour of training time for neural networks if your parameters are not well tuned or your code is not optimized.

There is almost no skeleton code. Please start this assignment early!

Feed Forward Neural Network

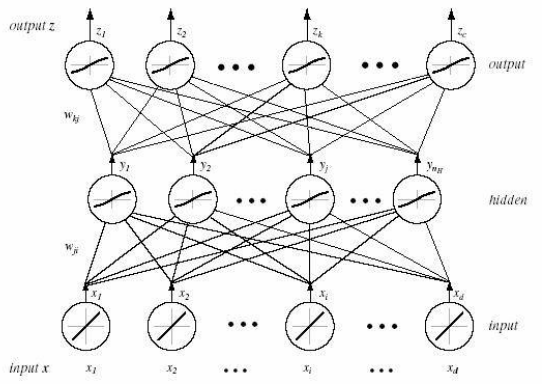

Figure 1: Example of a neural network with one hidden layer.

In this assignment, you will 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. Use 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 a bias term (its weights are still learned). The output layer size is set to the number of classes. As before, each label will have to be one-hot encoded (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:

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

3. The output layer should use the softmax function as the activation function, shown below. This should look similar to the sigmoid or logistic function, but generalized to the multi-class case.

$$g_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_{k=1}^{n_{out}} e^{z_k}}$$

   You will need to use the following facts:

$$\frac{\partial g_j}{\partial z_j} = g_j(\mathbf{z})(1 - g_j(\mathbf{z}))$$

$$\frac{\partial g_j}{\partial z_i} = -g_i(\mathbf{z})g_j(\mathbf{z}) \text{ for } i \neq j$$

   The softmax function normalizes the values fed into each output unit so that the final output vector corresponds to class probabilities (the kth output unit representing the probability of the kth class).

   Hidden units should use the rectified linear (ReLU) activation function, given as:

$$g(z) = \max(0, z)$$

   You may set the derivative at 0 to be 0.

4. You will be expected to train your neural network using cross-entropy loss (also known as logistic loss) as your loss function. Suppose y is the ground truth label (using the same one-hot encoding as stated previously) and $z(x)$ is a vector containing each value of the units in the output layer given the feature vector x. The cross-entropy loss is given as:

$$J = -\sum_{k=1}^{n_{out}} y_k \ln z_k(x)$$

   This should look similar to what we've seen earlier for logistic regression, but generalized to the multi-class case.

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

Problems

1. Derive the stochastic gradient descent backpropagation updates for all parameters (V and W) given a single data point (x, y). To do this, you must compute the partial derivative of J with respect to every V_{ij} and W_{ij}, using the backpropagation algorithm. Use the notation provided above. Please be clear when adding new notation used in the derivation. Show your work!

2. MNIST provides 60k labeled samples and we have provided a test set with 10k unlabeled samples. You must replace the original MNIST test set with our new test set file, `t10k-images-idx3-ubyte`.

   In your code, split the MNIST labeled data into a 50k training set and a 10k validation set (write this code yourself, do not use sklearn or equivalent). Use the validation set to report validation accuracy and to tune hyperparameters in the next problem.

3. Implement and 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 report:

   - Parameters that you tuned including learning rate, when you stopped training, how you initialized the weights, etc.
   - Final training accuracy and validation accuracy
   - Running-time (Total training time, in hours/minutes)
   - Plots of total training loss and classification accuracy on training set vs. iteration (two plots). You may want to compute the loss and accuracy only every $10,000$ or so iterations.
   - Your Kaggle score.

   You must implement this neural network from scratch in Python, using only basic linear algebra / utility functions (numpy and scipy). You cannot use any framework that does backpropagation for you.

4. **(Optional)** After you have implemented this basic multi-layer neural network and have reported all results, you may go above and beyond to improve your neural network for your Kaggle submission. Please mention any of these extra features you have implemented in your report. Some ideas:

   - Use more than one hidden layer or increase the number of hidden layer units.
   - Implement $l2$ regularization. This is known as weight decay, because it simply involves subtracting λw from each weight w during every iteration.
   - Implement mini-batch gradient descent, different optimization methods, momentum, different initialization schemes, or more complicated regularization

schemes, etc. See the last section for ideas.

- Add one or more convolutional layers before the hidden layer. See the last section for ideas.

The state-of-the-art error rate on this dataset using an ensemble of deep convolutional neural networks is around 0.5%. For this assignment, you should, with appropriate parameter settings, get approximately or better than 4% error using a neural network with one hidden layer. Without any additional tricks, your GSIs are able to get less than 3% test error in less than 3 minutes of training on a Macbook Pro.

For trying to win Kaggle, your GSIs are able to get 1.1% error using an ensemble of 3 single-hidden layer networks using data augmentation and other tricks.

Implementation Details and Tricks

Suggested Structure

This is the basic gist of how to structure your code:

def trainNeuralNetwork(data, labels, params*)
images: training set (features)
labels: training set (labels)
params: hyper-parameters, i.e. learning rate η, decay rate, etc.

1. Initialize all weights, V, W using some initialization scheme
2. while (some stopping criterion):
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. store V, W

def predictNeuralNetwork(data)
data: test set (features)

1. Compute labels of all images using the weights, V, W
2. return labels

Stopping Criteria

There are several options for the stopping criterion:

1. Set a maximum number of iterations.
2. Stop when training loss stops falling. Because you are doing stochastic gradient descent, your change in the value of the cost function may be positive for some iterations and negative for others so be careful how you set up this criterion.
3. Stop when validation loss or error stops falling (same issue with stochasticity in criterion 2)
4. Stop when you feel like it and save the neural net model so you can resume training (see checkpointing). This is the most popular in practice!

Preprocessing Data

You **must** center and normalize all your features. You are encouraged to try other preprocessing methods. Remember to **shuffle** the data! If data is not shuffled and a neural network is trained on data ordered by label (e.g. the network sees all the 0 digits first), the network will not learn anything because at first it will simply learn to always output 0.

Step Size and Convergence

The *single most important set of hyperparameters to tune* is the learning *schedule*. This is a combination of learning rate and how you decay your learning rate. Step sizes (learning rate) cannot be too small or too large. With step sizes that are too small, training will be slow and SGD can easily get trapped in bad local minima. With step sizes that are too large, training will diverge. You **must** decay your learning rate. Typically people will decay the learning rate by multiplying by some constant γ every constant number of epochs (> 1). γ is often 0.5 or 0.9.

An epoch is one full-pass over the data. One epoch of SGD would be n iterations. One epoch of batch gradient descent would be 1 iteration.

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. The precise initialization method has a strong effect on the number of iterations it takes to converge. The most basic method is to initialize from a Gaussian distribution with mean 0 and variance σ^2 where σ^2 is some small fixed constant < 1.

More complex neural networks use other methods of initialization which you are free to look up. These might not be useful for the shallow fully-connected neural network you will be implementing, so more advanced initializations schemes might have little effect.

Numerical Stability

For computing cross-entropy, you might need to lower-bound the values that you take the log of, since you cannot compute $\log 0$.

When computing the softmax, you might need to use the exponential normalization trick for numerical stability (`http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/`).

Vectorization

Please vectorize your code when possible. This can save you hours of training time. Although you should be used to vectorizing your code by now, this is a reminder to make sure you vectorize your backpropagation updates as matrix and vector computations such as matrix-matrix multiplication or element-wise operations. Neural networks will take much longer than other assignments to train if your operations are not optimized. Therefore, it is important that you vectorize your computations in order to take advantage of the highly optimized operations provided to you by numpy. It is possible (and ideal) to implement the *entire* assignment using only a single for-loop that loops over gradient descent iterations.

Gradient Checking

Backpropagation can be tricky to implement correctly. If you network does not train and you have carefully looked at your code and tried tuning the learning schedule, one way to verify the gradients computed by your code is 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 theoretically becomes exact, but becomes prone to numerical precision issues. The choice $\epsilon = 10^{-5}$ works well. This technique is extremely slow, so don't forget to remove it from your code after you've used it.

Checkpointing

Since training these networks takes lots of time, we encourage you to write your code so that it saves its progress in a file when you terminate the program (see the Python `signal` package for how to capture an interrupt signal) or every fixed amount of iterations, and

to write code that loads this file and allows you to resume training. This way, you can terminate your program safely, save multiple models, etc.

You might find it useful to look into the Python `pickle` module, or `numpy.save`. This allows you to save and load arbitrary Python or numpy objects (such as your neural network weights) as files.

Extra Bells and Whistles

Kaggle Improvement

The two best ways to increase your Kaggle score are:

1. Implement *data augmentation*, where you artificially generate new data points by slightly jittering, stretching, and/or skewing your training data.
2. Train multiple neural networks and *ensemble* their predictions. This means that given a test data point, the predicted label is a majority vote of multiple neural networks. In terms of the bias-variance decomposition, this corresponds to reducing variance.

Optimization

The single best way to speed up your code is to use mini-batch gradient descent instead of SGD. In mini-batch gradient descent, you sample k data points instead of 1 data point and average the gradient update over those data points. If $k = n$ this becomes batch gradient descent. Typically k is some number between 16 and 256 (50 is a good starting point). Note that using larger batches uses more memory.

Another useful trick is momentum, where you add a fraction of the previous gradient update to the current gradient update. A more advanced feature that the adventurous might attempt to implement is batch normalization.

Note that if you choose to implement momentum or mini-batches this affects your learning rate. Specifically, momentum increases the magnitude of your gradient (so you have to decrease the learning rate). With mini-batches you can usually increase your learning rate since gradient descent is smoother.

Regularization

Common forms of regularization include dropout (randomly zeroing out hidden units, see: `https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf`) and $l2$ regularization, also known as weight decay. These tend to only be useful for nets with more than one hidden layer, however.

Convolutional Neural Networks

The truly adventurous may attempt to implement convolutional layers. Remember that convolutional filters simply mean that weights are shared across different locations. A simple implementation of the backpropagation update for a convolutional layer weight would simply involve averaging the weight's partial derivative over all the places a weight is shared, using a for-loop.

A more efficient implementation involves treating the image as a single vector, and treating convolution as a matrix multiplication on that vector. The details are left to you.