Machine Learning 6.867 - Pset 3

November 9, 2015

1 Multi-Class SVM

Due to time constraint we did not do this problem.

2 Neural Networks

Neural networks are used in machine learning to make predictions, similar to logistic regression, SVM, or regression. We can represent neural networks using a graph with nodes and edges (see Bishop figure 5.1). Assume that we observe data $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)}), n = 1, \ldots, N$, where $\mathbf{x}^{(n)} \in \mathbb{R}^{D+1}$ (including the bias term) and $\mathbf{y}^{(n)} \in \{0,1\}^K$. Let $(\mathbf{x}, \mathbf{y}) = ([x_0, x_1, \ldots, x_D], [y_1, \ldots, y_K])$ be a general observation, where $x_0 = 1$ is a constant term for the bias. In this example, only one term of $[y_1, \ldots, y_K])$ can be one; the rest are zero. We create nodes for each of the features x_i, referred to as *inputs*, and nodes for each of the class labels y_i, referred to as *outputs*. Next, we introduce a series of nodes in the middle of the graph, called *hidden units*, and we draw edges connecting the **inputs** $\rightarrow$**hidden units**$\rightarrow$ **outputs**. The key idea in neural networks is that we can model the hidden units as functions of the inputs, and model the outputs as functions of the hidden units.

For 2-layer neural networks, we have one layer of hidden units denoted by $[z_0, z_1, \ldots, z_M]$. There are weights $\mathbf{w}^{(1)}_{ji}$ for each edge $x_i \rightarrow z_j$ and $\mathbf{w}^{(2)}_{kj}$ for each edge $z_j \rightarrow y_k$, which are unknown and will be learned through training the model. However, there are no edges from the inputs to z_0, because we assume $z_0 = 1$ is a constant term for the bias. Let $\sigma(\cdot)$ denote the logistic function, which we will use as the *activation function* for both the hidden and output layers of our neural network. The predicted value for output k given parameters $\mathbf{w} = (\mathbf{w}^{(1)}, \mathbf{w}^{(2)})$ and input $\mathbf{x}$ will be:

$$h_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w^{(2)}_{kj} \sigma\left(\sum_{i=0}^{D} w^{(1)}_{ji} x_i\right) + w^{(2)}_{k0}\right). \tag{1}$$

2.1 Implementation

2.1.1 Gradient Descent

We implemented a 2-layer regularized neural network using gradient descent. The loss function, which is the negative log-likelihood, is equal to:

$$\ell(\mathbf{w}) \equiv \sum_{n=1}^{N} \sum_{k=1}^{K} \left[-y^{(n)}_k \log(h_k(\mathbf{x}^{(n)}, \mathbf{w})) - (1 - y^{(n)}_k) \log(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w}))\right] \tag{2}$$

Let $\tilde{\mathbf{w}}^{(1)} = (\mathbf{w}^{(1)}_{ji})_{i \neq 0}$ and $\tilde{\mathbf{w}}^{(2)} = (\mathbf{w}^{(2)}_{kj})_{j \neq 0}$. We add a regularizer term, and the final cost function becomes:

$$J(\mathbf{w}) \equiv \ell(\mathbf{w}) + \lambda(\|\tilde{\mathbf{w}}^{(1)}\|^2_F + \|\tilde{\mathbf{w}}^{(2)}\|^2_F), \tag{3}$$

where $\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}$ is the matrix Frobenius norm, and λ is a fixed parameter chosen via cross-validation. Note that we exclude the bias terms from our regularizer in order to avoid penalty if the data is shifted. To take the derivative of this function, we introduce some more notation:

$$a_j^{(1)}(\mathbf{x}) \equiv \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{4}$$

$$z_j(\mathbf{x}) \equiv \begin{cases} \sigma(a_j^{(1)}(\mathbf{x})), & j = 1, \dots, M \\ 1, & j = 0 \end{cases} \tag{5}$$

$$a_k^{(2)}(\mathbf{x}) \equiv \sum_{j=0}^{M} w_{kj}^{(2)} z_j(\mathbf{x}) \tag{6}$$

We refer to $a_j^{(1)}(\mathbf{x})$ and $a_k^{(2)}(\mathbf{x})$ as the *activations* for a fixed value of $\mathbf{x}$. It follows that $h_k(\mathbf{x}, \mathbf{w}) = \sigma(a_k^{(2)}(\mathbf{x}))$. Taking partial derivatives of J with respect to $\mathbf{w}_{kj}^{(2)}$ and $\mathbf{w}_{ji}^{(1)}$ for $i \neq 0$ and $j \neq 0$, we find:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{kj}^{(2)}} = 2\lambda \mathbf{w}_{kj}^{(2)} + \frac{1}{N} \sum_{n=1}^{N} \left(\frac{-y_k^{(n)}}{h_k(\mathbf{x}^{(n)}, \mathbf{w})} + \frac{1 - y_k^{(n)}}{1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})} \right) h_k(\mathbf{x}^{(n)}, \mathbf{w})(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})) z_j(\mathbf{x}^{(n)}) \tag{7}$$

$$= 2\lambda \mathbf{w}_{kj}^{(2)} + \frac{1}{N} \sum_{n=1}^{N} \left(-y_k^{(n)}(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})) + (1 - y_k^{(n)}) h_k(\mathbf{x}^{(n)}, \mathbf{w}) \right) z_j(\mathbf{x}^{(n)}) \tag{8}$$

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{ji}^{(1)}} = 2\lambda \mathbf{w}_{ji}^{(1)} + \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(-y_k^{(n)}(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})) + (1 - y_k^{(n)}) h_k(\mathbf{x}^{(n)}, \mathbf{w}) \right) w_{kj}^{(2)} z_j(\mathbf{x}^{(n)})(1 - z_j(\mathbf{x}^{(n)}))\mathbf{x}^{(n)} \tag{9}$$

For the bias terms, there is no regularizer penalty, so we obtain:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{k0}^{(2)}} = \frac{1}{N} \sum_{n=1}^{N} \left(-y_k^{(n)}(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})) + (1 - y_k^{(n)}) h_k(\mathbf{x}^{(n)}, \mathbf{w}) \right) \tag{10}$$

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_{j0}^{(1)}} = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(-y_k^{(n)}(1 - h_k(\mathbf{x}^{(n)}, \mathbf{w})) + (1 - y_k^{(n)}) h_k(\mathbf{x}^{(n)}, \mathbf{w}) \right) w_{kj}^{(2)} z_j(\mathbf{x}^{(n)})(1 - z_j(\mathbf{x}^{(n)}))\mathbf{x}^{(n)} \tag{11}$$

Therefore, we have calculated analytic expressions for the gradients of J with respect to the different groups of parameters $\mathbf{w}^{(2)}$ and $\mathbf{w}^{(1)}$. From these expressions, we coded a function in MATLAB to train a our neural network using *batch gradient descent*. The function takes in all the input data $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)}), n = 1, \dots, N$ at once, and does gradient descent on the total objective value over the N data points each time, until convergence. Model parameters are regularization parameter λ, number of nodes in the hidden layer M, and initial solution $(\mathbf{w}_{init}^{(1)}, \mathbf{w}_{init}^{(2)})$. The function outputs the optimal solution $\mathbf{w} = (\mathbf{w}^{(1)}, \mathbf{w}^{(2)})$ that minimizes the final cost function $J(\mathbf{w})$. The formula for the update is as follows:

$$\mathbf{w}_{new}^{(1)} = \mathbf{w}_{old}^{(1)} - \eta \nabla_{w^{(1)}} J(\mathbf{w}_{old}), \qquad \mathbf{w}_{new}^{(2)} = \mathbf{w}_{old}^{(2)} - \eta \nabla_{w^{(2)}} J(\mathbf{w}_{old}) \tag{12}$$

Because the objective function is non-convex, we run our training function with multiple initial solutions to avoid reporting a single local optimum. After running multiple times, we report the optimal solution $\mathbf{w} = (\mathbf{w}^{(1)}, \mathbf{w}^{(2)})$ that minimizes $J(\mathbf{w})$ over all trials.

2.1.2 Stochastic Gradient Descent

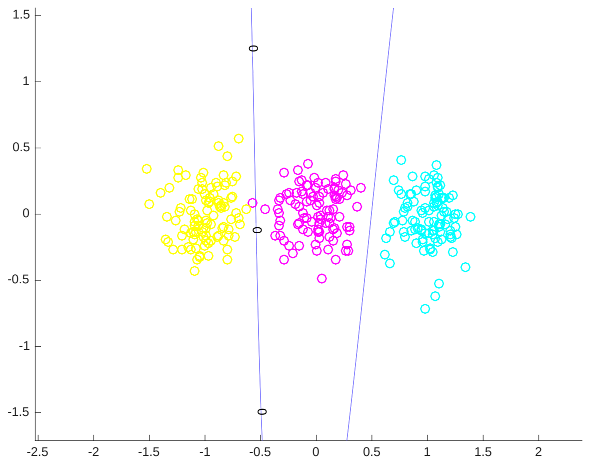
We implemented an alternate function in MATLAB to train our neural network using *stochastic gradient descent*. We first take one training point and run gradient descent on the objective function on this point. With the output $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$ as new initial values, we then take a second point and run gradient descent on the objective function on this point. Repeat the process until convergence. If necessary, we can reshuffle the data and loop through the data multiple times.

2.2 Computational Results

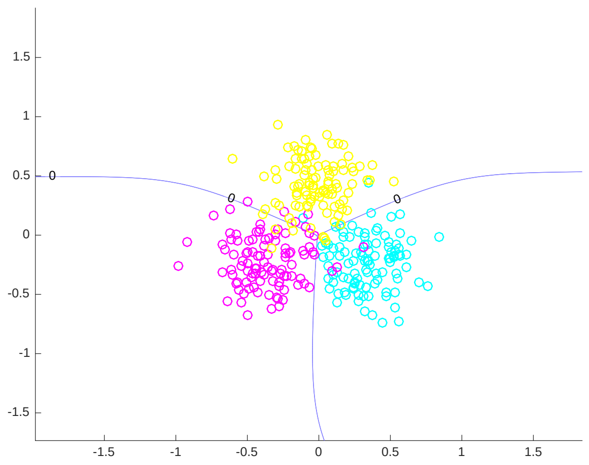
We tested our neural network on various data sets, including toy problems and real-world MNIST data on hand-written digits. Both batch gradient descent and stochastic gradient descent methods are tested on the sets.

2.2.1 Toy Problems

The first toy problem is on a linearly separable 3-class data. The decision boundary of a good ANN classifier we found is illustrated in Figure 1a. We present the validation set accuracy results in Table 1, under different number of units in the hidden layer and the trade-off parameter in regularization. We see from the table that we achieve highest validation accuracy under $\lambda = 0.001$ and $M = 2$. This gives us test accuracy of 99.3%.



(a) Toy Data 1



(b) Toy Data 2

Figure 1: Plots of decision boundaries from Artificial Neural Networks on training toy data.

The second problem we considered is another 3-class data with points that are more closely spaced. The decision boundary of our ANN classifier is presented in Figure 1b. Similarly, the accuracy from cross-validation is given in Table 2. The best accuracy is obtained when $M = 4$ and $\lambda = 0.001$. This gives the testing set accuracy of 91.3%.

Table 1: Validation accuracy on Toy Data 1 under different parameter specification

λ / M	2	3	4	5	6
0	0.9933	0.9900	0.9933	0.9900	0.9933
0.001	0.9967	0.9833	0.9867	0.9800	0.9933
0.01	0.9833	0.9733	0.9667	0.9833	0.9900

Table 2: Validation accuracy on Toy Data 2 under different parameter specification

λ / M	2	3	4	5	6
0	0.9100	0.9167	0.9267	0.9167	0.9267
0.001	0.8967	0.8233	0.9333	0.9167	0.9333

There are some computational considerations when choosing the parameters for the model. In particular, it is very important to choose reasonable initial values for $\mathbf{w}$ and the step size. We chose the initial values to be a random numbers between $[-0.05, 0.05]$ because when the initial values are close to zero, the gradient on the logistic function would have a sizable magnitude. The step size is chosen to satisfy the Robbins-Monro criterion: we chose $\eta = ss/(iter + 100)^{0.6}$, where $iter$ is the number of current iteration, and ss is chosen to be 25 for these toy data sets. Regarding the convergence criteria, the lower it is, the better accuracy we have generally; however the longer run-time it takes as well.

Between batch and stochastic gradient descent implementations, we found that the batch method is more sensitive to the choice of initial $\mathbf{w}$. We think this might be because when doing stochastic gradient descent, some data points can pull the variables from local minimum, whereas batch gradient descent tends to get stuck somewhere more easily. The numbers we present here are from stochastic gradient descent, but can also replicated by batch method if the parameters are chosen carefully.

2.2.2 MNIST Data