6.867 Homework 3: Neural Networks

November 10, 2015

1 Approach

So far we have only discussed machine learning problems in which we have specified the basis functions we intend to use. However, since the optimal basis functions could be complicated and non-intuitive, such as in the case of images, we may want an approach that allows us to learn the basis functions. Such an approach is Artificial Neural Networks(ANN).

The idea of a neural network replaces the known basis functions with features, z_j which are parametric functions of activations, a_j learned from the input data in the first layer of learning, and then utilize a second layer to learn the relationship of the output data from those selected features.

$$a_j^{(1)} = \sum_{i=1}^{N} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{1}$$

$$z_j = g(a_j) \tag{2}$$

Where $g(x)$ is a nonlinear map of input to $(0,1)$, generally tanh or sigmoid, and a_j is an activiation, which depends on the data, $x_i,\ i = \{1, ...N\}$ and learned weights $w_j i^{(1)}$ and bias $w_0^{(1)}$.

The second layer learns in a similar fashion and can be describe for K output classes and M features in the hidden layer is thus

$$a_k^{(2)} = \sum_{j=1}^{M} w_j k^{(2)} z_j + w_{k0}^{(2)} \tag{3}$$

$$f_k = \tilde{g}(a_k^{(2)}) \tag{4}$$

Where $\tilde{g}(x)$ is the activation function, another nonlinear map, not necessarily the same as $g(x)$ and f_k is really our prediction $h_k(x, w)$. However for this problem, we will consider them equal, both sigmoids,

$$\tilde{g}(x) = g(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

1.1 Gradient Calculation

First of all we we be utilize stochastic gradient descent to train our network by updating the weights with each incoming new data point, as opposed to a batch implementation. To do so, we must first find the gradient of our cost function, J which depends on the loss function $l(w)$ but also applies a regularization of both sets of weights, $w^{(1)}, w^{(2)}$

$$J = l(w) + \lambda(||w^{(1)}||_F^2 + ||w^{(2)}||_F^2) \tag{6}$$

Where the norms above are the matrix Frobenius norm and the loss function, $l(w)$ is the negative log-likelihood given by

$$l(w) = \sum_{i=1}^{N} \sum_{k=1}^{K} -y_k^{(i)} \log(h_k(x^{(i)}, w)) - (1 - y_k^{(i)}) \log(1 - (h_k(x^{(i)}, w)) \tag{7}$$

Taking the partial differential of the cost with respect to each variable, $w^{(1)}, w^{(2)}$ leads to the gradients. For the gradient with respect to the second layer weights

$$\nabla_{w_k^{(2)}} J(w) = \frac{\partial J}{\partial a_{nk}^{(2)}} (\nabla w_k^{(2)} a_{nk}^{(2)}) + 2\lambda w_k^{(2)} \tag{8}$$

$$= \frac{\partial J_n}{\partial h_{nk}} (\tilde{q}'(a_{nk}^{(2)})) z_n + \lambda w_k^{(2)} \tag{9}$$

algebraically we can solve for the partial of the cost with respect to each prediction where

$$\frac{\partial J_n}{\partial h_{nk}} = -\frac{y_k^{(n)}}{h_k(x^{(n)}, w)} + \frac{1 - y_k^{(n)}}{1 - h_k(x^{(n)}, w)} \tag{10}$$

Introducing a new variable, $\delta_{nk}^{(2)}$,

$$\delta_{nk}^{(2)} = \frac{\partial J}{\partial a_{nk}^{(2)}} \tag{11}$$

Looking at the partial with respect to the first layer

$$\nabla_{w_j^{(1)}} J(w) = \frac{\partial J}{\partial a_{nj}^{(1)}} (\nabla w_j^{(1)} a_{nj}^{(1)}) + 2\lambda w_j^{(1)} \tag{12}$$

$$= \delta_{nj}^{(1)} x^{(n)} \tag{13}$$

Introducing a new variable, $\delta_{nj}^{(1)}$,

$$\delta_{nj}^{(1)} = \sum_{k=1}^{K} \delta_{nk}^{(2)} w_{kj}^{(2)} g'(a_{nj}^{(1)}) \tag{14}$$

1.1.1 Stochastic Gradient Descent

To actually optimal weights we will utilized stochastic gradient descend which updates the weights with each new data point,

$$w^{(t+1)} = w^{(t)} + \eta_t \nabla_w J(w^{(t)}) \tag{15}$$

Where η is the learning rate

1.2 Implementation

We will use the back prop algorithm to implement the training of our neural network. The back propagation algorithm is the classic method for training multilayer neural nets. It was introduced in the late 1960s, and was adapted for neural nets in the early 1990s. The algorithm consists of two main phases. the *propagation* phase, and the *weight update* phase.

During the propagation phase, data is fed to the input of the ANN and the activations are propagated forward. After this, the error is propagated backward from the output of the ANN, producing a δ at each neuron which indicates the computed necessary change at each neuron.

In the weight update phase, each neuron updates its weights according to the error gradient at the node.

This is repeated for each data point and corresponding label. After the entire training set has been used in the back propagation algorithm, the neural net is trained for future predictions.

2 Results

We tested out neural net implementation on a few different datasets. We started by testing with two toy dataset provided by the staff, and move on to the highly popular real MNIST dataset.

2.1 Toy Data Set

The toy datasets contains three different classes. They were used mostly to work out the kinks in our ANN implementation, before moving on to the more field-standard MNIST dataset.

The first toy dataset is well separated between the classes, so it will not take too much fine tuning to be able to get good results from our ANN. The second dataset has more overlap between the classes, so the ANN will need to be more carefully optimized.

We ran our ANN implementation on the toy data set using a large range of hidden nodes and regularization constants, and cross validated to find the optimal number, using both batch and stochastic gradient descent. Below is a table displaying the error rate on the first validation set across a variety of number of hidden nodes and regularization constants.

λ vs. m	10^{-10}	10^{-5}	10^{-3}	10^{-2}	10^{-1}
2	99.0	99.0	99.0	91.7	68.0
3	99.3	99.3	99.3	99.0	67.7
4	99.3	99.3	99.3	99.0	67.7
5	99.3	99.3	99.3	99.0	67.7

Here is how our ANN implementation performed on the second dataset.

λ vs. m	10^{-10}	10^{-5}	10^{-3}	10^{-2}	10^{-1}
2	89.0	88.4	88.0	66.8	33.3
3	92.7	93.0	93.0	93.0	33.3
4	92.7	93.0	93.0	93.3	33.3
5	92.3	93.0	92.7	92.7	33.3

As expected, the first dataset is separated well, while the second dataset has some persistent errors. with a proper implementation it is fairly easy to score very highly on the first toy dataset, as it is designed to be mostly separable. For the best choice of number of hidden nodes and gradient descent algorithm, we were able to achieve a correct percentage of 99.3% for toy dataset 1, and 93.3% for toy dataset 2.

2.2 MNIST

The MNIST dataset is a very common dataset in the field of machine learning. It consists of 60,000 training examples, and 10,000 test examples. The inputs are the grayscale pixel values of a 28x28 image of handwritten digits between 0 and 9. The output of the dataset is a vector of length 10, containing nine zeros and one one, indicating the actual number that was written.

This dataset is much more complex than the toy dataset, but still manageable for an ANN. After testing a many values for the variety of different free parameters, we were able to achieve a best correct percentage of //TODO: fill in the best value. Below is a table with other values for the free parameters and the corresponding scores they produce.

//TODO: insert the table!!

The best performing parameters intuitively are blah blah blah...