Implementing Multilayer Perceptrons (MLP)

2017 - CS/DS 866 Machine Learning II

Nigel Fernandez* IMT2013027
Apoorv Singh† IMT2013006
Bhamidi Satwik‡ IMT2013010

IIIT-Bangalore

March 12, 2017

1 Goal Statement

We attempted to solve the problem of recognizing handwritten digits by framing a classification problem on the MNIST database of handwritten digits [1]. We implemented multilayer perceptrons (MLP) in Python to solve the classification problem. The main goal was the implementation of MLP without use of high level libraries like TensorFlow, Keras among others.

2 Backpropogation

To supplement our understanding of MLP from class we read the first two chapters of 'Neural Networks and Deep Learning' [2] which explains neural networks from ground up starting from different types of neurons, the architecture of neural networks, the feedforward equation and most importantly the learning algorithm of the neural network, i.e., the backpropogation algorithm. We used stochastic gradient descent as part of the learning algorithm.

Contradictory to the name, MLP are composed of sigmoid neurons or other artificial neurons based on the activation function and not perceptrons. Architecture wise they are the same as a feedforward multilayer neural network, i.e., a neural network with more than one hidden layer. The principal component

*NigelSteven.Fernandez@iiitb.org
†ApoorvVikram.Singh@iiitb.org
‡KameshwaraSatwik.Bhamidi@iiitb.org

of the neural network algorithm is its learning algorithm or the backpropogation algorithm coupled with gradient descent. The four equations defining backpropogation are as follows:

$$\begin{aligned}
\delta^L &= \nabla_a C \odot \sigma'(z^L) \rightarrow ① \\
\delta^l &= ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \rightarrow ② \\
\frac{\partial C}{\partial b^l_j} &= \delta^l_j \rightarrow ③ \\
\frac{\partial C}{\partial w^l_{jk}} &= a^{l-1}_k \delta^l_j \rightarrow ④
\end{aligned}$$

The first equation calculates the output error at the output layer, the second equation propagates the error from the output layer to each of the previous hidden layers one layer at a time, the third equation calculates the gradient of the cost function with respect to the biases and the fourth equation calculates the gradient of the cost function with respect to the weights. Equation 3 and 4 are computed from equation 2 as shown in the algorithm of MLP in section 3. The weights and biases are updated by going against the gradient to minimize the cost.

In the four equations, δ^l_j refers to the error in the j^{th} neuron of the l^{th} layer, L refers to the output layer, σ is the sigmoid function, z^l_j is the weighted input to the j^{th} neuron of the l^{th} layer added with its bias b^l_j, a^l_j is the activation from the j^{th} neuron of the l^{th} layer resulted by applying the activation function (ex: sigmoid) to z^l_j, C is the cost function computed as $C(x) = \frac{|y(x)-a(x)|^2}{2}$ where $y(x)$ is the correct label and $a(x)$ is the predicted label or the activation from the output layer, $\nabla_a C$ represents the gradient of the cost function with respect to the activations in a layer.

3 Algorithm

The broad algorithm of MLP as implemented by us is shown in 1.

4 Data Set

The MNIST data set consists of three lists for training, validation and test data. The number of data points are 50000, 10000 and 10000 respectively.

Algorithm 1 Multilayer Perceptron

procedure MLP(training_data, test_data, epochs, mini_batch_size, η)
  for epoch in epochs **do**
    for mini_batch in mini_batches **do**
      for input x and output y in mini batch **do**
        Compute the feedforward values for each layer $l = 2, 3, ..., L$ by $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
        Compute the output layer error $\delta^L = \nabla_a C \odot \sigma'(z^L)$
        Backpropogate the error for each layer $l = L-1, L-2, ..., 2$ using $\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l)$
      As part of stochastic gradient descent, for each layer $l = 2, 3, ..., L$ update the weights using $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ and biases using $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$ where m is the mini batch size

We wrote a Python script `data_loader.py` to load and reshape the input data for convenient usage with MLP. The loaded data returns a tuple of three lists for training, validation and test data. Each list contains tuples of two elements where the first element is a numpy ndarray of 784 (28X28 pixels) rows and 1 column (a column vector) and the second element is a one hot vector representing the label (0-9) (1 in the index corresponding to the label and 0 elsewhere). The second element has dimensions of 10 rows and 1 column and is also a column vector.

5 Implementation and Results

We implemented MLP in Python using only the external library `NumPy` for vectorized calculations. We instantiated an MLP of architecture [784, 25, 25, 25, 10] specifying an input layer of 784 neurons, one for each input pixel, three hidden layers of 25 neurons each and an output layer of 10 neurons, one neuron for each numeric digit from 0-9. We chose ReLU as the activation function to avoid the problem of vanishing gradients with sigmoid. This MLP run over 100 epochs provided us with an accuracy of 87.36%.

We read about parameter initializing and tuning of neural networks from [3] and [4]. Following some basic principles we initialized our weight matrix with small random numbers of mean 0 and variance $2/n$ with n being the number of training data point samples. We experimented with various variance values with a variance of 1 providing an accuracy of around 65% while small random numbers as variances increasing the accuracy to around 95% over 5 epochs. We

also read that a single hidden layer being sufficient to compute any function that contains a continuous mapping from one finite space to another which we assume our classification to be. We therefore reduced the number of hidden layers to one. For the number of neurons in the hidden layer we read about the following rule of thumb which gives an upper bound for the number of hidden neurons:

$$N_h = \frac{N_s}{\alpha \cdot N_i + N_o}$$

where N_h is the number of hidden neurons, N_s is the number of training data points, N_i is the number of input neurons, N_o is the number of output neurons and α is an arbitrary scaling factor from 2 to 10. In our model, these variables take the values $N_s = 50000$, $N_i = 784$, $N_o = 10$ and $\alpha = 2$ to give $N_h = 31$ neurons.

Using the parameter initializing information above, we instantiated an MLP of architecture [784, 31, 10] specifying an input layer of 784 neurons, one for each input pixel, one hidden layers of 31 neurons and an output layer of 10 neurons, one neuron for each numeric digit from 0-9. Since the neural network had one hidden layer and the problem of vanishing gradient will not occur, we experimented with the choice of sigmoid function as the activation function . This MLP run over 100 epochs provided us with an accuracy of 96.27%.

Bibliography

[1] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST Database of Handwritten Digits*. URL: http://yann.lecun.com/exdb/mnist/.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[3] Xiu-Shen Wei. *Must Know Tips/Tricks in Deep Neural Networks*. URL: http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html.

[4] Di Jeff Heaton. *Introduction to Neural Networks with Java*. Heaton Research, Inc, 2008.