Implement Perceptron (Single and Multi-layer) Training Algorithm to Learn 8-input Boolean Function with Pocket Algoritm

Introduction

In this section, we implemented both perceptron training algorithm and multilayer perceptron training algorithms to learn linearly-separable (i.e. “AND” and “OR”) and non-linearly-separable (i.e. “XOR”) 8-input Boolean functions, respectively, and the “Pocket Algorithm” to improve the performance of perceptron algorithm, tested under environments with different levels of random noise.

Implementation and Experiment Environment

Implementation tool: Python 3.6, Numpy, Pandas, Scipy

Operating System: Windows 10

Training Set Design

In order to train a correct model for 8-input Boolean functions, basically the training set should contain at least 28 = 256 samples, each as a 9-dimensional Boolean vector, from (0, 0, 0, 0, 0, 0, 0, 0, L1) to (1, 1, 1, 1, 1, 1, 1, 1, L256), where Li is the label of the sample whose value is based on the result of corresponding Boolean function.

The 256-sample training set can be directly used to train linearly-separable Boolean functions as the perceptron algorithm will converge as long as all samples are correctly learned. However, we need to add additional samples to train the non-linearly separable “XOR” function because of the imbalancy of the training set ( only 2 (i.e. 00000000 and 11111111) out of 256 samples are labelled as “0” which have very little impact on the weight update through back propagation). Hence, in order to balance the training set, we added 50 samples same with each of the two minority samples.

The Implementation of Perceptron Training Algorithm

First of all, we randomly initialize the weight vector, then do the iteration untill it is converged, i.e. all samples have been correctly learned, or exceeded the limit of iteration times. In the iteration, we traverse all samples and check if its sign is same with the label’s. If not then update the weight. The detailed algorithm is below:

Input: X, y

Output: weight vector trained

X: training samples, y: labels

weight = np.random.rand(1, X.shape[1]+1)

converged = False

n = 0

while converged == False:

if n == n\_iter:

return

converged = True

for i in range(X.shape[0]):

if sign(y[i]) \* sign(predict(X[i])) <= 0:

weight[0] += learning\_rate \* sign(y[i])

weight[1:] += learning\_rate \* sign(y[i]) \* X[i]

converged = False

n += 1

The Implementation of “Pocket Algorithm”

The main idea of “Pocket Algorithm” is to find out the weight that can result in the most correctly learned samples in the training dataset, until all the samples are correctly learned or the iteration is finished. The pseudo code is below:

Input: X, y

Output: weight vector trained

X: training samples, y: labels

Randomly initialize Weightpocket and Weightpercep

Rpk, Rpe, Pkt, Ppe = 0, 0, 0, 0

For i in range(n\_iter):

r = randomly chosen sample index in X

If sign(y[r]) \* sign(predict(X[r])) > 0:

Rpe += 1

If Rpe > Rpk:

Ppe = number of correctly learned samples in X

If Ppe > Pkt:

Weightpocket = Weightpercep

Rpk = Rpe

Pkt > Ppe

If Pkt == |y|:

Return

Else:

Update weight

Rpe = 0

The Implementation of Multilayer Perceptron Training Algorithm

@Will

Experiment Performance of the “Pocket Algorithm”

Method of the Experiment

The aim of the experiment is to compare the performance of perceptron training algorithm using “Pocket Algorithm” and without using this algorithm, under the environment with different level of noise randomly added. Here is the method of our experiment:

1. The set of noise levels (max: 1.0): 0 (i.e. no noise), 0.1, 0.3, 0.5
2. Under each of the noise levels, test the learning of 8-input “AND” function 10 times for each perceptron training algorithms (with and without “Pocket Algorithm”).
3. Calculate average time costs of training process and average error rates.
4. Compare the performance based on the results in (3)

Experiment Result

|  |  |  |  |
| --- | --- | --- | --- |
| Noise Level | Algorithm | Average Time Cost (4 decimals) | Average Error Rate (4 decimals) |
| 0 | Pocket | 7.3631 | 0 |
| No Pocket | 0.0411 | 0 |
| 0.1 | Pocket | 7.1424 | 0.0355 |
| No Pocket | 18.2870 | 0.3684 |
| 0.3 | Pocket | 8.1633 | 0.0461 |
| No Pocket | 21.6345 | 0.4008 |
| 0.5 | Pocket | 11.3095 | 0.0434 |
| No Pocket | 25.1008 | 0.3770 |

Now we visualize the result:

Conclusion

From figures above, it can be easily concluded that the “Pocket Algorithm” has significantly better performance than the normal perceptron training algorithm when the training dataset has some level of noise, though the normal algorithm can have better performance when the dataset has no noise. However, for real-world dataset, noise data is almost inevitable, thus generally, using “Pocket Algorithm” can significantly improve the performance of perceptron training algorithm.