Performace of Back Propagation on MNIST Dataset

Name Chengyang Wu
StudentID 515030910415
F1503017

Abstract

In this report, we briefly review the theories and principles of the back propagation algorithm, including the concept of neurons and activations as well as mathematical derivations of equations and formulas. A simple implementation of this algorithm in neural networks is proposed and corresponding analysis is conducted on a few hyperparameters. Based on the above, we draw some basic conclusions on the performance of the back propagation algorithm and the neural network.

1. Introduction

Neural networks are commonplace nowadays. Various definitions exist and the most widely accepted one is *Neural networks are concurrently connected networks composed of adaptive simple units. The organization is able to simulate biological neural systems and the interactive response to real world objects.* In the aspect of machine learning, we focus on the application of neural networks in classifying data.

The basic components of neural networks are neurons. In a biological neural network, neurons are connected to each other. When a neuron is activated, it will send signals to adjacent neurons. Each neuron has an activation threshold, as long as the input signal of a neuron exceeds its shreshold, the neuron is activated.

2. Method Review

The model of a neuron is shown as Figure 1. The neuron accepts m signals from other neurons and the inputs are conveyed with an independent weight for each. In most cases there is also a bias to adjust the signal. The neuron compares the value of the combination with its threshold and generates the output through the activation function.

$$v(k) = \sum_{i=1}^{m} x_i * w_{ki} + b_k$$

$$y(k) = \phi(v(k))$$

The ideal activation function is called a hard limiter, which

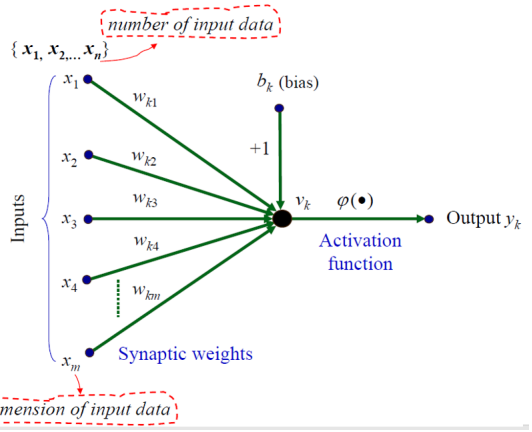

Figure 1. model of a neuron

is shown is Figure 2. In this figure, *1* corresponds to activation, and *0* stands for the contrary. Nevertheless, the hard limiter is impractical due to lack of contunuity. In practice,

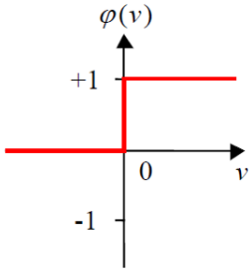

Figure 2. hard limiter

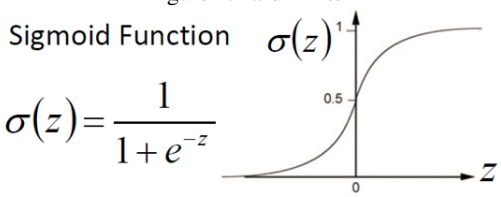

Figure 3. sigmoid function

a more applicable choice is the sigmoid function, squashing

the input into a narrow range between 0 and 1. The derivative of a sigmoid function is

$$\phi(v) = \frac{1}{1+e^{-v}}$$

$$\phi'(v) = \frac{e^{-v}}{(1+e^{-v})^2} = (1-\phi(v)) * \phi(v)$$

The connection of many neurons from certain layers and structures, which we call neural networks. A simplest kind of neural network is called a perceptron, consisting of only two layers. With only one layer of functional neurons, it can only be utilized to solve linear seperable questions. Otherwise, if the data cannot be separated by a linear hyperplane, the perceptron will run into fluctuation and can hardly generate a stable solution. In order to make up for the shortcomings, multi layers of functional neurons are adopted. The layers in the middle are called hidden layers. Figure 4 illustrates a simple example.

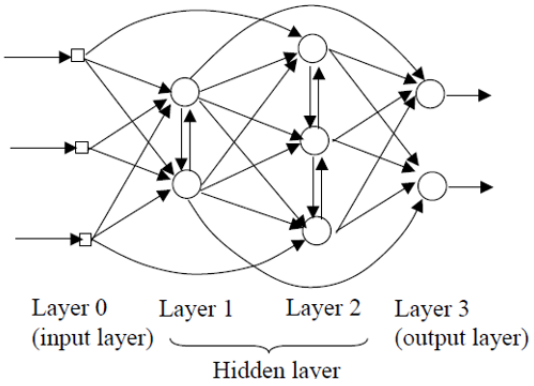

Figure 4. hidden layer

A more commonplace structure is called multi-layer feedforward neural network. As can be seen from Figure 5, every neuron in a layer is connected to all the neurons of the next layer, and no cross-layer or intra-layer connection exists. The input layer receives data from outside and the hidden layers process the signals together with the output layer. The learning process of a neural network is actuall the adjustment of connection weight between neurons according to training data and the thresholds. In other words, the learnt information are contained in the weights and thresholds.

The back propagation algorithm best stands for the rules of a multilayer neural network, and it is the most successful one until now. Moreover, the algorithm can not only be applied to feedforward networks, but can also be used to train recurrent neural networks as well.

Denote by $\alpha_h = \sum_{i=1}^{d} v_{ih} * x_i$ the input signal to neuron *h* on the hidden layer and $\beta_j = \sum_{h=1}^{q} w_{hj} * b_h$ the input signal to neuron *j* on the output layer where b_h stands for the output of neuron *h* on the hidden layer. Back propagation

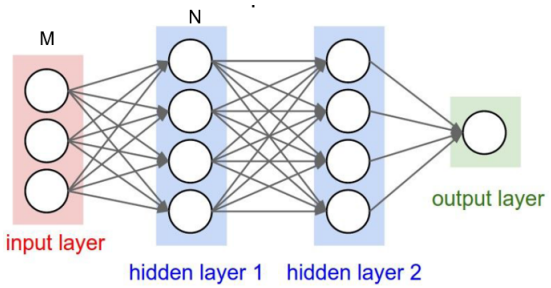

Figure 5. mult-layer feedforward neural networks

aims to compute the derivatives for each network parameter. Suppose a training sample is expressed as $\{\mathbf{x}_k, \mathbf{y}_k\}$ and the output of the neural network is $\widehat{\mathbf{y}}_k = (\widehat{y}_1^k, \widehat{y}_2^k, ..., \widehat{y}_l^k)$. Thus the square root error of the network on $(\mathbf{x}_k, \mathbf{y}_k)$ is

$$E_k = \frac{1}{2} * \sum_{j=1}^{l} (\widehat{y}_j^k - y_j^k)^2$$

We have to determine the weights from the input layer to the hidden layer, the weights from the hidden layer to the output layer, the threshold of the hidden layer and the threshold for the output layer. Back propagation is a recurrent algorithm which estimates the update of parameters according to a wide-sense perceptron rule

$$v \leftarrow v + \Delta v$$

The back propagation is based on gradient descent. For a given learning rate η, we have

$$\Delta w_{hj} = -\eta * \frac{\partial E_k}{\partial w_{hj}}$$

. Notice that the derivatives here follow a chain rule

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \widehat{y}_j^k} * \frac{\partial \widehat{y}_j^k}{\partial \beta_j} * \frac{\partial \beta_j}{\partial w_{hj}}$$

The sigmoid function follows a special property

$$f'(x) = f(x) * (1 - f(x))$$

thus we have

$$\begin{aligned} g_j &= -\frac{\partial E_k}{\partial \widehat{y}_j^k} * \frac{\partial \widehat{y}_j^k}{\partial \beta_j} \\ &= -(\widehat{y}_j^k - y_j^k) * f'(\beta_j - \theta_j) \\ &= \widehat{y}_j^k * (1 - \widehat{y}_j^k) * (y_j^k - \widehat{y}_j^k). \end{aligned}$$

then the update of w_{hj} follows

$$\Delta w_{hj} = \eta g_j b_h$$

similarly

$$\Delta\theta_j = -\eta g_j$$
$$\Delta v_{ih} = \eta e_h x_i$$
$$\Delta\gamma_h = -\eta e_h$$

where

$$\begin{aligned} e_h &= -\frac{\partial E_k}{\partial b_h} * \frac{\partial b_h}{\partial \alpha_h} \\ &= -\sum_{j=1}^{l} \frac{\partial E_k}{\partial \beta_j} * \frac{\partial \beta_j}{\partial b_h} * f'(\alpha_h - \gamma_h) \\ &= \sum_{j=1}^{l} w_{hj} g_j f'(\alpha_h - \gamma_h) \\ &= b_h * (1 - b_h) * \sum_{j=1}^{l} w_{hj} g_j. \end{aligned}$$

The learning rate $\eta \in (0,1)$ controls the update step during every iteration. A η too large leads to fluctuations, yet one too small slows down the convergence. Sometimes we use different η for different layers so as to achieve precision control.

For every train sample, back propagation algorithm processes it as follows: the input signal is provided to the input layer and then fed forward layer by layer until the output layers generates the results. Then the error is calculated and fed back reversely to the hidden layers. Finally, we adjust the connections weights and threshold values according to the error of every hidden layer. The process cycles on until the restrictions are met, like the error is small enough.

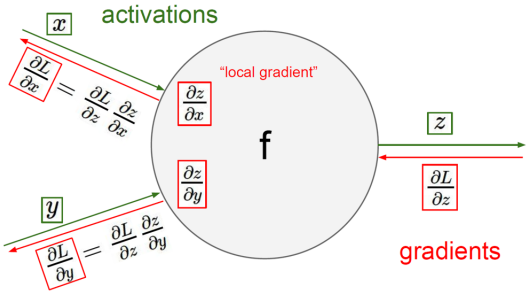

Figure 6. back propagation

The standard back propagation updates only respect to one sample at a time. If we update with respect to a number of samples, we can update less and save more time. Also, accumulating the error avoids the inter-neutralization of independent errors. However, a major short coming for accumulated error backpropagation is the difficulty of further decrease after reaching some particular extent. Under this circumstance, the standard BP actually performs better.

Another concern for this algorithm is overfitting. In practice, two methods are adopted to avoid the problem. One is early stopping, trying to estimate the error with cross-validation set. Once the error for the cross-validation set increases, training is ceased. Another is regularization, aiming to include the complexity of the network into the cost function.

$$E = \frac{\lambda}{m} * \sum_{k=1}^{m} E_k + (1 - \lambda) * \sum_i w_i^2$$

where $\lambda \in (0,1)$ is utilized to tradeoff cost for complexity.

Notice that gradient descent may encounter a local minima instead of a global minima. One way to fix this is multiple initialization points. Apart from this, various other methods can also be utilized to boost neural network training:

Data preprocessing. Change the input to zero mean and unit variance.

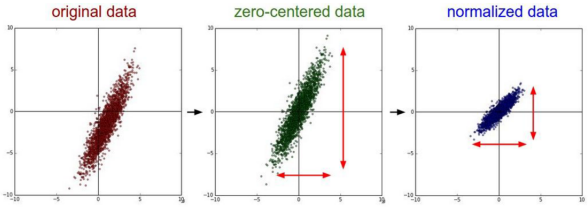

Figure 7. data preprocessing

Select proper lost function. Cross entropy is better than square error.

Weight initialization. Initialize as small random numbers rather than all zeros.

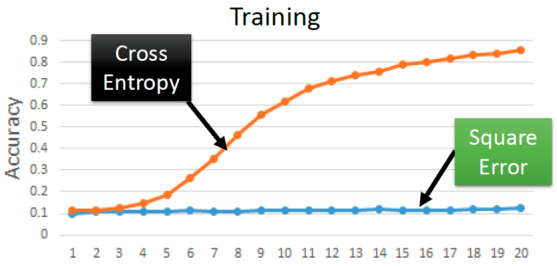

Figure 8. cross entropy vs square error

Avoid vanishing gradient. ReLU and tanh perform better than sigmoid.

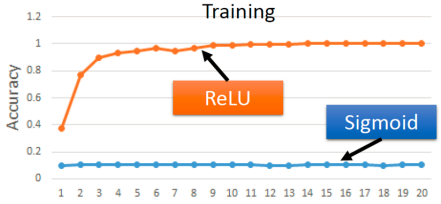

Figure 9. relu vs sigmoid

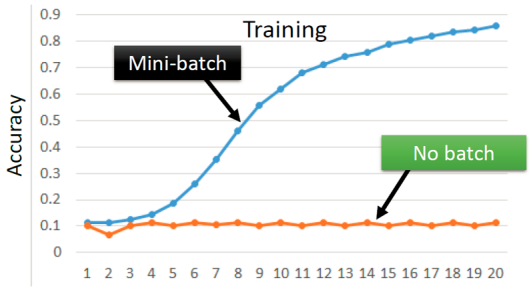

Figure 10. mini batch vs no batch

Use mini batch. Updating once every batch helps converge faster.

Adjust the learning rate. It should be large in the beginning but slowly decreasing every epoch. Possible methods include weight decay and momentum.

Dropout. Randomly disable neurons while training and reuse them while testing. Thus different structures would ensemble to yield the best results.

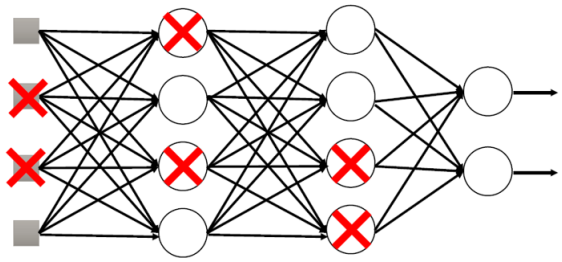

Figure 11. dropout

3. Comparison and Discussion

My implementation of neural network on the MNIST dataset utilizes sigmoid function. During each epoch of the stochastic gradient descent process, a random batch is fetched for the network to perform update using back propagation. The learning rate is adjusted corresponding to the cost function so that a smaller error yields a smaller stepsize. Although the implementation supports **arbitrary number of hidden layers**, my experimental results so far suggest that **one single hidden layer** is enough for training a network with **an accuracy of over 96% on the test dataset** in **no more than 65 seconds** provided that the parameters are chosen carefully.

To be brief, I put forward here the effect of the following parameters of the performance of the network:

epoch

The epoch stands for the number of batches the network fetches to update. The results in following table are generated with batchsize=500 and one hidden layer containing 80 neurons.

Table 1. effect of epoch

epoch	100	500	1000	5000
time/s	13.66	69.36	148.9	735.39
accuracy/%	92.4	96.26	96.91	97.14

The accuracy is close to saturation after 500 epoches. Continuous training may still lift it a little, but not too much. It is not worth consuming ten more times of complexity.

batchsize

The batchsize is the number of sample the network fetches every epoch. The following results are obtained under the parameters epoch=500 and one hidden layer with 80 neurons.

Table 2. effect of batchsize

batchsize	100	200	500
time/s	13.82	27.38	69.36
accuracy/%	92.87	94.5	96.26

It is obvious that with the increase of the batchsize the accuracy also increases. However, the saturations problem happens as before. Raising the batchsize to 1000 or even higher not only adds to the training time, but may cause the error functions to neutralize each other as well.

number of neurons

As mentioned before, the best results are obtained with only one hidden layer, so we focus our attention on choosing the appropriate number of neurons in this hidden layer.

The following results are obtained with epoch=500 and batchsize=500.

Table 3. effect of number of neurons

neurons	60	80	100
time/s	64.44	69.36	80.94
accuracy/%	96.26	96.26	96.02

A quick observation reveals the surprising fact that a more complex network may sometimes yield a worse result. 60 neurons may sometimes prove to be more effective than 100 neurons. Nevertheless, the randomness of stochastic gradient descent implies that the results may not be reliable, so we cannot quickly jump to the conclusion that 60 neurons would be the best. For stability, I actaully set 80 as the default value.

4. Conclusion

The principles and realizations of the neural network gave me new insight into machine learning. Unlike other supervised learning methods which have strong theoretical support, neural networks seem to be more mysterious as for

the way they work. However, the sharp contrast of the overall performance deeply impressed me. Faster than almost all other approaches I have come into before, neural networks can finish training in no more than 15 seconds and still convey an accuracy above 90%, and it is not surprising to estimate that, given an hour or two, like the other methods, neural networks can achieve near-perfect performance. Regardless of these, the shocking efficiency in training and the effectiveness in testing already make it a first choice under many circumstances. Moreover, the model is merely in its prototype form without the assistance of cross entropy, rectified linear units and softmax function, not to mention convolutional and recurrent techniques. This greatly arouses my interest. The fact is that, by imitating the systems of animals, we get far better performance metrics than those delicate manual structrues. This reminds me that there is still a long way to go and the neural networks are just a small step forward on the arduous journey.

Above all, the process of exploring and modifying benefits me to a great extent. Although this is the last project this term, it has left me an unforgettable memory. Hope I can encounter more challenging and interesting tasks in the future to put my knowledge and skills into practice.

References

[1]Z.-H.Zhou.Machine Learning.Beijing.Tsinghua Univ. Press.2016.ISBN 978-7-302-42328-7