A Practical Analysis of the Convergence of Back Propagation

Andrei Purcarus, *McGill University* and Sean Stappas, *McGill University*

***Abstract*—A vectorized version of the back propagation algorithm for fully connected artificial neural networks was implemented in order to provide a practical analysis of its convergence. Several improvements, such as input normalization, the use of the hyperbolic tangent as an activation function, and the use of a larger batch size in stochastic gradient descent, were found to be effective in reducing training time. Other improvements, such as reducing the learning rate for higher layers in the network, were found to be ineffective for shallow networks. Several improvements were also suggested to allow for the study of the convergence of back propagation for deeper networks.**

I. Introduction

THIS report provides a practical analysis of the convergence of the back propagation algorithm for a fully connected neural network. To provide this analysis, we first implemented a vectorized version of the back propagation algorithm for a fully connected network of arbitrary size. Then, we applied a series of "tricks of the trade" to try to speed up the convergence of the algorithm. Finally, we compared our best results to those of a deep convolutional network and of a random baseline.

II. Implementation of a Fully Connected Network

We started by implementing a fully connected neural network in Python. Our starting point was the traditional back propagation algorithm given in AI textbooks [1]. However, in order to obtain good performance out of the network and avoid the interpretation overhead of Python, we decided to vectorize the algorithm and take advantage of the efficient matrix multiplication implementation provided by NumPy.

Assuming we have a network with k layers, we label the bias weight vectors as b_j and the weight matrices as W_j, for $j = 1, \cdots, k-1$. In addition, we label our activation function as f, the input vector as x, and the expected output vector as y. The algorithm then proceeds as follows:

1) Forward propagate the input x through the network, saving the intermediate values in_j and out_j for every layer.
2) Compute the matrix Δ using
$$\Delta_{k-1} = f'(in_k) \cdot (y - out_k) \tag{1}$$
3) Back propagate the matrix Δ through the network using
$$\Delta_j = f'(in_{j+1}) \cdot W_{j+1}^T \Delta_{j+1},\ j = 1, \cdots, k-2 \tag{2}$$
4) Compute the gradient of the mean squared error function with regularization using
$$\frac{\partial E}{\partial b_j} = -\Delta_j + \lambda b_j,\ j = 1, \cdots, k-1 \tag{3}$$
$$\frac{\partial E}{\partial W_j} = -\Delta_j out_j^T + \lambda W_j,\ j = 1, \cdots, k-1 \tag{4}$$
where λ is a regularization parameter that penalizes large weights.

The gradient can then be used to perform gradient descent and train the network.

III. MNIST Data Set

We next had to select a data set to perform our experiments on. We chose the widely used MNIST data set, which consists of a set of $60,000$ hand written digits that serve as training examples and $10,000$ hand written digits that serve as test examples. A sample of this data set is shown in Figure 1. This set provided us with a large amount of data to test our neural network with.

Fig. 1. Sample digits from the MNIST data set [2].

IV. Convergence Improvements

We next moved to analyzing the convergence of back propagation when using stochastic gradient descent. The majority of the improvements in this section are drawn from LeCun's 1998 paper *Efficient BackProp* [3], which provided an analysis of several "tricks of the trade" and how they improve convergence. Unless otherwise specified, we used a single hidden layer containing 100 nodes, the hyperbolic tangent activation function, a learning rate of 0.01, a batch size of 100, and no momentum.

A. Input Normalization

To speed up convergence, every node in every layer should have an input with an average close to zero. This ensures that the weight updates during learning are not biased in any particular direction. In addition, the input to each node should have the same standard deviation in order to equalize the learning speed across nodes. To achieve this, the input normalization, activation function, and weight initialization must be carefully coordinated [3].

First, we normalize the input training data through shifting and scaling. To ensure that the mean of the input data is zero, we subtract the mean of the input vectors from each input vector, using Equation (5).

$$x_i \leftarrow x_i - \bar{x} \tag{5}$$

Once the mean is subtracted, we divide each input vector by the standard deviation σ_x of the input vectors using Equation (6). This ensures a variance of 1 over the input data.

$$x_i \leftarrow \frac{x_i}{\sigma_x} \tag{6}$$

The parameters $\bar{x}$ and σ_x must be stored and later applied to the test input data once training is complete.

B. Sigmoid

The choice of activation function is also important to ensure optimal convergence. We compare two sigmoid functions: the logistic sigmoid function and the hyperbolic tangent sigmoid function. The logistic sigmoid function is given by Equation (7).

$$f_{logistic}(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

The hyperbolic tangent sigmoid function is given by Equation (8). This function is recommended by LeCun for several reasons [3]. First, it is symmetric around the origin and therefore does not induce any bias. The output variance will also be close to 1 if the input variance is 1. Next, it has the property that $f(\pm 1) = \pm 1$ and that ± 1 are not asymptotes. In fact, these are points where the second derivative has a maximum. Therefore, the input as well as the target values of the sigmoid can be safely initialized in the full range from -1 to $+1$. This is in contrast with the logistic sigmoid, where target values of ± 1 cannot be used because they are asymptotes.

$$f_{tanh}(x) = 1.7159 \tanh\left(\frac{2}{3}x\right) \tag{8}$$

A graph of both sigmoid functions is shown in Figure 2. Here, the properties of the hyperbolic tangent sigmoid function described above can clearly be seen.

A comparison of the performance of both sigmoid functions for our network is shown in Figure 3. Note that we adjusted the learning rate for the logistic sigmoid to compensate for the fact that its error function is $(2 \cdot 1.7159)^2$ times smaller than that of the hyperbolic tangent sigmoid. We can see that using the hyperbolic tangent sigmoid leads to faster convergence compared to using the logistic sigmoid, as expected.

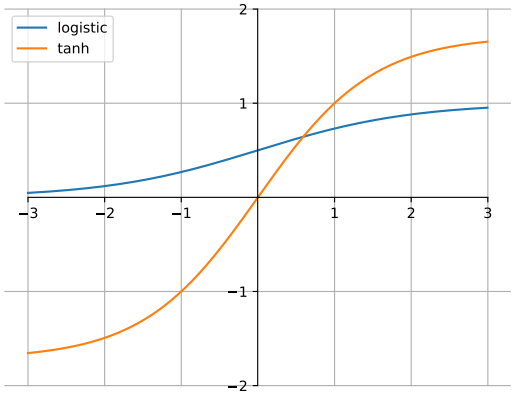

Fig. 2. Graph of the logistic sigmoid function and tanh sigmoid function.

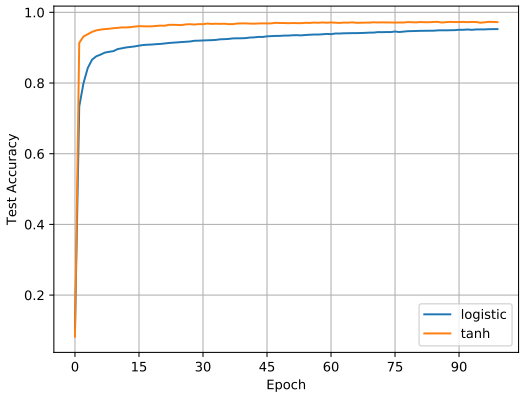

Fig. 3. Test accuracy when using the logistic sigmoid versus the tanh sigmoid over 100 epochs.

C. Weight Initialization

To ensure that the average input to each layer in the network is close to zero, the weights must also be properly initialized. Given that the input is normalized and that the hyperbolic tangent sigmoid is used, the initial weights should be drawn from a distribution with a standard deviation of σ_w as given by Equation (9), where m is the fan-in of the layer [3]. We chose to use a Gaussian distribution to select the initial weights.

$$\sigma_w = m^{-1/2} \tag{9}$$

D. Learning Rate

The weight update equation for a node i is given by Equation (10), where w_i and η_i are the weight and learning rate of node i, respectively.

$$w_i \leftarrow w_i - \eta_i \frac{\partial E}{\partial w_i} \tag{10}$$

We initially used a global learning rate η for all layers. The learning curve for various values of η is shown in Figure 4. This graph shows that a learning rate that is too high, such as 0.1, causes slow, jagged and non optimal convergence.

A closer view of the curves for better performing learning rates is shown in Figure 5. Here, we can see that the learning

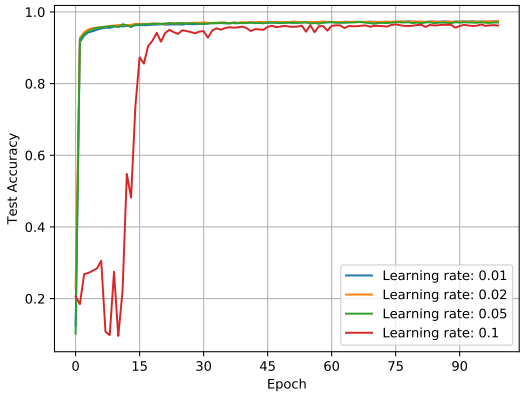

Fig. 4. Test accuracy when using various learning rates over 100 epochs.

rate of 0.02 has the best performance, since it causes quick convergence to a high test accuracy.

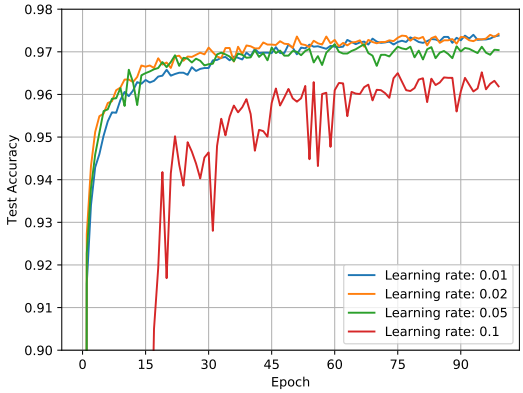

Fig. 5. Test accuracy when using various learning rates over 100 epochs. Cropped to show test accuracies in the 0.90 to 0.98 range.

E. Layer Decay

During training, the weight updates will typically be larger in the higher layers of the network. This is due to the fact that the second derivative of the cost function is generally larger in these higher layers [3]. To equalize the weight updates across all layers, the learning rate can be made a decreasing function of the layer depth, with the lower layers having a higher learning rate than the higher ones. This can be implemented with a layer decay term δ, as given by Equation (11), where η_i is the learning rate in layer i, and layer i is higher than layer $i-1$.

$$\eta_i = \delta \eta_{i-1} \tag{11}$$

The resulting performance when using various values for the layer decay δ is shown in Figure 6. We can see that the layer decay does not have a significant effect on the convergence. This is most likely due to the fact that the network has only one hidden layer. The effect would be more pronounced in much deeper networks.

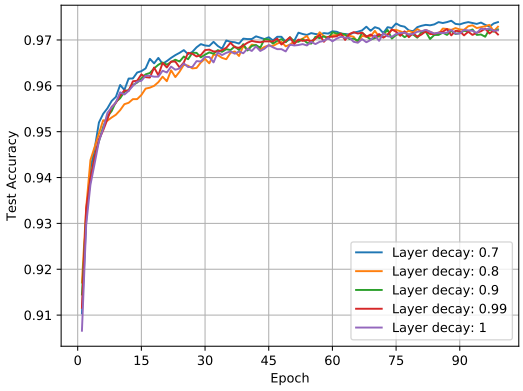

Fig. 6. Test accuracy when using various layer decay values over 100 epochs.

F. Momentum

Momentum is an added term to the weight update equation that typically helps accelerate convergence [3]. The update equation for the weight change Δw is given by Equation (12), where μ is the momentum.

$$\Delta w(t+1) \leftarrow -\eta \nabla E + \mu \Delta w(t) \tag{12}$$

The results of using various values for the momentum μ are shown in Figure 7. We can see that, with a high momentum value of 0.9, the learning curve rises quickly, but does not converge to a global optimum. An intermediate momentum value of 0.3 leads to relatively fast convergence to a high test accuracy.

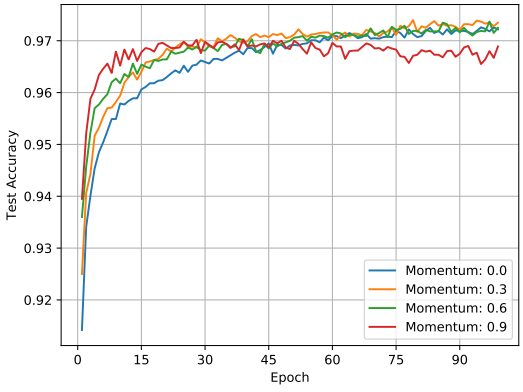

Fig. 7. Test accuracy when using various momentum values over 100 epochs.

G. Batch Size

The batch size represents the number of input examples to consider for a single weight update in stochastic gradient descent. This is sometimes referred to as a "mini-batch" size. A batch size of 1 is equivalent to online stochastic gradient descent, where each training example is used for a weight update.

The results of using various batch sizes are shown in Figure 8. We can see that a batch size of 1 leads to very jagged learning. In general, a higher batch size leads to more

accurate weight updates, but slower convergence. This can be seen in the discrepancy between the learning curves with batch sizes of 10 and 100. Here, the best batch size appears to be 100, since it causes relatively fast convergence to a high test accuracy.

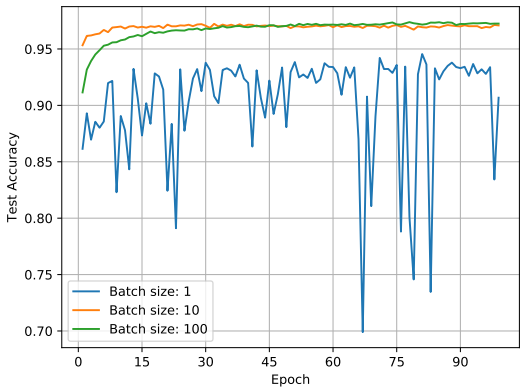

Fig. 8. Test accuracy when using various batch sizes over 100 epochs.

H. Network Size

With the network tuned to use all the previously described improvements, we tested the effect of varying the network size. The results are shown in Figure 9. The general trend is that increasing the number of hidden units improves performance. We can see, however, that increasing the number of layers does not yield any significant improvement. The best performing network arrangement is thus one with a single hidden layer with 300 hidden units.

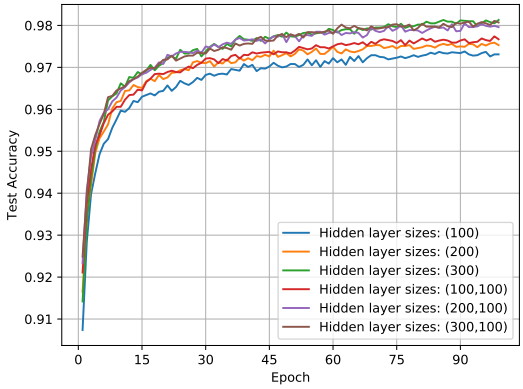

Fig. 9. Test accuracy when using various hidden layer configurations.

V. Comparison to Other Approaches

We next compared our best performing network, which contains one hidden layer with 300 units, to a deep convolutional network taken from TensorFlow [4] and a random baseline. We expected that a convolutional network would significantly outperform the fully connected network since the features learned by a convolutional layer are applicable to the entire image. Thus, the classifier exhibits some translational invariance to its input and is better suited to image recognition. This is shown clearly in Figure 10, where the convolutional network attains a test accuracy of $99.3\,\%$ compared to the fully connected test accuracy of $98.1\,\%$. Both of these approaches are still much better than the random baseline, which achieves an expected accuracy of $10\,\%$ and thus was not shown.

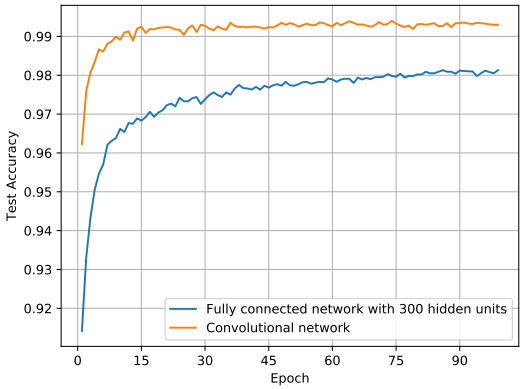

Fig. 10. Test accuracy when using a fully connected network with 300 hidden units and a deep convolutional network.

VI. Conclusions

In conclusion, we compared the effects of different tricks on the convergence and performance of the back propagation algorithm applied to a fully connected neural network. Some tricks, such as normalizing the input and using a hyperbolic tangent activation function to reduce biases in learning, and using a larger batch size to better approximate the true gradient, were shown to be effective. Others, such as using a learning rate reduction for higher layers, were found to have little effect for a shallow network. One possible expansion of this study would be to measure the effects of the different tricks for deeper networks. This would likely require the addition of the rectified linear activation function (ReLU), as well as measures to control overfitting such as the addition of dropout layers, in order to obtain reasonable training times and avoid problems like the vanishing gradient experienced by deeper networks.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[2] J. Steeves. (2015, Sept) MNIST Hand Written Digits Classification Benchmark. [Online]. Available: https://knowm.org/mnist-hand-written-digits-classification-benchmark/

[3] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 9–50, DOI: 10.1007/3-540-49430-8_2. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-49430-8_2

[4] Deep MNIST for Experts. [Online]. Available: https://www.tensorflow.org/get_started/mnist/pros