Backpropagation on MNIST

Abrar Anwar
aa76875
University of Texas at Austin
abraranwar123@gmail.com

Abstract

A simple feed forward neural network with two layers is implemented using numpy in order to classify MNIST data. Minibatch is used along with a decaying learning rate using various activation functions to get as high as around 98% accuracy. In addition, various hidden layer sizes are explored in order to find the best hidden units count so that we can optimize the speed-accuracy trade off.

1. Introduction

1.1. Backpropagation

Backpropagation optimizes the weights and biases of the layers of a neural network such that it optimizes an objective function using training data. It does so by computing the gradient of the loss function with respect to the weights. Given activation function $g(\cdot)$, for the ith layer, the input to g is z_i, $g(z_i)$, which is the output after a linear function. This linear function is the $W_i x_i + b_i$, where W_i and b_i are the weights and biases of the ith layer.

The forward propagation process of a 2 layer network is as follows:

$$\begin{aligned} z_1 &= W_1 x + b_1 \\ a_1 &= g(z_1) \\ z_2 &= W_2 a_1 + b_2 \\ a_2 &= g(z_2) \\ z_3 &= W_3 a_2 + b_3 \\ y = a_3 &= g(z_3) \end{aligned}$$

The goal of backpropagation is to optimize these as we get more training data. Given some loss function $L(\hat{y}, y)$, we want to update the weights with respect to the loss function. Let us say if we have K layers, layer K is the final one and $k \in [0, K-1]$. Backpropogation looks like this:

$$\begin{aligned} dA_K &= L'(\hat{y}, y) \\ dz_k &= dA_{k+1} g'(z_k) \\ dW_k &= x_k^T dz_k \\ dA_k &= dz_k W_k \\ db_k &= \sum dz_k \end{aligned}$$

Each partial deriviative actually represents $\frac{\partial L}{\partial *}$, where the $*$ represents each potential variable. We calculate these partial derivatives typically by taking advantage of the chain rule. It is easy to calculate $\frac{\partial L}{\partial A}$ since it's simply the error with respect to the loss function's derivative for the last layer, or the errors of the output. With this, we can start calculating the rest using chain rule.

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} \tag{1}$$

(1) shows that we can calculate the derivative using a value that's already known along with the partial derivative of the activation function, which is easily calculated since it's the sigmoid function in most cases.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial W} \tag{2}$$

(2) shows that we want to multiply by the partial derivative of z with respect to W. This easily turns into a_{i-1}, which is the input to the current layer, equivalent to the output of the previous layer. We can verify this by looking at the structure of a 2 layer network seen at the beginning of this paper. The equations above were formulated with help from Andrew Ng's Machine Learning course notes [1].

2. Method

2.1. Dataset

The data used is the MNIST digits dataset, as seen in Figure 1. It incorporates 60k data samples for the training set and 10k for the output set. Due to so many data samples, I added batch inputs to speed up the training process. Due to this, the gradients were also divided by the batch size in order to prevent large jumps in the gradient. In addition, the input data, which usually has a range between 0 and 255 was scaled down to be between 0 and 1.

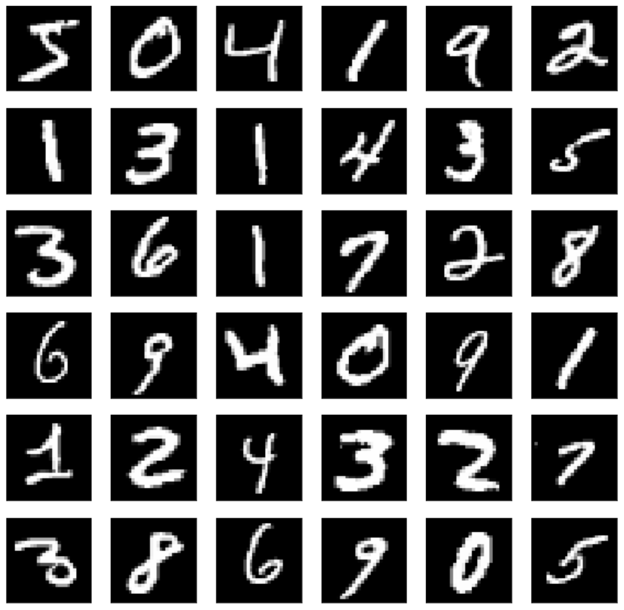


Figure 1. MNIST digits dataset

2.2. Network Architecture

I built a feedforward neural network architecture in Python using numpy that can handle a variable number of layers and easily switch between activation functions. There exist multiple hyperparameters that can be changed for a neural network, such as the learning rate and the number of hidden units. In addition, there exist a variety of configurations in regards to activation functions used.

It is common for a common network architecture for a 2 layer network to consist of first, a nonlinear activation function after the first hidden layer followed by a linear activation for the output layer. We will try various nonlinear activation functions for the activation of the hidden layers, but the final output will always follow a linear activation. In addition, the loss function used is softmax cross entropy loss, which adds a softmax activation to the final layer after calculating the score outputs of the neural network.

The weights of each layer are randomly initialized to be between normally distributed with a mean of 0 and a variance of 1. We will see this causes issues with a certain activation function.

$$L(\hat{y}, y) = -\sum y \log softmax(\hat{y})$$
$$L'(\hat{y}, y) = softmax(\hat{y}) - y$$

3. Results

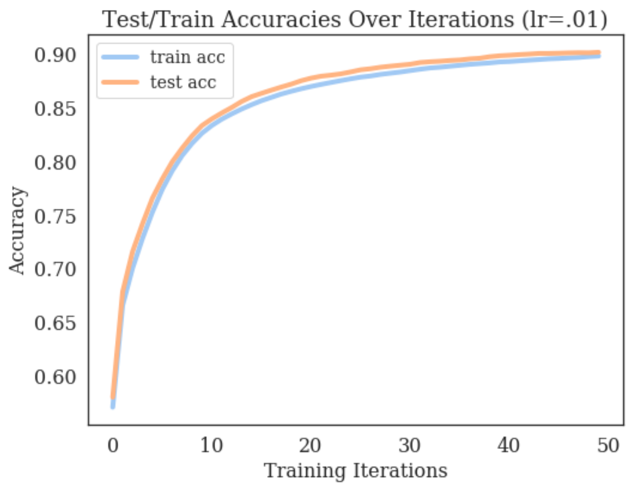


Figure 2. Test/train with batches

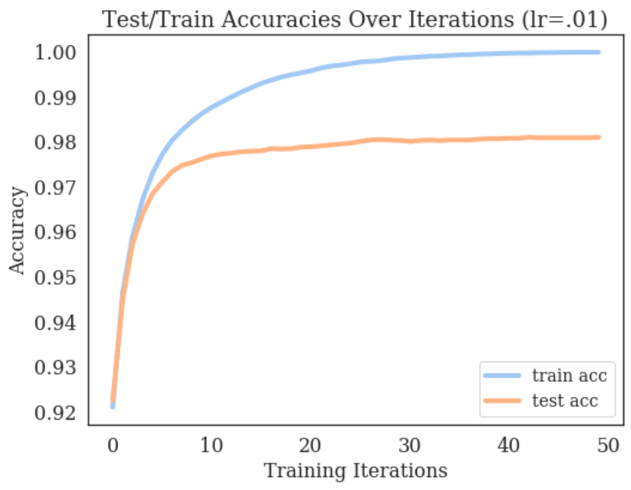


Figure 3. Test/train without minibatches

3.1. Hyperparameters

In Figure 2 and 3, we can see two trials, one running with batches and one without. We see that having no batches converges much faster than having a batch size of 250; however, later trials show that having batches are effective, but different learning rates should be used.

In regards to learning rate, we see in Figure 5 that a learning rate of 1 seems to converge faster; however, motivated by making large motions in the gradient at the beginning and refining later on, a decaying learning rate could be more effective. We can see that in comparison to the constant learning rates, a decaying learning rate appears to be just as competitive as a constant learning rate of 1. A decaying learning rate can also be slightly more robust than having a

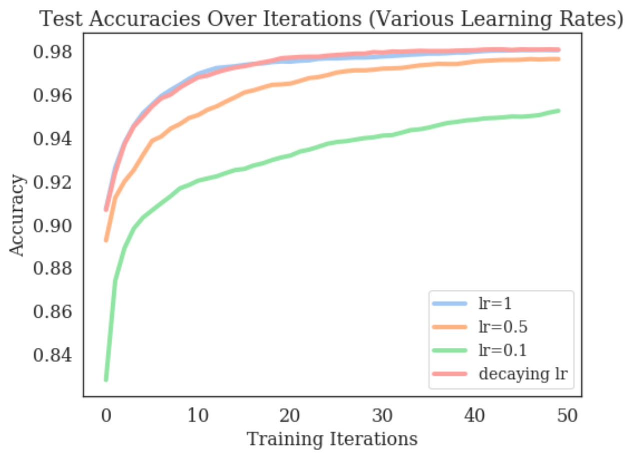


Figure 4. Various learning rates with batch size of 250, including decaying learning rate

constant learning rate of 1.0. All future trials consist of this decaying learning rate.

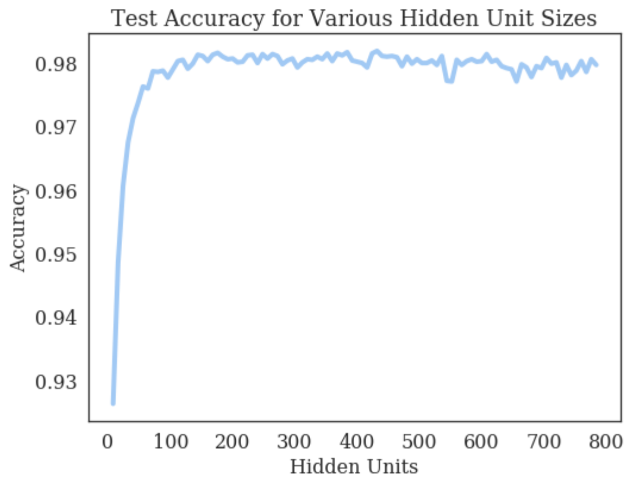


Figure 5. Accuracy for multiple hidden unit sizes for 2 layer NN using sigmoid activation for hidden layer

We can also choose the number of hidden units for the hidden layer. We can see in Figure **??** that after a hidden unit count of about 100 is approximately at the same accuracy for everything greater. When more hidden units are added, the time to run a single epoch increases significantly. Due to this, we run for all later trials, a hidden size of 100 is optimal in time and in accuracy.

3.2. Activation Functions

As said previously, various nonlinear activation functions can be used for the first layer of the neural network. We try Sigmoid, ReLU, and tanh to see which one preforms the best. The equations of these are below:

$$g_{sigmoid}(x) = \frac{1}{1+exp(-x)};\ g'_{sigmoid}(x) = g(x)(1-g(x))$$

$$g_{tanh}(x) = tanh(x);\ g'_{tanh}(x) = 1 - g(x)^2$$

$$g_{ReLU}(x) = max(0,x);\ g'_{ReLU}(x) = 0 \text{ if } x < 0; 1 \text{ otherwise}$$

In Figure 6, we can see the various activation functions being replaced for the middle layer. It is interesting to note that the tanh activation converges incredibly quickly, but both plateau to the same accuracy. ReLU shows to be much farther behind, which is odd. I believe this has to do with the random initialization of weights. These initialization of the weights work well for tanh and sigmoid, but not for ReLU.

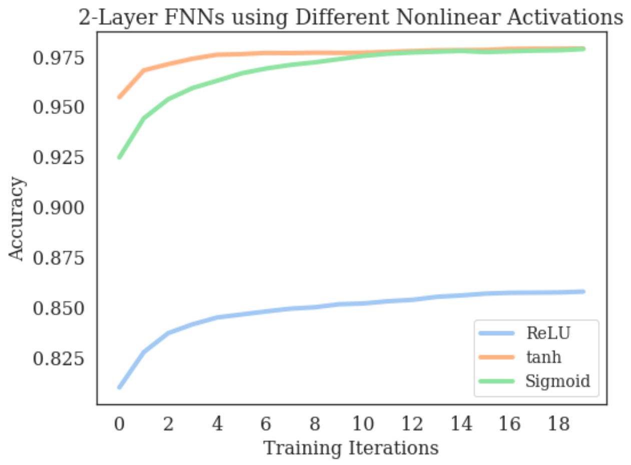
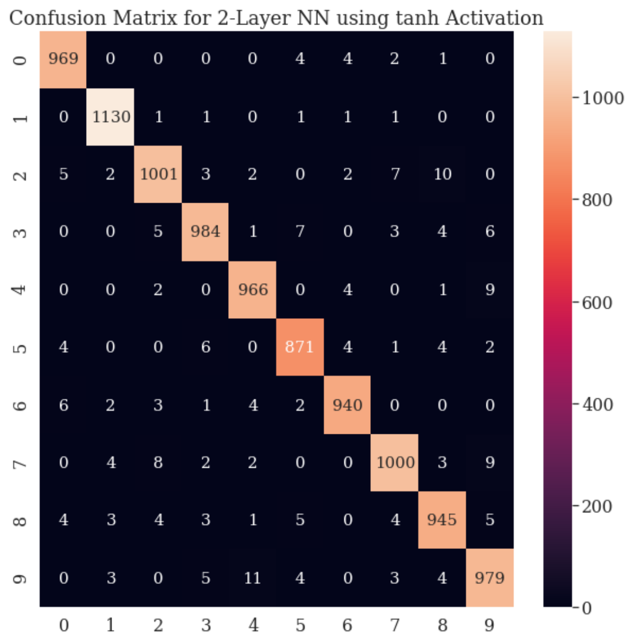


Figure 6. Trying different activation functions

Finally, since we know the tanh activation has the highest accuracy, in Figure 7, we can see the confusion matrix for the test data for this result. Accuracies are really good, however, we can see errors still exist in some parts of the network. For example there are more errors where 4 and 9 are mixed up as well as 2 and 7, and 2 and 8. These inaccuracies can be reduced through a convolutional neural network which would be able to handle local regions effectively.

4. Summary

In conclusion, this assignment on implementing backpropogation using a 2 layer neural network allowed me to explore different activation functions and learning rates, and their impact on overall accuracy. We learned that tanh is generally better than sigmoid, and that ReLU likely depends on the initialization of the weights. Different weight initializations could be tested in the future to see what impact it could have on the training process



	0	1	2	3	4	5	6	7	8	9
0	969	0	0	0	0	4	4	2	1	0
1	0	1130	1	1	0	1	1	1	0	0
2	5	2	1001	3	2	0	2	7	10	0
3	0	0	5	984	1	7	0	3	4	6
4	0	0	2	0	966	0	4	0	1	9
5	4	0	0	6	0	871	4	1	4	2
6	6	2	3	1	4	2	940	0	0	0
7	0	4	8	2	2	0	0	1000	3	9
8	4	3	4	3	1	5	0	4	945	5
9	0	3	0	5	11	4	0	3	4	979

Figure 7. Confusion matrix for 2 layer NN using tanh activation in hidden layer

References

[1] A. Ng and K. Katanforoosh. CS229 Lecture Notes, October 2018.