Digit Recognition using Backpropagation

Nisarg Chokshi
North Carolina State University
nmchoks2@ncsu.edu

***Abstract*—This is a digit recognition task and there are 10 digits (0 to 9) or 10 classes to predict from. Results are reported using prediction error, which is nothing more than the inverted classification accuracy. We make use of backpropagation to create this network. Backpropagation, short for "backward propagation of errors," is an algorithm for supervised learning of artificial neural networks using gradient descent. Given an artificial neural network and an error function, the method calculates the gradient of the error function with respect to the neural network's weights.**

I. Introduction

Researchers had long been interested in finding a way to train multilayer artificial neural networks that could automatically discover good "internal representations," i.e. features that make learning easier and more accurate. Features can be thought of as the stereotypical input to a specific node that activates that node (i.e. causes it to output a positive value near 1). Since a node's activation is dependent on its incoming weights and bias, researchers say a node has learned a feature if its weights and bias cause that node to activate when the feature is present in its input.

The term backpropagation strictly refers only to the algorithm for computing the gradient, not how the gradient is used; but the term is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent. Backpropagation generalizes the gradient computation in the Delta rule, which is the single-layer version of backpropagation, and is in turn generalized by automatic differentiation, where backpropagation is a special case of reverse accumulation (or "reverse mode").

II. Code changes

A. mlp.py

This is the caller code that is responsible for defining the hyperparameters for the neural network. Here, I have also calculated the training accuracy, validation accuracy and test accuracy. This file is also generating the raw data for the predictions of the testset.

B. activation.py

Completed the functions for sigmoid and sigmoid derivative functions which are crucial for feedforward and backpropagation calculations.

C. bp.py

Added code for feedforward and backpropagation based on the algorithm discussed in the lectures and found in the lecture slides and other textbooks.

III. Effect of Hyperparameters

A. Changing Hidden Layer size

Number of hidden nodes represent the required capacity to learn function for a complex function. The more complex the function, the more learning capacity the model will need. Slightly more number of units then optimal number is not a problem, but a much larger number will lead to the model over fitting i.e. If you provide a model with too much capacity, it might tend to over fit and just try to memorize the data set.

Here I changed the hidden layer size to check where I got the highest accuracy. A hidden layer of size 80 was the most optimal wherein the accuracy was about 91%.

B. Adding Hidden Layers

Generally bigger neural networks are needed when your computation is complicated with many minimas and are able to give better performance in terms of loss. But they require more time to train and are not needed when working with gradient descent kind of problems. With our problem, adding more layers produced spiky sort of a curve as more epochs were calculated.

C. Changing Learning Rate

If the learning rate is too small than optimal value then it would take a much longer time (hundreds or thousands) of epochs to reach the ideal state. Or, on the other hand, If our learning rate is too large than optimal value then it would overshoot the ideal state and our algorithm might not converge.

This is clearly observed when the learning rate is reduced by a factor of 10 to 0.0001 wherein it takes a lot more epochs to reach the acceptable accuracy ranges.

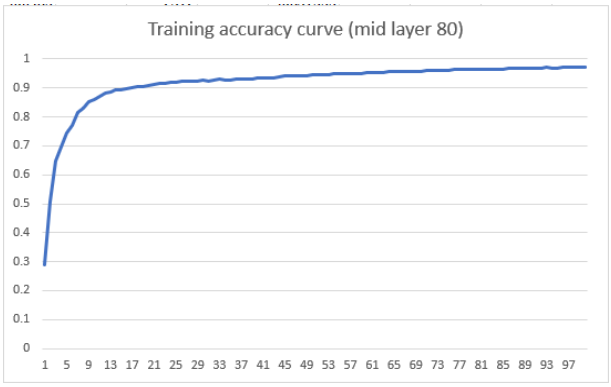

Fig. 1. Training accuracy curve for hidden layer of 80.

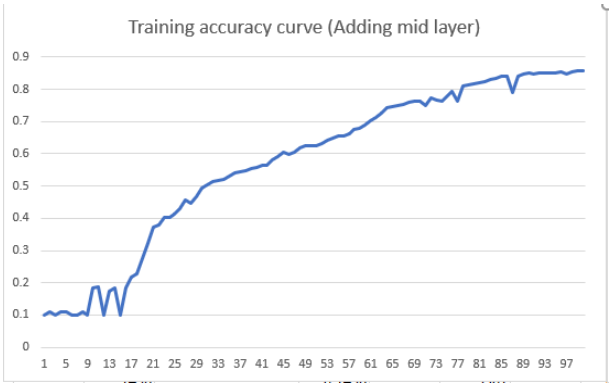

Fig. 2. Training accuracy curve for a network of 784-20-40-20-10.

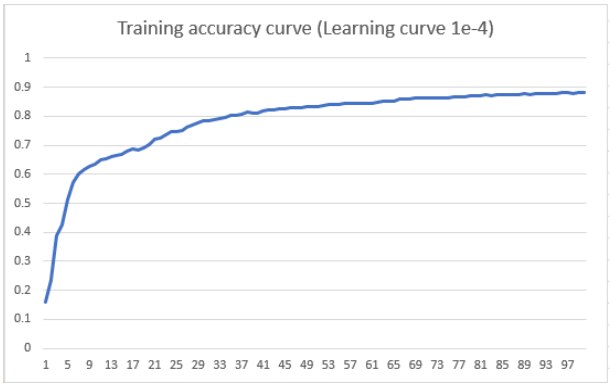

Fig. 3. Training accuracy curve for learning rate = 0.0001.

IV. Results

Relatively successful classification for Digit recognition was achieved from the dataset from the MNIST database by using a 3 layer neural network.

Network Structure	Learning Rate	Validation Acc	Test Acc
784,20,10	1.00E-02	83.02	82.21
784,20,10	1.00E-03	90.49	90.28
784,20,10	1.00E-04	84.62	82.97
784,40,10	1.00E-02	70.8	71.06
784,40,10	1.00E-03	91.12	91.14
784,40,10	1.00E-04	85.98	84.96
784,80,10	1.00E-03	91.04	90.7
784,80,10	1.00E-04	87.34	86.33
784,120,10	1.00E-02	88.53	87.65
784,120,10	1.00E-03	91.44	90.83
784,120,10	1.00E-04	86.94	86.21
784,150,10	1.00E-03	91.69	91.1
784,150,10	1.00E-04	87.33	86.31
784,180,10	1.00E-03	91.44	90.95
784,180,10	1.00E-04	87.18	86.39

Fig. 4. Comparison of various hyperparameters.

The most optimal performance was achieved with the following parameters:

Network Structure 784, 150, 10

Learning Rate = 0.001

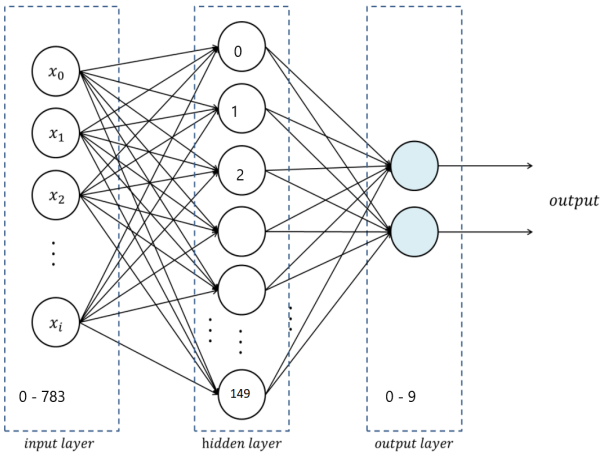

Fig. 5. Network Structure.

This model gave a training accuracy of 97.10% validation accuracy of 91.69% and a test accuracy on the trained network of 91.10%

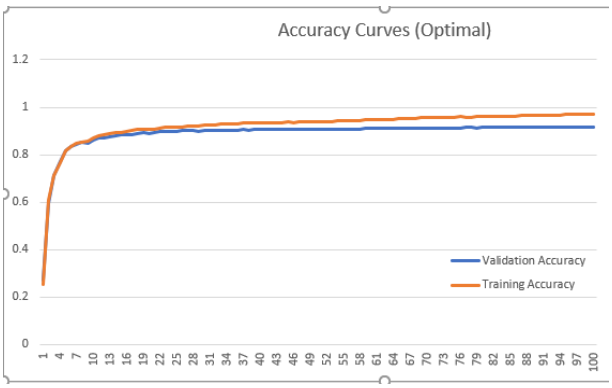

Fig. 6. Accuracy curve.

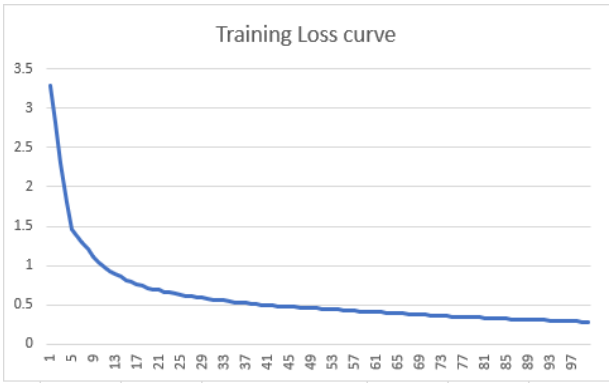

Fig. 7. Training Loss curve.