Neural Network Report

DANIEL GUO

DECEMBER 2018

DANIEL GUO
Freshman
CCS Computing & Mathematics
dguo@ucsb.edu
(408) 767-0880

1 Introduction

This report will talk about a neural network made in Python to approximate the function

$$f(x) = 2a(-x + 1) - 1,$$

where $a(x)$ is the ReLU function.

Two libraries were used:

random to assign initial weights and biases and to generate datasets for training and testing

matplotlib to visualize the results and the convergence of the loss function

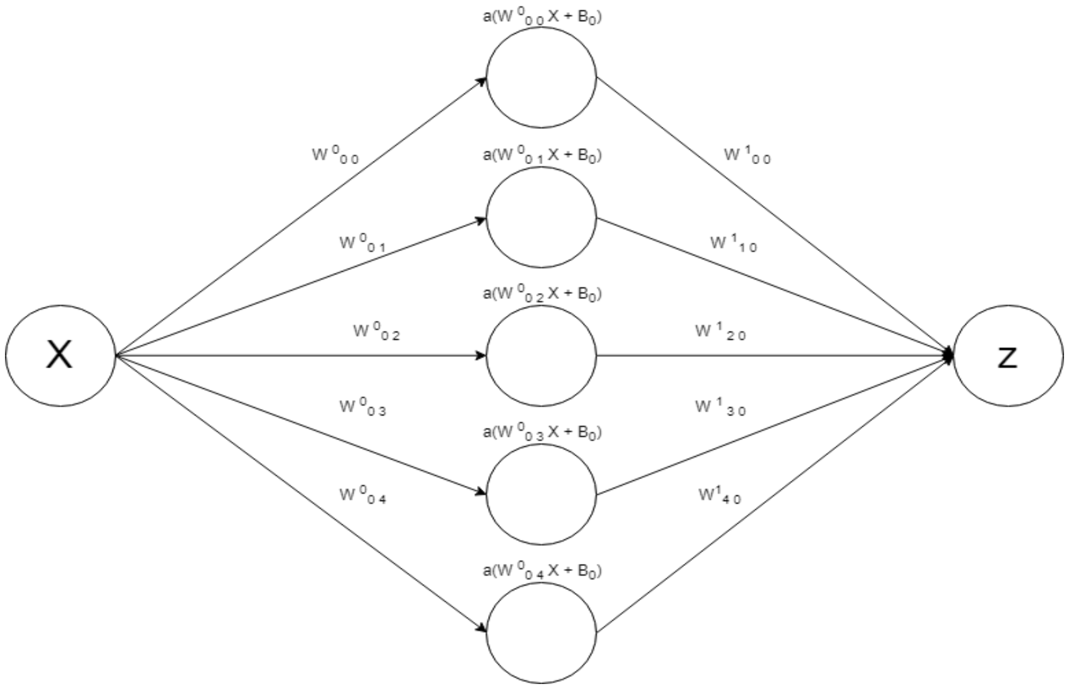
The full code and trained model can be found **here**.

2 General Design

The neural network consists of 1 hidden layer of 5 neurons. The decision for this architecture was made arbitrarily.[1] The chosen activation function was the rectified linear unit, or ReLU, and the chosen loss function was the mean squared error.

2.1 Visualizing the neural network

Let $f(x)$ denote the function we are trying to approximate. Let x denote the input parameter. Let y denote the output of the function $f(x)$. Let z denote the output of the neural net. Let $a(x)$ denote the activation function. Let $L(z, y)$ denote the loss function. Let N_i^j denote the neurons, where j is the layer number and i is the location of the neuron in that layer. Let W_{ij}^k denote the weights, with i, j, and k representing the neuron number on the left, the neuron number on the right, and the layer number to its left, respectively. Let B_i denote the biases, with i representing the layer number to its left.



[1] more discussion about this decision in the last section

2.2 Output of the neural network

$$\begin{aligned} z(x) = &W^1_{00}a(W^0_{00}x + B_0) \\ &+ W^1_{10}a(W^0_{01}x + B_0) \\ &+ W^1_{20}a(W^0_{02}x + B_0) \\ &+ W^1_{30}a(W^0_{03}x + B_0) \\ &+ W^1_{40}a(W^0_{04}x + B_0) + B_1 \end{aligned}$$

2.3 Training Process

There are 3 main parts to the training process:

1. **generating datasets** The training samples are generated and are run through the *feedforward* and *backprop* methods. The values of the weights and biases are recorded in a text document and the training stops when either the average loss reaches an acceptable error or a certain number of training iterations have passed.
2. **feeding foward** A dataset is put through the neural network to update all the values of the neurons.
3. **back propogation** The output of the neural network and the corresponding solution are used to change the weights and biases to minimize the loss function.[2]

3 Design Details

Here we talk about the details of the model and how each of the previously mentioned steps are implemented.

3.1 The *neuralnet* class

The *neuralnet* class has the following fields:

weights This is a matrix containing all of the weights, where the row is the layer number and the column is the location of the weight in that layer.

biases This is a list containg the biases.

nodevals This is a list of matrices, where each matrix represents the values stored in the neurons, where the row is the layer number of the neuron and the column is the location of the neuron in that layer. Each matrix represents the values for a particular input x, and how many matrices there are depends on the size of the training sample used.[3]

3.2 The *train* method

The *train* method takes in the following parameters: a learning rate, an acceptable error at which point the training is considered complete, the maximum number of training iterations at which point the training must stop, and the size of the datasets it generates.
First, it creates a list of numbers $[-100, 100]$ and a list of the corresponding solutions. This is the training sample. It runs this through the $feedforward$ and *backprop* methods, which will change the values of *nodevals*, *weights*, and *biases* accordingly[4]. It repeats this process and graphs the loss every 100 iterations (to visualize the convergence) until the model reaches the acceptable error or it has gone through the maximum number of training iterations. When either of these occur, the training is complete and the weights and biases are saved in a text document.

[2]the results of one trained model are discussed in the Results section
[3]2-dimensional lists are used to implement matrices
[4]the details of this are discussed in the next section

3.3 The *feedforward* method

The *feedforward* method takes in 1 parameter, the inputs from the training sample.
It updates all the values in *nodevals* using inputs in the training sample and carrying out the calculations with the respective weights and biases for each neuron.

3.4 The *backprop* method

The *backprop* method takes in 2 parameters, the learning rate and the corresponding solution set of the inputs that were put through the *feedforward* method from the training sample.
For each input, it creates a matrix of the partial derivatives of the loss function with respect to the weight or bias in the corresponding position in the *weights* and *biases* matrix/list using the values in *nodevals* for that input. After doing this for each of the inputs, it finds the average of them and updates the values of the weights and biases accordingly with the learning rate, which should (over several iterations) cause the value of the loss function to converge.[5]

4 Mathematics of the Back Propogation

Our loss function is $L(z, y) = (z - y)^2$. The partial derivative of the loss function with respect to a weight or bias, which we denote here with λ, is $\frac{\partial L}{\partial \lambda} = 2(z - y)(\frac{\partial z}{\partial \lambda})$. We find the value of $\frac{\partial z}{\partial \lambda}$ by finding the partial derivatives of $z(x)$ with respect to λ. We denote the derivative of the activation function by $\alpha(x)$.

$$\alpha(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

With this, we can find the partial derivative of $z(x)$ with respect to our weights and biases.

$$\begin{aligned}
\frac{\partial z}{\partial W^0_{0i}} &= W^1_{i0}\alpha(W^0_{0i}x + B_0)x \\
\frac{\partial z}{\partial W^1_{i0}} &= N^1_i \\
\frac{\partial z}{\partial B_0} &= W^1_{00}\alpha(W^0_{00}x + B_0) \\
&\quad + W^1_{10}\alpha(W^0_{01}x + B_0) \\
&\quad + W^1_{20}\alpha(W^0_{02}x + B_0) \\
&\quad + W^1_{30}\alpha(W^0_{03}x + B_0) \\
&\quad + W^1_{40}\alpha(W^0_{04}x + B_0) \\
\frac{\partial z}{\partial B_1} &= 1
\end{aligned}$$

We use these to find the values of $\frac{\partial L}{\partial W^k_{ij}}$ and $\frac{\partial L}{\partial B_i}$ for each input x. Then we find the average values of these from all the inputs from the training sample, and we have the gradient of the loss function:

$$< \frac{\partial L}{\partial W^0_{00}}, \frac{\partial L}{\partial W^0_{01}}, \frac{\partial L}{\partial W^0_{02}}, \frac{\partial L}{\partial W^0_{03}}, \frac{\partial L}{\partial W^0_{04}}, \frac{\partial L}{\partial W^1_{00}}, \frac{\partial L}{\partial W^1_{10}}, \frac{\partial L}{\partial W^1_{20}}, \frac{\partial L}{\partial W^1_{30}}, \frac{\partial L}{\partial W^1_{40}}, \frac{\partial L}{\partial B_0}, \frac{\partial L}{\partial B_1} > .$$

We multiply this by the learning rate and subtract each element of the gradient from its respective weight or bias. This is the learning part of the training, where the loss function takes steps in the direction of a local minimum. Repeating this process with numerous training samples will cause the loss function to converge to (or at least close to) a local minimum.

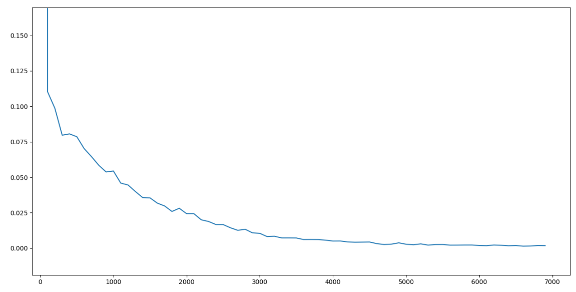
[5]details discussed in next section

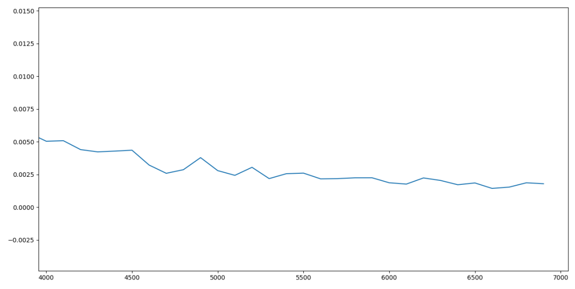
5 Results

The trained model included in the repository can approximate the mentioned equation to an average error of 0.0009930519902668856 on a set of 1000 numbers $[-100, 100]$ after 24676 training iterations using training samples of 1000 numbers $[-100, 100]$ with a learning rate of 10^{-5}.

5.1 Convergence of Loss Function

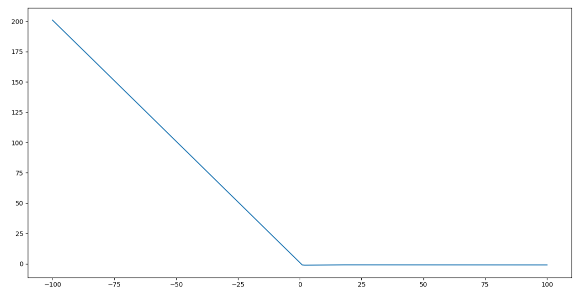
The following graphs show the convergence of the loss function over the number of training samples. The loss was plotted for every 100 training samples.

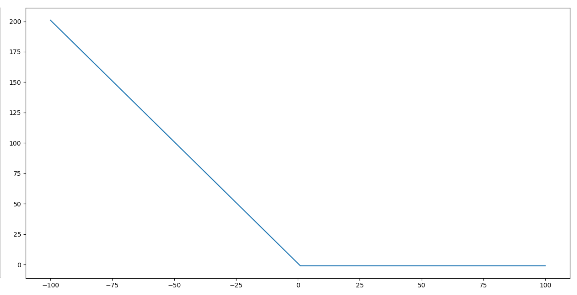




5.2 Performance

Graphing the function and the approximation of the trained model side by side,





they look very similar. Upon close inspection of the approximated values we can see some errors.

$$\begin{aligned}
z(-100) &= 200.96844751081605 &&\approx f(-100) = 201\\
z(-75) &= 151.01089226526028 &&\approx f(-75) = 151\\
z(-50) &= 100.99582757835947 &&\approx f(-50) = 101\\
z(-25) &= 50.980762891458646 &&\approx f(-25) = 51\\
z(0) &= 0.9656982045578304 &&\approx f(0) = 1\\
z(25) &= -0.9684557118210204 &&\approx f(25) = -1\\
z(50) &= -0.9853378098635837 &&\approx f(50) = -1\\
z(75) &= -1.0022199079061467 &&\approx f(75) = -1\\
z(100) &= -1.0191020059487097 &&\approx f(100) = -1
\end{aligned}$$

It seems that within our interval $[-100, 100]$, our trained model approximates the mentioned equation decently well.

6 Further Questions

How is the learning rate chosen? I wonder if the learning rate can be set proportionally to the loss from the previous training sample to result in a smoother and more stable convergence.
How many neurons are optimal in the hidden layer for a particular function? In a previous model using 3 neurons, the trained model didn't seem to be able to come close in accuracy, and the addition of the 2 neurons used in this model helped grealty. To what extent does more neurons mean more accuracy?
Overall, I'd like to learn more of the mathematics that goes on in making these and other decisions.