Exploration of the Mathematics behind Shallow Neural Networks

Thomas Narramore and Rhys Jordan

Introduction:

We set out to deeper understand the mathematics behind neural networks by exploring shallow neural networks (NN). Our exploration included finding the importance of non-linear activation functions and attempting to manual solve a shallow NN. We also coded an implementation of a shallow neural network to explore different hyperparameters as well as gain a better understanding of the mathematical steps involved in predicting a continuous output from a single input. Through this process we learned a lot about the behind the scenes of NNs and discovered the importance of certain aspects of neural networks that we questioned before.

The shallow NN that we will be inspecting is one that maps one input to one output with three nodes as seen in figure 1.

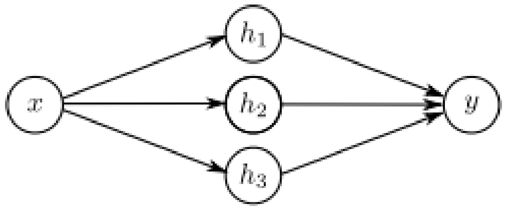


Figure 1: Image of Shallow NN with three nodes.

Basic Mathematics behind Shallow Neural Networks

Let's start out by explaining some of the math that makes up a shallow NN. Starting with forward propagation. If we are looking at the NN in figure 1, it has six weights and four bias. These can be represented as a list $\phi = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$.

$$\begin{aligned} y &= \mathrm{f}[x, \phi] \\ &= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]. \end{aligned}$$

Figure 2: Function representation of three node NN.

Using the initialized weights, we form three linear functions $\theta_{10} + \theta_{11}x, \theta_{20} + \theta_{21}x,$ and $\theta_{30} + \theta_{31}x$. Then those functions are put through an activation function a in this paper we will only use two activation functions reLU and Sigmoid. ReLU takes any values below zero and maps them to zero. Sigmoid bounds the output between zero and one. The equation and graph are displayed below in figure 3 and 4.

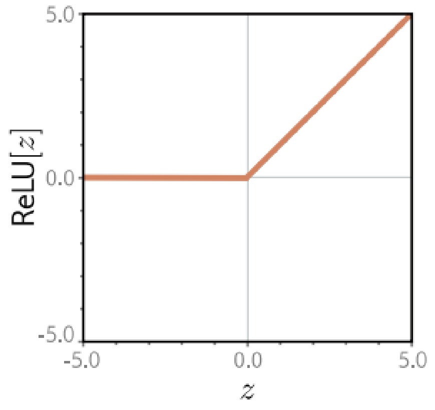


Figure 3: ReLU function with equation,

$$f(x) = \max(0, x)$$

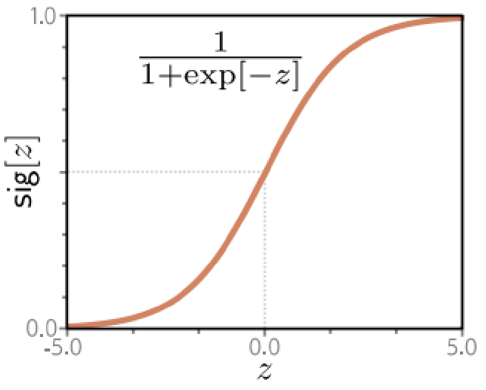


Figure 4: Sigmoid function with equation,

$$f(x) = \frac{1}{(1 + e^{-x})}$$

Once those functions are passed through the activation function, they are weighted again by ϕ_1, ϕ_2 *and* ϕ_3. Finally, a bias ϕ_0 is added. You can see the final equation in figure 2 above and below in figure 4 is where those parameters exist visually.

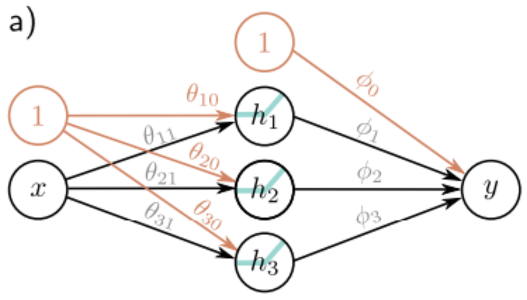


Figure 4: Visual representation of the weights and bias applied to the NN in figure 1.

This process is often referred to as forward propagation (Kim). The biggest part of a NN is determining the values of the weights and bias which are called parameters. These values are determined during the training phase of a neural network.

To find optimal values of the weights we seek to minimize the loss function. A loss function measures the difference between the expected value and predicted output of the network. Once the loss is calculated then using a process called gradient descent the parameters are updated with respect to a learning rate.

Let's look at what gradient descent is and how to calculate it. The first step is to calculate the derivatives of loss with respect to the parameters as seen in figure 5 which tells us the uphill gradient of the loss function. The goal is to move downhill in respect to the loss function because we want to minimize the loss, so we adjust the parameters in the opposite direction of the derivative as shown by the negative in figure 6. The learning rate determines how much we move in the direction determined by the previous step and is represented by α in figure 6.

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}.$$

Figure 5: Derivative of Loss Function with respect to Parameters

$$\phi \longleftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi},$$

Figure 6: Update done to the parameters

Backpropagation is an effective algorithm for calculating the derivative of the loss with respect to the parameters. The development of backpropagation was an important step in the machine learning world and allow access to a wider field of problems that were previously limited due to time and cost constraints (“Backpropagation”). It is extremely efficient because it uses mostly matrix multiplication which is computationally more efficient than taking the derivatives the traditional way.

Can we manually solve a NN?

The goal of a neural network is to find the optimal parameters to predict an output that is as close as possible to the expected output. To do this, we are trying to minimize the loss function. So, we asked the question if it was possible to manually solve the loss function to get those optimal weight without training the network using gradient descent. We knew that this is intractable, but we were curious about why this is exactly. We showed that doing this would equivalent to solving a non-linear system of equations to minimize the error of every training datum, which is intractable.

Implementation

We wrote two scripts in the programming language python to help us explore a neural network the specifics of a neural networks such as visualizing forward propagation and the training process.

The first script we created was to explore the forward propagation process explained in the mathematics section. The parameters are manually determined and static so there is no logic to train the parameters. The goal of this script was to replicate figure 7 from section 3.1.2 in *Understanding Deep Learning* by Simon J.D. Prince and gain a better understanding of what each parameter did and how it affects the overall output function. The example in the book used the reLU activation function so that is what we implemented in the script to replicate the picture.

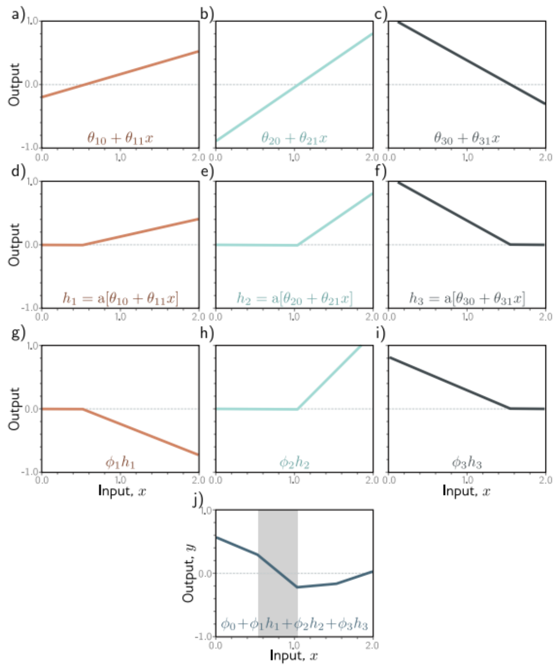


Figure 7: Visual Repersenation of the Feed Forward Process

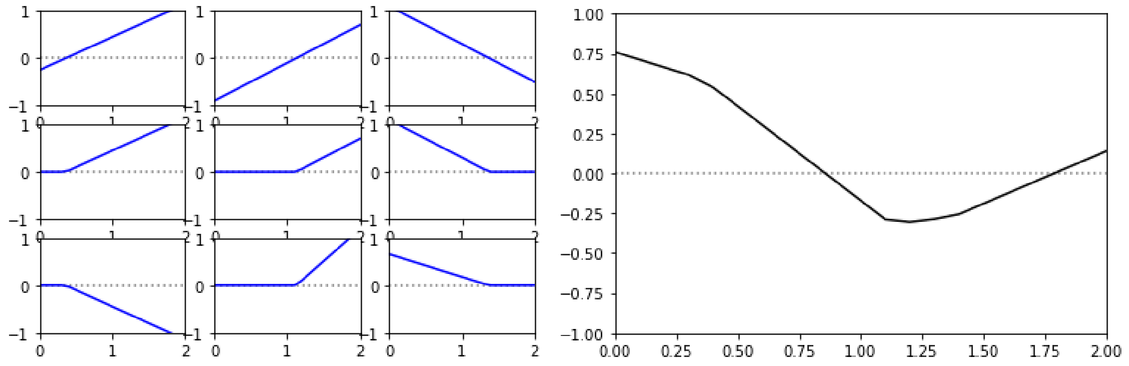


Figure 8: Outputs of UAT_test.py

After continuous trial and error, we were able to get an accurate representation with the parameters:

$$\{\phi_0 = 0.1, \phi_1 = -1, \phi_2 = 1.7, \phi_3 = 0.6, \theta_{10} = -0.25, \theta_{11} = 0.7, \theta_{20} = -0.9, \theta_{21} = 0.8, \theta_{30} = 1.1, \theta_{31} = -0.8\}$$

We started with just trying to replicate the set of linear functions shown in the first row of figure 7. After finding parameters that closely represented those functions, we implemented a reLU activation function which transformed those linear functions to the functions in the second row of figure 7. Finally, we adjusted the weights applied to those functions to transform them into the last row of function in figure 7. Our resulting function can be seen in figure 8.

The second script is an implementation of a shallow neural network with three nodes as seen in figure 1. Lots of this implementation is based off the article by Ricky Kim about building a shallow neural network from scratch. His article used a planar dataset, and we adjusted our code to solve the regression problem of approximating a continuous function. A big thing we changed was the cost function. We implemented a mean squared error (MSE) function to calculate loss while he used the cross-entropy which is most often used in classification problems. Mean squared error is the default loss function for regression problems and it is easy to implement (Brownlee). The equation for MSE is seen below in figure 9.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE	= mean squared error
n	= number of data points
Y_i	= observed values
$\hat{Y}_i$	= predicted values

Figure 9: The equation for mean square error

We also wanted the ability to visualize the approximation of the line after each iteration to make observations as we changed the hyperparameters. So, we implemented a plot function that would plot the expected output and the networks output.

Discoveries

We began by verifying that linear activation functions only approximate linear functions. We worked out the equations for solving an arbitrarily wide shallow neural network with a linear activation function and arbitrary number of inputs and outputs. We found that terms grouped together to output a linear function, so there was no benefit to using the neural network structure. We then tried solving the same network, but with a quadratic activation function. When the equation describing the network was expanded, it worked out to a 4th degree polynomial. This makes sense because in a single layer network, the activation functions is contained in itself one time – the equation describing the output layer in terms of the weights of the hidden layer is quadratic, and the weights of the hidden layer have quadratic activation functions. Composing a quadratic function with another quadratic function yields a 4th degree polynomial. Adding another layer with quadratic activations functions would be the equivalent of composing a fourth degree polynomial within a quadratic, yielding an 8th degree polynomial. In general, a neural network with n layers with kth degree activation functions can approximate a polynomial of degree k^(n+1). This is decent – polynomials can approximate continuous functions arbitrarily well. It seems that using a polynomial activation function turns the problem into the problem of interpolating data with a polynomial. For a function of one variable this can be done in polynomial time, which is good news. The bad news is that now it is just polynomial interpolation, which is not that interesting. More investigation would be needed to determine if this is still easy to solve for multivariable functions, and whether there is any practical use for quadratic activation functions in neural networks. The mathematical detail of this work is in the attached slides.

After justifying that a neural network must have a non-linear activation function to solve more than a linear regression problem we explored some non-linear activation functions. Using the shallowNN.py script we observed that a network using a sigmoid activation function converged on the expected output much faster than a network with a reLU activation function. We used a network with three nodes, a learning rate of 0.07, and MSE as the loss function. After only 10 training iterations the network using a sigmoid activation function has a loss of 0.0033 as seen in figure 11. Approximating the same output and after 10 training iterations the network using a ReLU function had a loss of 0.0428 as seen in figure 10.

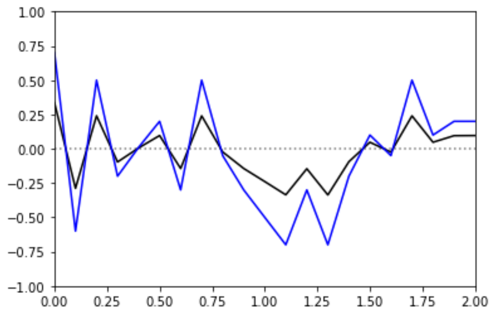


Figure 10: Output approximated with ReLU activation function

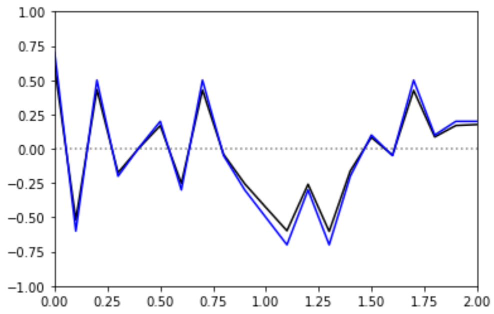
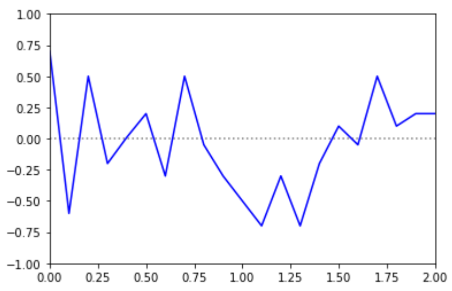


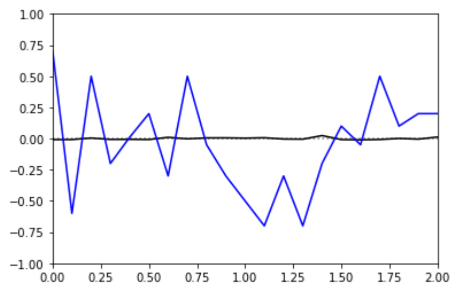
Figure 11: Output approximated with Sigmoid activation function

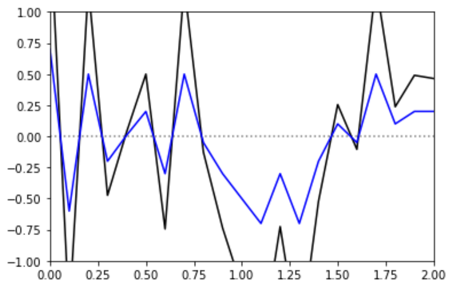
Even though we observed this in our script sigmoid activation functions are not widely used in the hidden layers of neural networks especially when the networks get bigger and have multiple layers. This is because sigmoid is much more computationally expensive than reLU. In a large neural network fast computation is important because without it networks could take ages to train (Sakshi).

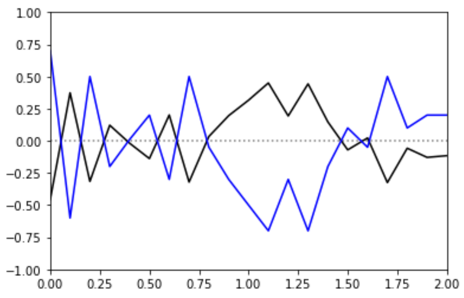
The learning rate was also a hyperparameter that we messed around with. We started with a smaller learning rate of $\alpha = 0.07$. Decreasing the learning rate slowed down how fast the line converged to the expected output which makes sense because the weights are changing less each iteration. Then when we increased the learning rate it created crazy visualizations showing how drastically the output changed with each iteration.

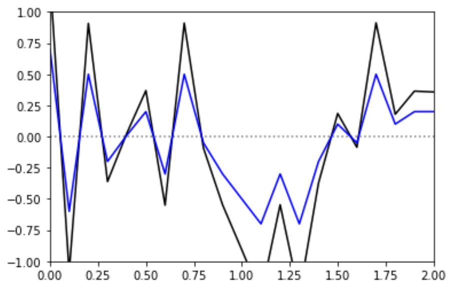


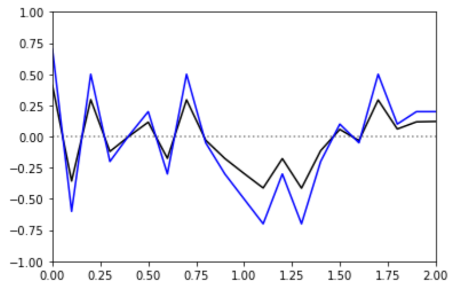
Iteration 1

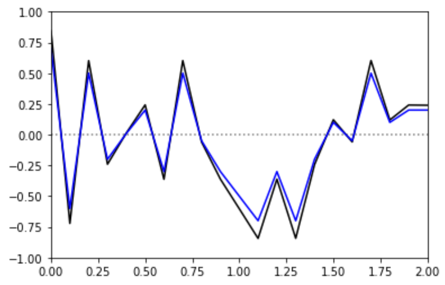

Iteration 2

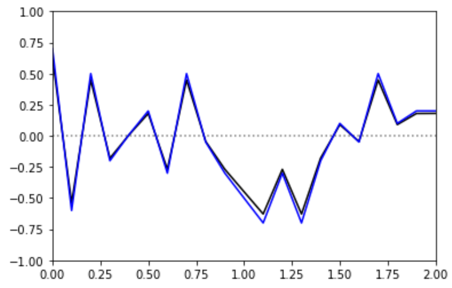

Iteration 3

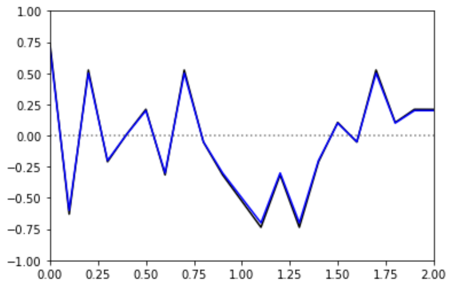

Iteration 4


Iteration 5


Iteration 6


Iteration 7


Iteration 8


Iteration 9

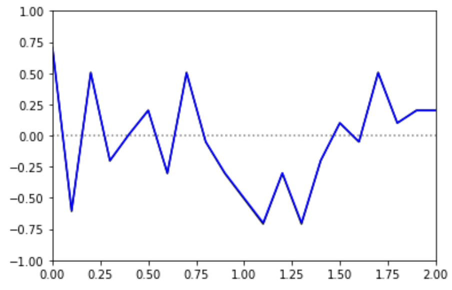

Iteration 10

Figure 12: Graphs produced by shallowNN.py with a learning rate of 1.5 and sigmoid activation function.

This visualization shows that changing the weights at a higher rate makes the output jump around more before it converges. Something like a decaying learning rate might be useful when creating a neural network to predict more difficult outputs. A decaying learning rate starts with a high learning rate and as loss decreases so does the learning rate (Prince). It might be useful to increase the parameters more at the beginning of training to find when the loss is the least. Then as the function converges decreases the learning rate to get closer to a minimum of the loss function.

Conclusion

Throughout this process we were able to explore the mathematical concepts that make up neural networks while focusing on a three-node shallow neural network. We gained intuition on why efficient neural networks are built the way that they are. Breaking down a simple shallow neural network allowed us to explore different parameters and hyperparameters and immediately see how it affected the output. Also allowed us to visualize how those parameters and hyperparameters affected the output at each training iteration.

It would be interesting to continue to look at breaking down NNs and what they are doing behind the scenes to better understand how they are able to predict outputs. We would like to better explore backpropagation and the math behind this concept which was out of scope for this project. We would also like to try and see how big of data the shallowNN.py implementation can handle as well as increase the number of nodes that we examined.

Work Cited

"Backpropagation." *Brilliant.Org*, brilliant.org/wiki/backpropagation/. Accessed 3 May 2024.

Brownlee, J. (2020, August 25). *How to choose loss functions when Training Deep Learning Neural Networks*. MachineLearningMastery.com. https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Kim, Ricky. "Shallow Neural Network from Scratch ." *LinkedIn*, 10 Mar. 2018, www.linkedin.com/pulse/shallow-neural-network-from-scratch-deeplearningai-assignment-kim/.

Prince, Simon J.D. *Understanding Deep Learning*. The MIT Press, 2023.

Sakshi, Tiwari. "Activation Functions in Neural Networks." *GeeksforGeeks.Org*, 17 Feb. 2023, www.geeksforgeeks.org/activation-functions-neural-networks/.