Backpropagation In Neural Networks

MTE 203

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:

The University of Waterloo

Professor Harris

Prepared By:

Rayan Ahmad

20953459

150 University Ave W.

Waterloo, Ontario, N2N 2N2

November 24th, 2023

Table of Contents

Table of Figures

Table of Tables

Introduction

Neural networks, sophisticated computational models inspired by the biological neural networks of animal brains, serve as the cornerstone of modern artificial intelligence. These networks emulate the remarkable human ability to discern patterns and make connections, enabling machines to "learn" from vast amounts of data. From an engineering perspective, replicating the complex functioning of neurons requires a mathematical and programmatic representation of these biological processes. The utility of neural networks spans a variety of applications, revolutionizing fields such as data analytics, pattern recognition, and image processing. They excel in tasks that include identifying trends within large data sets, recognizing objects within images, and even converting handwritten text into digital characters. This versatility has made them indispensable tools in the technology sector and beyond. Training neural networks is akin to human learning, reliant on experience and data. The calibration of these networks is imperative to ensure they predict outcomes accurately based on the data they are presented with. Consider a neural network designed to recognize birds: initially, it may mistakenly identify a bird as a cat. However, through a training process where it is shown multiple images of birds and the correct identification is reinforced, the network learns to classify birds accurately. At the heart of this learning process lies backpropagation, a pivotal algorithm in machine learning. Backpropagation fine-tunes the neural network by adjusting its parameters in the direction that reduces the error between the network's prediction and the actual data. This is achieved using mathematical concepts such as the gradient of a cost function, which quantifies the network's performance. To navigate the complexities of backpropagation, one must be knowledgeable with key mathematical concepts including partial derivatives, the chain rule, and gradients. These elements are not only integral to understanding how neural networks learn but are also foundational to the discussions that will follow in this report. As we delve deeper into the intricacies of backpropagation, a simple multilayer perceptron implemented from scratch, will be used to explore these mathematical tools and demonstrate their application in predicting the result of an XOR gate based on its inputs.

Background

Artificial neural networks are computational models designed to emulate the neurological processes of the human brain. These networks comprise of artificial neurons, the fundamental units in a network layer. Each neuron includes an input, an activation function, a bias, and a weight connecting it to the next neuron in the subsequent layer [1]. The activation function determines whether a neuron should be activated, based on its input. Activation functions output a range of values, with the sigmoid function in Equation 1 being a common example [2]. The sigmoid function's graph in Figure 1 shows that as x approaches negative infinity, y approaches 0, and as x approaches positive infinity, y approaches 1.

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

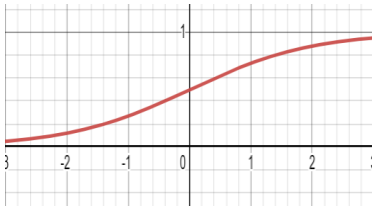

Figure 1: Sigmoid Activation Function Plotted

Neural networks consist of three types of layers. The first one being the input layer which receives initial parameters and data as the input. It processes and passes this information onto the next layer [2]. Next are hidden layer(s), these layers can be one or multiple and take their input from the input layer or preceding hidden layers [2]. They further analyze and process the data before passing it to the next layer. Lastly, the output layer, which provides the final results after processing all the data, has single or multiple neurons depending on the complexity of the problem [2]. See Figure 2 for an example.

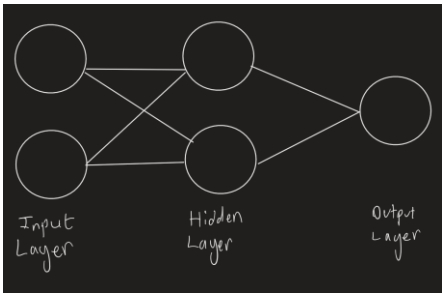

Figure 2: Simple Neural Network with Input, Hidden, and Output Layers

Forward propagation is when inputs are fed into the neural network, each neuron passes its input through an activation function and then to each of the neurons in the next layer, applying weights and biases [3]. This is continued in all layers until the output layer, with predicted output being the final value, this is described in Equations 2 and 3, where $a^{[l]}$ is the current layer's activation, $W^{[l]}$ is the weight between the previous neuron and current neuron, and $b^{[l]}$ is the current neurons bias.

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \quad (2)$$

$$a^{[l]} = \sigma\left(z^{[l]}\right) \quad (3)$$

Initially, these weights and biases are set randomly and must be fine-tuned for the network's specific application. Training data, comprising of inputs and their expected outputs, plays a crucial role in fine-tuning neural networks [2]. The network processes the input from the training data and generates its output. The loss, or difference, between the predicted output and the actual output is then calculated. The network computes this loss for each piece of training data and averages these losses to determine the total cost, which indicates the network's predictive accuracy. The cost function is a mean squared error function where the weights and biases are inputs, and its output is the cost, a single value determined by the training data [4]. This is shown in Equation 4, where the W, and b are the weights and biases of the entire network, m is the number of training examples, y_i is the desired output, and $a^{[l]}$ is the predicted output.

$$C(W, b) = [W_1, b_1, \dots, W_l, b_l] = \frac{1}{2m}\sum_{i=1}^{m}\left(y_i - a^{[l]}\right)^2 \quad (4)$$

The goal is to minimize this cost, which can be achieved using gradient descent, which involves calculating the gradient of the cost function, this is shown in Equation 5. This indicates the direction of steepest ascent. By subtracting the gradient of the cost function from the input weights and biases continuously, the network approaches a local minimum [3]. To calculate the gradient, the chain rule is employed. Figure 3 shows the relationship between the cost function and the weights and biases. Hence, the gradient can be derived by taking the partial derivative of C with respect to W^l, b^l, and $a^{(l-1)}$ as shown in Equations 6, 7, and 8, respectively.

$$\nabla C = \left[\frac{\partial C}{\partial W_1}, \frac{\partial C}{\partial b_1}, \ldots, \frac{\partial C}{\partial W_l}, \frac{\partial C}{\partial b_l}\right] \quad (5)$$

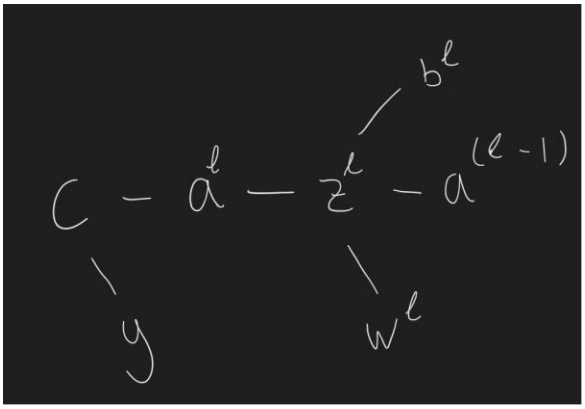

Figure 3: Tree Diagram of Cost Function

$$\frac{\partial C}{\partial W^{[l]}} = \frac{\partial C}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial W^{[l]}} \quad (6)$$

$$\frac{\partial C}{\partial b^{[l]}} = \frac{\partial C}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial b^{[l]}} \quad (7)$$

$$\frac{\partial C}{\partial a^{[l-1]}} = \frac{\partial C}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial W^{[l]}} \quad (8)$$

The partial derivatives are solved individually in Equations 9, 10, 11, 12, and 13.

$$\frac{\partial C}{\partial a^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - a^{[l]}\right) \quad (9)$$

$$\frac{\partial a^{[l]}}{\partial z^{[l]}} = \sigma'\left(z^{[l]}\right) \quad (10)$$

$$\frac{\partial z^{[l]}}{\partial W^{[l]}} = a^{[l-1]} \quad (11)$$

$$\frac{\partial z^{[l]}}{\partial b^{[l]}} = 1 \quad (12)$$

$$\frac{\partial z^{[l]}}{\partial a^{[l-1]}} = W^{[l]} \quad (13)$$

Subbing the partial derivatives back results in Equations 14, 15, 16

$$\frac{\partial C}{\partial W^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - a^{[l]}\right) \cdot \sigma'\left(z^{[l]}\right) \cdot a^{[l-1]} \quad (14)$$

$$\frac{\partial C}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - a^{[l]}\right) \cdot \sigma'\left(z^{[l]}\right) \quad (15)$$

$$\frac{\partial C}{\partial a^{[l-1]}} = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - a^{[l]}\right) \cdot \sigma'\left(z^{[l]}\right) \cdot W^{[l]} \quad (16)$$

Backpropagation is the process of adjusting the network's weights and biases based on the gradients calculated. This involves changing the current layer's weights, biases, and weights and biases to affect activations from previous layers [5]. The adjustments for each output neuron and layer are accumulated and applied to the entire network. The learning rate, α, determines the size of the steps

taken towards the minimum of the cost function during the training process. A smaller learning rate ensures more precise convergence, avoiding overshooting the minimum, but can make the training process slower and potentially get stuck in local minima. Conversely, a larger learning rate accelerates the training process but risks overshooting the minimum, leading to unstable convergence or even divergence [5]. After applying backpropagation to all training data, the average adjustments for each weight and bias are computed and applied. The new weights and biases are calculated using Equations 17, and 18, respectively. This process is repeated recursively, working back from the output to the input layer and is applied on all the hidden layers weights and biases as well. The network is finally fine-tuned and ready for practical use.

$$W_{new} = W_{old} - \alpha \frac{\partial C}{\partial W} \quad (17)$$

$$b_{new} = b_{old} - \alpha \frac{\partial C}{\partial b} \quad (18)$$

Approach

In the remainder of this paper, a simple multilayer perceptron (MLP) will be implemented from scratch using Python. The MLP's ability will be demonstrated by using it to solve the XOR (exclusive OR) problem. The XOR function is a fundamental binary operation applied to two binary inputs (0 or 1) [6]. The XOR function outputs 1 if the inputs are different (i.e. one input is 0 and the other is 1) and outputs 0 if the inputs are the same (both 0 or both 1), see Table 1. The XOR function is not linearly separable, which means it is impossible to separate the outputs (0s and 1s) with a single straight line [6].

Table 1: Truth Table for XOR Function

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

The MLP will consist of an input layer, at least one hidden layer, and an output layer. The input layer will receive the binary inputs, the hidden layer(s) will process these inputs, and the output layer will produce the final prediction. The MLP will be trained using a set of examples from Table 1. During training, the network will learn to map the given inputs to their corresponding outputs. Backpropagation will be utilized to train the network. The effectiveness of the MLP will be demonstrated by collecting data on the number of iterations and its accuracy at every 100 iterations. This data will then be graphed to illustrate the network's increasing accuracy in predicting the correct outputs over all its training iterations. The performance of the MLP will be evaluated based on its final accuracy in predicting the XOR function's outputs. The accuracy will be calculated as the proportion of correct predictions made by the network compared to the total number of predictions.

Implementation

To construct the MLP, Python is utilized for its versatility and ease of use in handling complex calculations. The NumPy library, renowned for its efficient handling of large matrices, is used [7]. The design of the MLP for the XOR problem will be as follows; the input layer, given the binary nature of the XOR inputs, will consist of two neurons. The hidden layer, to capture the non-linearity of the XOR

function, will contain two neurons. This design choice is critical, as a single neuron would result in a linear model, incapable of accurately solving the XOR problem. Finally, the output layer contains a single neuron, representing the binary output of the XOR function. Activation of this neuron corresponds to an output of one, otherwise zero.

```
inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2, 2, 1
hidden_weights = np.random.uniform(size=(inputLayerNeurons, hiddenLayerNeurons))
```

The network initializes with random weights and biases for both the hidden and output layers. The number of weights between the input and hidden layers equals four, aligning with the connections between each input neuron and each hidden neuron.

```
hidden_bias = np.random.uniform(size=(1, hiddenLayerNeurons))
output_weights = np.random.uniform(size=(hiddenLayerNeurons, outputLayerNeurons))
output_bias = np.random.uniform(size=(1, outputLayerNeurons))
```

The sigmoid function is chosen as the activation function due to its ability to introduce non-linearity into the network. Both the function and its derivative are implemented in code, as the derivative is essential for the backpropagation process.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

The inputs and expected outputs for training the MLP are represented as matrices.

```
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([[0], [1], [1], [0]])
```

The forward propagation process involves calculating the dot product of the input and the weights, adding the hidden layer bias, and applying the sigmoid function. This yields the hidden layer's values. These values, along with the output layer's weights and bias, undergo a similar process to predict the network's output [9].

```
hidden_layer_activation = np.dot(inputs, hidden_weights)
hidden_layer_activation += hidden_bias
hidden_layer_output = sigmoid(hidden_layer_activation)
output_layer_activation = np.dot(hidden_layer_output, output_weights)
output_layer_activation += output_bias
predicted_output = sigmoid(output_layer_activation)
```

Then for backward propagation, the learning rate is declared to be 0.1, the initial step involves calculating the error as the difference between the actual outputs and the predicted outputs of the network. This error is then element-wise multiplied by the derivative of the sigmoid function applied to the predicted outputs, referred to as d_predicted_output. This derivative calculation is crucial for understanding the necessary adjustments to reduce the error [9].

```
lr = 0.1
error = outputs - predicted_output
d_predicted_output = error * sigmoid_derivative(predicted_output)
```

The error is backpropagated to the hidden layer by multiplying d_predicted_output with the transpose of the output weights, resulting in error_hidden_layer. This step assesses each hidden layer neuron's contribution to the output error.

```
error_hidden_layer = d_predicted_output.dot(output_weights.T)
d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)
```

Finally, the hidden and output weights are updated using the transposed hidden layer output and d_predicted_output, scaled by the learning rate. This step is part of gradient descent to minimize error. The output bias is adjusted by summing d_predicted_output across all samples and scaling by the learning rate. The input-to-hidden weights are updated using the transposed input matrix and d_hidden_layer, again scaled by the learning rate. The hidden layer bias is modified by summing d_hidden_layer across all samples, with the learning rate applied [9].

```
output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * lr
hidden_weights += inputs.T.dot(d_hidden_layer) * lr
hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr
```

Now one successful forward and backward propagation has been completed, this process needs to be repeated for the accuracy of the network to be improved. Using a for-loop we can run this process 10000 times.

Results

An epoch is the number of iterations completed, in this case its 10000 [7]. To track the improvement in the network, the iteration number and accuracy is recorded in an array for every 100th iteration.

```
if _ % 100 == 0:
        accuracy_data.append(1 - np.mean(np.abs(outputs - predicted_output)))
```

Finally using matplotlib, a graph showing the number of iterations or epochs on x-axis is graphed against the accuracy of the network on the y-axis [8].

```
plt.figure(figsize=(12, 6))
plt.plot(epochs, accuracy_data, marker="o", color="b", linestyle="-",
linewidth=2, markersize=4,)
plt.title("Accuracy over Epochs")
plt.xlabel("Number of Epochs")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```

One of the primary indicators of the MLP's performance is the change in accuracy across different numbers of training iterations, or epochs. The graph in Figure 4 distinctly illustrates this relationship. Initially, as the number of epochs increases, the network's accuracy in predicting the XOR problem's output shows a significant improvement. This trend is indicative of the network effectively learning from the training data, with the backpropagation algorithm successfully minimizing the loss function and adjusting the weights and biases.

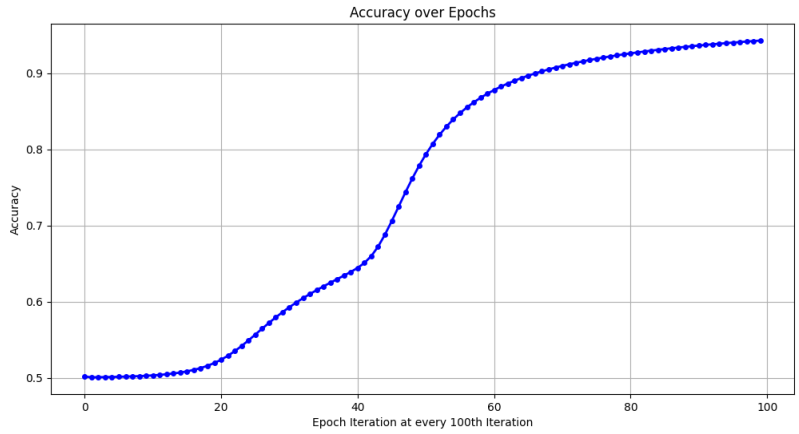

Figure 4: Accuracy vs. Epochs at Every 100th Iteration

However, an interesting observation is noted when the number of epochs exceeds 70 on the x-axis. Beyond this point, the graph demonstrates a plateau in the improvement of accuracy. Essentially, additional training iterations beyond 7000 epochs do not contribute substantially to enhancing the model's accuracy. This plateau can be attributed to the network reaching a point where further adjustments to weights and biases yield diminishing returns in terms of predictive accuracy. It suggests that the network has effectively learned the pattern of the XOR function to its capacity within the given architecture and parameter settings.

Discussion

The performance of the MLP, as evidenced by the final accuracy of 94%, indicates a high level of success in predicting the output of the XOR problem. This result is significant, considering the XOR problem's historical role in demonstrating the limitations of single layer perceptrons and the need for more complex architectures in neural networks [6]. The MLP's ability to accurately predict the XOR outputs validates the effectiveness of using multiple layers and the backpropagation algorithm in neural network training. The high accuracy achieved by the MLP underscores a few important points. The effectiveness of network design, consisting of an input layer, a hidden layer, and an output layer, was adequately designed to capture the non-linear nature of the XOR problem. The training process involving forward and backpropagation with an appropriate number of epochs, was effective in adjusting the network's weights and biases to minimize the error in predictions. The utility of backpropagation was useful and reinforces the importance of the backpropagation algorithm in neural network learning, especially in problems involving non-linear relationships.

While the MLP performed well on the XOR problem, several avenues for future research emerge from this study including exploring different network architectures, such as varying the number of hidden layers or neurons, could provide insights into the MLP's performance on more complex problems. Adjusting learning rate or activation functions may have a better impact on the efficiency and accuracy of the network. Extending the MLP to more complex and practical problems, such as image recognition or natural language processing, is a natural next step and could test its adaptability and robustness. Comparing the MLP's performance with other machine learning models, like convolutional neural

networks or recurrent neural networks, on the same or similar problems could offer a broader perspective on its relative strengths and weaknesses.

Conclusion

This report has successfully demonstrated the foundational principles and practical application of backpropagation in neural networks through the implementation of a MLP to solve the XOR problem. The basic concept of neural networks, drawing inspiration from biological neural processes and translating these into computational models using mathematical and programmatic representations, led to the creation of the artificial neuron. This includes the structure of the neuron, its weights, biases, and activation function, and the various types of layers in a neural network, namely the input, hidden, and output layers. The role of activation functions, particularly the sigmoid function, was explained to illustrate how neurons are activated based on input data. The process of forward propagation was detailed, demonstrating how the network processes input to produce an output. Training data was emphasized as crucial in determining the network's accuracy, leading to the exploration of the cost function. The cost function, an important element in neural network training, serves as a measure of the network's performance based on its weights and biases. Gradient descent emerged as a key method for optimizing these parameters, with the derivation of equations to calculate the function's gradient. A significant aspect of this study was the implementation of the backpropagation algorithm. This algorithm, central to the learning process of the network, was used to iteratively adjust weights and biases, refining the model based on the gradients obtained and the learning rate. The XOR problem, historically significant in illustrating the limitations of single-layer networks, provided a practical challenge for our MLP. The MLP was implemented in Python and included an input layer, a hidden layer to introduce non-linearity, and an output layer. The backpropagation algorithm was instrumental in training the network, involving forward propagation for predictions and backward propagation for updating network parameters based on the error. The results of this implementation were encouraging. The MLP attained a 94% accuracy in solving the XOR problem, clearly demonstrating improved performance with increasing epochs. However, a plateau in accuracy improvement was noted, signaling an optimal training duration and the network's capacity for this specific task. The backpropagation algorithm in training a neural network can be extended to address more complex and practical problems, such as image recognition or natural language processing. Further avenues of discussion include comparing the MLP's performance with other machine learning models, like convolutional neural networks or recurrent neural networks, on the XOR problem to get a better understanding of the model’s individual weaknesses and strengths.

References

[1] R. J. Erb, “Introduction to backpropagation neural network computation - pharmaceutical research,” SpringerLink, https://link.springer.com/article/10.1023/A:1018966222807 (accessed Nov. 20, 2023).

[2] M. Madhiarasan and M. Louzazni, “Analysis of Artificial Neural Network: Architecture, types, and forecasting applications,” Journal of Electrical and Computer Engineering, https://www.hindawi.com/journals/jece/2022/5416722/ (accessed Nov. 22, 2023).

[3] J. Zhang, “Gradient descent based optimization algorithms for deep learning models training,” arXiv.org, https://arxiv.org/abs/1903.03614 (accessed Nov. 23, 2023).

[4] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back‐propagating errors," Nature, California, 1986. (accessed Nov. 23, 2023).

[5] B. J. Wythoff, "Backpropagation neural networks: A tutorial," Elsevier Science Publishers B.V., Amsterdam, 1992. (accessed Nov. 23, 2023).

[6] J. Singh and R. Banerjee, "A Study on Single and Multi-layer Perceptron Neural Network," 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2019, pp. 35-40, doi: 10.1109/ICCMC.2019.8819775. (accessed Nov. 24, 2023)

[7] “NumPy documentation#,” NumPy documentation - NumPy v1.26 Manual, https://numpy.org/doc/stable/ (accessed Nov. 24, 2023).

[8] “Using matplotlib#,” Using Matplotlib - Matplotlib 3.8.2 documentation, https://matplotlib.org/stable/users/index (accessed Nov. 24, 2023).

[9] ChatGPT was used to help create parts of the code.