Implementing and Optimizing Neural Network in C++

Abdulaziz Alshehri

Electrical and Computer Engineering

The University of Texas at San Antonio

BSE 1.506 San Antonio, TX 78249

Abdulaziz.alshehri@my.utsa.edu

Chao-Jia Liu

Electrical and Computer Engineering

The University of Texas at San Antonio

BSE 1.506 San Antonio, TX 78249

chao-jia.liu@my.utsa.edu

***Abstract*—Numerous research papers and tutorials emerge, guiding individuals on implementing neural networks. However, the majority of these resources are oriented to Python, leveraging its extensive libraries. In contrast, despite ongoing efforts to implement neural networks in C++ [1]-[6], there remains a relative scarcity of attention in this domain. In this project, our aim is to implement a neural network from scratch using C++. Employing two inputs, 0 and 1, we train our neural network to realize an Exclusive OR behavior. We offer insights into the full training process, allowing for the visualization of error and accuracy changes over successive training passes. Moreover, we examine five different aspects of our neural network: Net recent error, Average error for Target 0, Average error for Target 1, Overall average error, and Recent accuracy, to analyze trends. Additionally, we explore the impact of altering the network's topology, specifically its size, on performance metrics. Through optimization techniques, we strive to enhance overall accuracy. Furthermore, we store the results during each training pass and utilize Gnuplot [6][7], a C++ package for Cygwin, to generate comprehensive visualizations for our neural network. This facilitates easy comparison between different network structures and epochs.**

***Keywords*—Neural Network Implementation, C++ neural network from sketch, Gnuplot image plotting, and Optimization of neural network.**

I. INTRODUCTION

In the realm of artificial intelligence, neural networks stand out as a powerful tool for data analysis, pattern recognition, and decision-making. Their implementation spans various programming languages and platforms, each with its own strengths and challenges. When considering the implementation of neural networks, Python often comes to mind first due to its simplicity, extensive libraries, and rapid prototyping capabilities. However, C++ offers distinct advantages, particularly in terms of performance, memory management, and integration with existing codebases. For computationally intensive tasks or projects requiring fine-grained control over resource utilization, C++ can provide superior efficiency and scalability.

Another strength of C++ is that it enables programmers to gain a deeper understanding of the neural network's underlying structure and mechanisms. By implementing a neural network in C++, programmers can delve into the intricacies of linking neurons, connecting layers, and implementing feedforward and backpropagation algorithms. Figure 1 shows one of the neural network architectures implemented in this project. Furthermore, utilizing fundamental data structures and memory management techniques such as vectors and class structures allows for a more hands-on exploration of concepts taught in academic settings. Completing such a project enhances our comprehension of neural network principles and also solidifies our understanding of how to effectively apply the techniques covered in class.

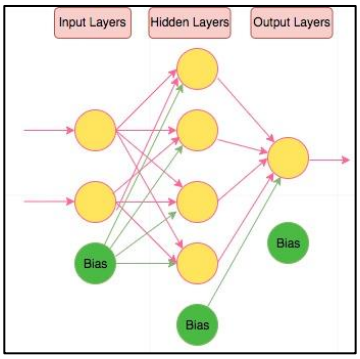


Figure 1. Neuron Network Structure, featuring one input layer with 2 neurons, one hidden layer each containing 4 neurons, and one output layer with 1 neuron. Each layer will be added a bias neuron.

II. MOTIVATION

To construct a neural network from scratch in C++, without relying on existing neural network libraries, entails fundamental concepts such as utilizing vectors for data storage and management and employing class structures to represent connections between layers and neurons.

Our motivation for undertaking this project is outlined below:

1) Neural Network Structure: We aim to comprehend the functionality of each layer within a neural network and understand how these layers are interconnected. By dissecting the architecture of neural networks, we seek to gain insights into their operational mechanisms and the flow of information between layers.
2) Neurons and Algorithms: Delving into the concept of neurons, we explore their fundamental workings and the underlying algorithms that drive their behavior. Understanding the intricacies of neuron operations is crucial for grasping the essence of neural network computation and learning processes.
3) Class Structure Design: We strive to master the art of designing class structures that effectively compartmentalize different components of the neural network based on their functionalities. By organizing our code in a clear and logical manner, we aim to enhance readability, facilitate maintenance, and ensure proper encapsulation of private and public members.

4) Utilizing Vectors in C++: We endeavor to comprehend the versatile functionality of vectors in C++, including creating matrix-like structures, dynamically adding data using the push_back function, and efficiently accessing data elements through looping mechanisms. Additionally, we explore techniques for optimizing data passing, such as passing references instead of duplicating entire datasets.
5) Basic Vector Functions: We familiarize ourselves with essential vector functions such as .size, .back, and .begin, which are indispensable for manipulating vector data structures efficiently and effectively.
6) Integrating Gnuplot for Visualization: By integrating the Gnuplot package into our development environment, we aim to streamline the process of visualizing data and generating graphs directly within our C++ program. This eliminates the need for manual data collection and external tools like Excel, enabling seamless visualization of results.

Overall, our motivation for embarking on this project stems from a desire to gain a comprehensive understanding of neural network fundamentals, enhance our programming skills in C++, and leverage advanced visualization techniques to analyze and interpret neural network behavior effectively.

III. METHODOLOGY

Throughout the development process, we utilized various C++ compilers, including Cygwin and Sublime Text on Windows, as well as Xcode on Mac, to ensure the robustness and compatibility of our code across different platforms. This diversified approach allowed us to rigorously test and validate our results, providing confidence in the reliability of our implementation.

In this section, we delve into the intricacies of our codebase, offering a comprehensive explanation of how to construct a neural network in C++. The code is structured into four main components: Class Training Data, Class Neuron, Class Net (Layer), and the Main Function. Each of these components has its unique and irreplaceable function, playing a crucial role in the architecture and operation of the neural network, as outlined in detail below.

Class Training Data. This pivotal component in our project ensures the generation of varied training datasets, essential for instructing the neural network to demonstrate Exclusive-OR (XOR) behavior. Additionally, it provides flexibility in adjusting the network structure through the "topology" variable, allowing for modifications in the number of hidden layers. Similarly, the variable "i" in the for loop inside the training data generation block allows for customization of the training time, known as epochs. Below, we elaborate on the functions employed within this class:

1) isEndOfFile. This function verifies whether the current index in the training dataset has reached the end of the file. If the current index equals or exceeds the size of the training dataset, it returns true, indicating the absence of further training examples to process.

2) GetTopology. This function retrieves the network topology, storing the number of neurons in each layer within the topology vector. This vector serves as a blueprint for defining the desired neural network structure. Figure 2 shows a neural network architecture defined by the topology function.

3) GetNextInputs. Responsible for retrieving input values for the subsequent training example, this function stores them in the InputValues vector. It returns the number of input values retrieved and returns 0 when the end of the training dataset is reached.

4) GetTargetOutputs. This function retrieves the target output values corresponding to the next training example, storing them in the TargetOutputValues vector. Similarly, it returns the number of target outputs retrieved and returns 0 upon reaching the end of the training dataset.

5) TrainingData. This function generates random training sets, enabling the neural network to learn XOR behavior effectively. It incorporates the usage of the C++ basic library "cstdlib" to leverage the rand() function, facilitating the generation of random training data.

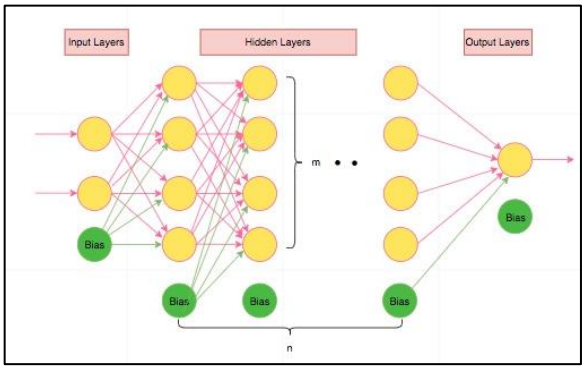


Figure 2. By setting the topology, we can change the neural network structure. For example, setting topology = {2, 4, 4, 4, 1} means that we have the input layer containing 2 neurons, *n* (which is 3) hidden layers with *m* (which is 4) neurons each, and one output layer with 1 neuron. A bias neuron will be added to each layer.

Class Neuron. The functionality of the Neuron class extends beyond merely creating a neuron; it serves as the computational core of the entire neural network. Within this class, not only are neurons created, but crucial computations such as processing input values, calculating output values, and facilitating feedforward operations occur. Additionally, weight computations and activation functions are also integral parts of this class. Exploring the specific functions utilized within the Neuron class:

1) Neuron. The Neuron constructor initializes a neuron with a specified number of output connections and layer index. It constructs connections with random weights and prepares the neuron to receive inputs and perform calculations during neural network operation.

2) SetOutputValue and GetOutputValue. These functions handle the output value of each neuron, with one setting the output value and the other returning it.

3) FeedForward. This function allows neurons to propagate forward by summing the previous layer's outputs, including the bias node, and applying the activation function to shape the output.

4) ComputeOutputGradients. It calculates the error reduction necessary for training by comparing the target value with the actual value and multiplying the difference by the derivative of the output value.

5) ComputeHiddenGradients. Unlike output gradients, this function calculates errors differently, involving the derivative of weights and the derivative of the transfer function output value.

6) sumDOW. This function, which stands for "Summarize Derivative of Weights" (DOW), aggregates error contributions at the nodes fed by the neuron, summing the connection weights from the neuron to others based on their index number.

7) ActivationFunction and ActivationFunctionDerivative. These functions are crucial for backpropagation learning, employing the hyperbolic tangent function and its derivative to produce output within the range of -1 to 1.

8) UpdateInputWeights. Perhaps the most crucial function in this class, it updates weights stored in the connection container of neurons in the preceding layer. It involves a loop through all previous layer neurons, including the bias, and calculates new delta weights based on the learning rate and momentum. The formula for updating weights is: *NewDeltaWeight = eta × neuron.GetOutputValue() × MyGradient + alpha × OldDeltaWeight.* Where eta represents the learning rate, determining the overall training rate, and alpha represents the momentum rate, influencing the previous delta weight to adjust the changing rate for the last training sample. Finally, it sets these new delta weights to update the weights accordingly.

Class Net (Layer). This class serves as the cornerstone of the neural network, coordinating the connections between layers and facilitating feedforward and backpropagation processes. Its primary responsibility lies in constructing the network structure based on the specified topology, organizing neurons into layers, and facilitating communication between them. While the computational heavy lifting is handled by the Class Neuron, Class Net focuses on managing the network's architecture and functionality. Below, we outline the key functions within this class:

1) Net. The constructor function defines the structure of the network by creating layers and neurons according to the specified topology. It iterates through each layer, initializing neurons with the appropriate number of output connections. Additionally, it ensures the inclusion of bias neurons in each layer, essential for network performance.

2) FeedForward. This function orchestrates the feedforward process, where input values are propagated through the network. It assigns input values to input neurons and then iterates through each layer, instructing neurons to calculate their output values based on the inputs received from the previous layer.

3) BackPropagation. This function drives the network's adaptation by updating weights based on error analysis. It starts by calculating the overall net error, employing the Root Mean Square Error (RMS) metric to gauge the network's performance. Subsequently, it implements a mechanism for measuring recent average error, providing insights into the network's training progress. The function then proceeds to compute gradients for both the output and hidden layers, crucial for adjusting connection weights effectively. Finally, it iterates through layers from outputs to the first hidden layer, updating connection weights to refine the network's performance.

4) GetResults. Retrieves the output values of the network without modifying them. This function populates a container with the output values from the output layer neurons, providing access to the network's current output.

5) GetNetRecentError. Calculates the recent error of the network by comparing the output values to the target values and storing the results in a matrix. This information is valuable for evaluating the network's performance and identifying areas for improvement.

6) MyRecentErrors: Stores information about each input's recent errors for each target value. This vector structure facilitates data analysis and computation, allowing for the assessment of the network's learning progress over multiple iterations.

7) GetAverageErrorForTarget. Computes the average error for target values of 0 and 1 over the previous 100 training passes. This function provides insights into the network's performance trends for different target values and contributes to the calculation of overall average error and recent accuracy in the main function.

Main Function. The main function serves as the entry point for the program, where the neural network is invoked and various aspects of results are displayed. It provides visualizations of critical performance metrics such as Net recent error, Average error for Target 0 and 1, Overall average error, and Recent accuracy during each training pass, aiding in the analysis and monitoring of our neural network's progress. To facilitate visualization of these results, we incorporate vectors to store the outcomes during each training pass and utilize Gnuplot, an additional package for the Cygwin C++ compiler, to generate graphs depicting the performance of our neural network after each run. Additionally, we implement functionality for inference within the neural network, enriching our project by encompassing both training and inference capabilities. Below, we elaborate on some of the functions utilized in the main function:

1) Variables. We initialize crucial variables like “trainData” to initiate the Class Training Data, acting as the foundation for constructing the entire neural network. Additionally, variables such as “InputValues”, “TargetValues”, and “ResultValues” aid in facilitating the interconnection of the neural network within the aforementioned class. These variables are printed during each training pass, providing insights into the performance trend of our neural network.

2) Vectors. We utilize multiple vectors to store the outcomes of each training pass, enabling us to visualize the data through Gnuplot scripts. These vectors serve as repositories for various performance metrics, allowing us to track and analyze the neural network's behavior over time.

3) Generate graphs. By iterating through each vector and extracting the stored values, we generate graphs for each result using Gnuplot scripts. This process not only facilitates the visualization of each outcome but also highlights random

performance cases encountered in our project, thereby guiding potential future improvements.

4) Inference. A comprehensive neural network should not only focus on training but also possess the capability to make inferences. To achieve this, we implement a do-while loop to enable inference testing. This functionality allows users to input sets of data, enabling real-time assessment of our neural network's performance beyond the training phase.

IV. Optimization

Optimizing a neural network involves adjusting its parameters to achieve peak performance for a given task. Key factors in optimization include determining the optimal number of epochs and crafting an architecture with appropriate hidden layers. An epoch signifies one complete iteration through the training dataset. Insufficient epochs may lead to underfitting, where the model fails to capture underlying data patterns, while excessive epochs may result in overfitting, where the model memorizes the training data but performs poorly on new data. Optimizing the epoch count involves striking a balance between underfitting and overfitting while monitoring training and validation performance.

The number of hidden layers and neurons per layer significantly influences the model's ability to learn intricate patterns. Increasing the number of hidden layers allows the network to grasp more complex features but heightens the risk of overfitting. Optimizing the architecture entails experimenting with diverse configurations, such as the number of hidden layers and neurons, to achieve the optimal balance between model complexity and generalization. In essence, finding the optimal point for a neural network involves navigating the trade-off between accuracy and training time consumption, achieved through careful experimentation and adjustment of parameters.

V. Results

The result of this project primarily focuses on the process of building a neural network from scratch without utilizing any neural network-related libraries. Figure 3 illustrates the number of neurons created after executing the code. Another significant aspect of our results occurs during the training passes. Within each pass, we present the pass number, representing the training time, alongside a randomly generated set of inputs fed into the neural network. Additionally, we display the output predicted values generated by the neural network and the corresponding target values, which signify the expected outputs for the Exclusive-OR operation.

Furthermore, we provide insights into the neural network's performance by showcasing the Net recent error, indicating the proximity of the output values generated by the neural network to the actual target values. We also present the average error separately for target values equal to 0 and 1. This separation not only allows us to observe the decreasing trend of the average error for each target individually but also enables us to visualize the impact of each target on the overall average error during the training process. Finally, we display the overall average error and the recent accuracy to aid in the visualization of the graph and to gain a deeper understanding of how our neural network performs.

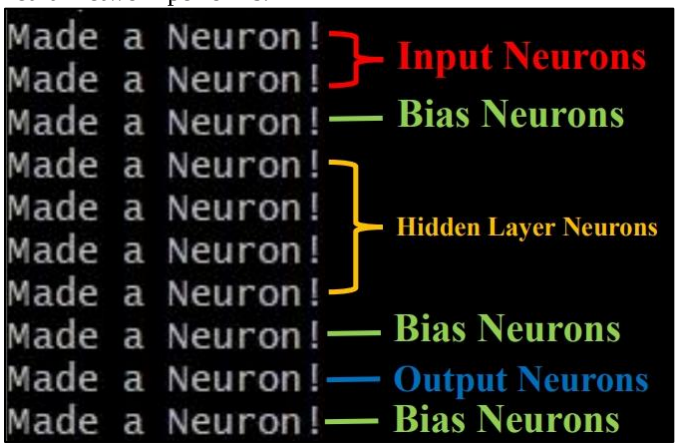


Figure 3. Illustrated our neural network code constructing the NN structure. The topology is set as {2, 4, 1}, resulting in the creation of 2 input neurons, a bias neuron, 4 hidden layer neurons with a bias neuron, and an output neuron with a bias neuron.

A. Performance

Figure 4 illustrates the output of an Exclusive-OR for easy comparison with our results.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4. The output of Exclusive-OR

Training Part:

Same Structure Different Epochs:

Figure 5 demonstrates an example when the topology is set as {2, 4, 1} for the neural network, with all values precise to 8 digits after zero.

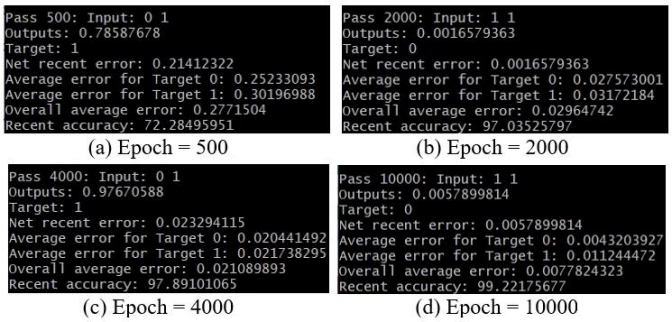
```
Pass 499: Input: 1 0
Outputs: 0.68920571
Target: 1
Net recent error: 0.31079429
Average error for Target 0: 0.2607805
Average error for Target 1: 0.30383896
Overall average error: 0.28230973
Recent accuracy: 71.76902707

Pass 500: Input: 0 1
Outputs: 0.78587678
Target: 1
Net recent error: 0.21412322
Average error for Target 0: 0.25233093
Average error for Target 1: 0.30196988
Overall average error: 0.2771504
Recent accuracy: 72.28495951
```

Figure 5. A result for the neuron network.

Increasing the epoch, i.e., training passes, results in a decrease in the overall average error for the neural network, leading to increased accuracy. Figures 6 to 9 depict a similar trend when we increase the number of hidden layers in our neural network structure, with the topology set as {2, 4, 1}, {2,

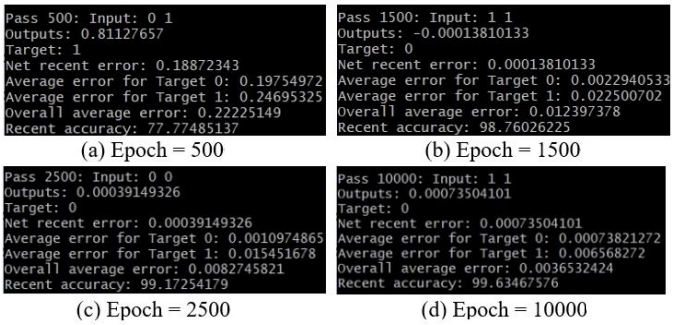
4, 4, 1}, {2, 4, 4, 4, 1}, and {2, 4, 4, 4, 1}, respectively. These figures provide evidence that increasing the epochs in the neural network will increase the accuracy. However, Figures 10 and 11 serve as counter-examples, illustrating that over-extending the hidden layers may lead to the neural network outputting values around 0.5 between 0 and 1, resulting in an accuracy consistently around 0.45 to 0.55.



(a) Epoch = 500 (b) Epoch = 2000

(c) Epoch = 4000 (d) Epoch = 10000

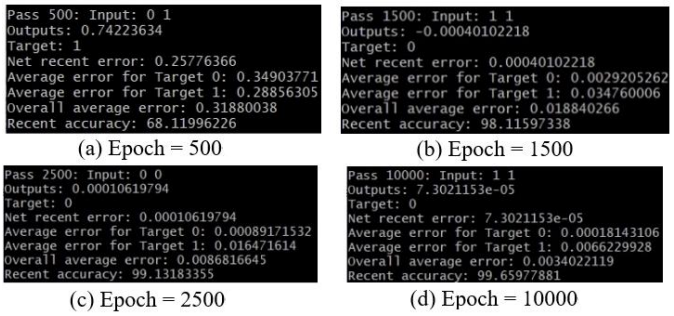
Figure 6. Topology is set as {2, 4, 1} and the result of different epochs.



(a) Epoch = 500 (b) Epoch = 1500

(c) Epoch = 2500 (d) Epoch = 10000

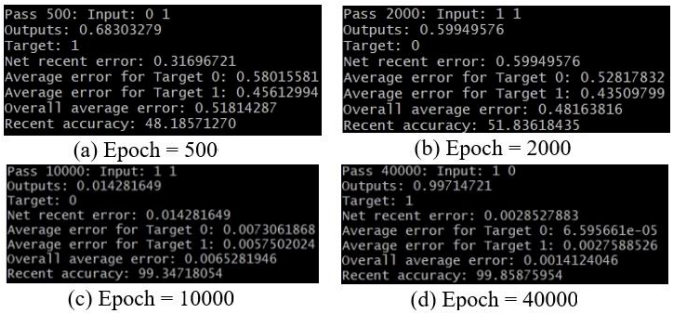
Figure 7. Topology is set as {2, 4, 4, 1} and the result of different epochs.



(a) Epoch = 500 (b) Epoch = 1500

(c) Epoch = 2500 (d) Epoch = 10000

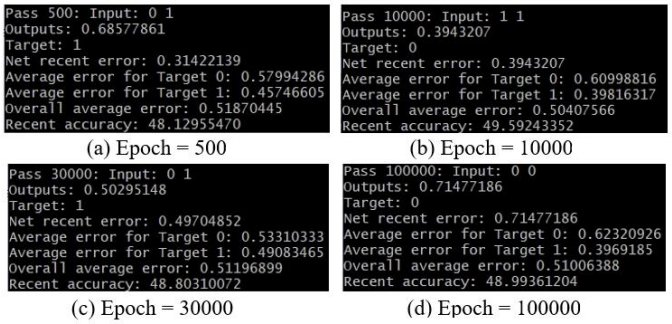
Figure 8. Topology is set as {2, 4, 4, 4, 1} and the result of different epochs.



(a) Epoch = 500 (b) Epoch = 2000

(c) Epoch = 10000 (d) Epoch = 40000

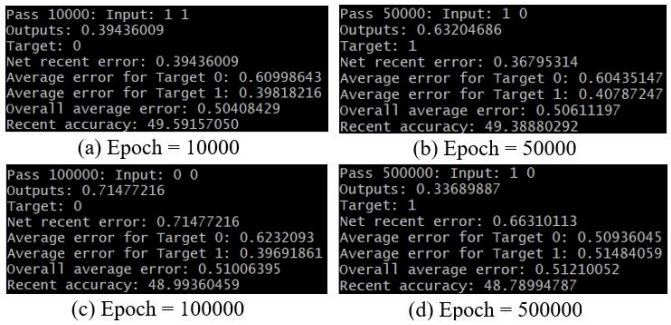
Figure 9. Topology is set as {2, 4, 4, 4, 4, 1} and the result of different epochs.



(a) Epoch = 500 (b) Epoch = 10000

(c) Epoch = 30000 (d) Epoch = 100000

Figure 10. Topology is set as {2, 4, 4, 4, 4, 4, 1} and the result of different epochs.



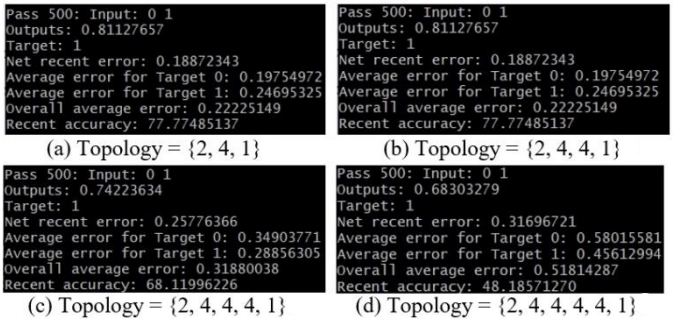
(a) Epoch = 10000 (b) Epoch = 50000

(c) Epoch = 100000 (d) Epoch = 500000

Figure 11. Topology is set as {2, 4, 4, 4, 4, 4, 4, 1} and the result of different epochs.

Same Epoch Different Structure:

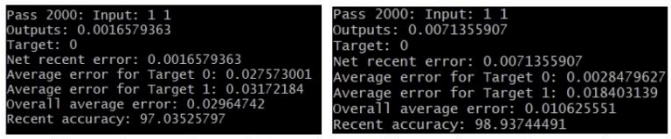
Before obtaining the actual results, we expected that increasing the number of hidden layers in our neural network, due to increased interconnections in the structure, might achieve higher accuracy compared to structures with fewer hidden layers for the same epoch. However, the numerical results indicate that having more hidden layers does not guarantee higher accuracy with the same number of epochs. Instead, each structure exhibits its unique trend in accuracy improvement. Figure 12 to 14 shows the results for the same epoch with different structures. This variability may occur because of the random sets of input and random weights assigned each time the code runs, resulting in different trends of accuracy incrementation.



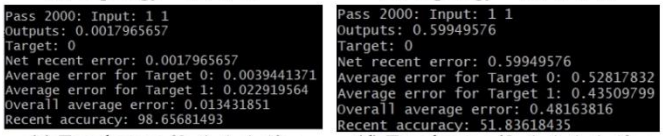
(a) Topology = {2, 4, 1} (b) Topology = {2, 4, 4, 1}

(c) Topology = {2, 4, 4, 4, 1} (d) Topology = {2, 4, 4, 4, 4, 1}

Figure 12. Epoch is set as 500 and the result of different structure.

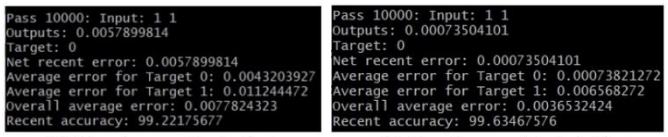


(a) Topology = {2, 4, 1} (b) Topology = {2, 4, 4, 1}

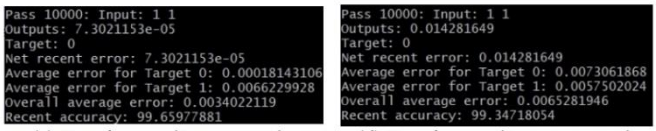


(c) Topology = {2, 4, 4, 4, 1} (d) Topology = {2, 4, 4, 4, 4, 1}

Figure 13. Epoch is set as 2000 and the result of different structure.



(a) Topology = {2, 4, 1} (b) Topology = {2, 4, 4, 1}



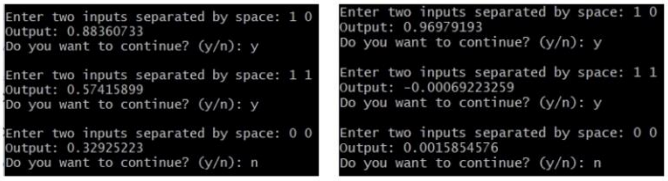
(c) Topology = {2, 4, 4, 4, 1} (d) Topology = {2, 4, 4, 4, 4, 1}

Figure 14. Epoch is set as 10000 and the result of different structure.

Inference Part:

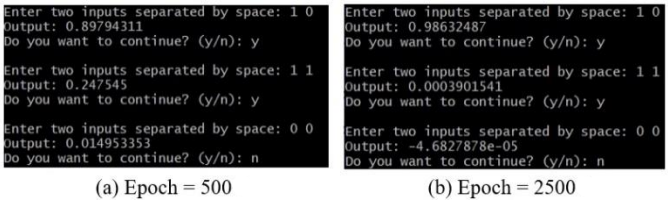
Same Structure Different Epochs:

Following a similar trend to the training results, increasing the epochs in the neural network enhances the accuracy, leading to better performance in inference, i.e., prediction output made by the neural network. Figure 15 to 18 shows the inference results for the same epoch with different structures. The only exception occurs if the selected epoch coincides with the adjustment of results, resulting in poor inference performance at that specific time. This observation emphasizes the importance of reviewing the overall accuracy graph before conducting inference with a neural network.



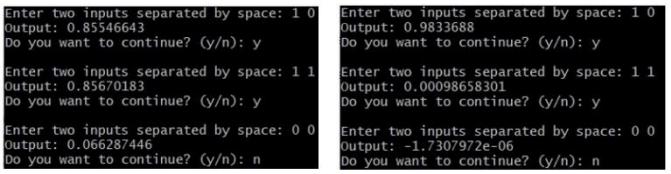
(a) Epoch = 500 (b) Epoch = 2000

Figure 15. Topology is set as {2, 4, 1} and the result of different epochs.



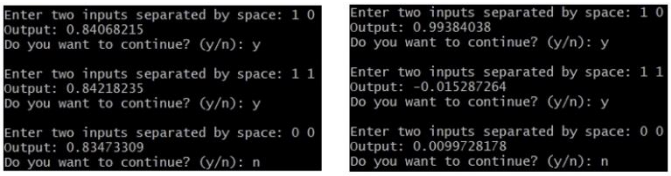
(a) Epoch = 500 (b) Epoch = 2500

Figure 16. Topology is set as {2, 4, 4, 1} and the result of different epochs.



(a) Epoch = 500 (b) Epoch = 2500

Figure 17. Topology is set as {2, 4, 4, 4, 1} and the result of different epochs.

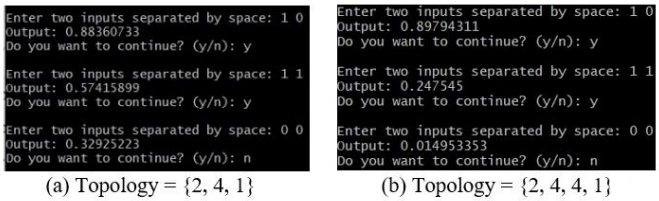


(a) Epoch = 500 (b) Epoch = 10000

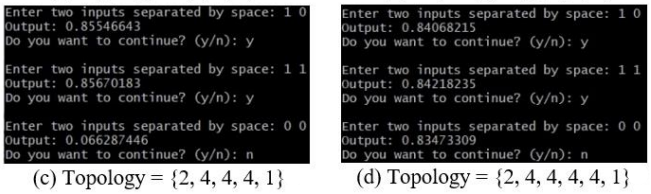
Figure 18. Topology is set as {2, 4, 4, 4, 4, 1} and the result of different epochs.

Same Epoch Different Structure:

Figure 19 to 21 shows the inference result for the same structure but using different epochs. Generally, we can observe that the neural network cannot understand what to do when the epoch is small, around 500 epochs, for example. If we set a larger epoch, 2000 for instance, most of the neural networks already understand what the output value should be. The only exception is the Topology = {2, 4, 4, 4, 4, 1} configuration; for epoch = 2000, this neural network has not yet figured out what to do. This is also linked back to what we mentioned earlier: each neural network structure has its unique time for accuracy improvement. Finally, when we set the epoch to be 10000, all the neural networks give accurate prediction outputs, and as expected, the more hidden layers the neural network has, the higher accuracy it reaches.

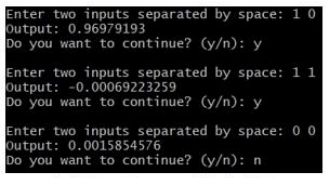


(a) Topology = {2, 4, 1} (b) Topology = {2, 4, 4, 1}

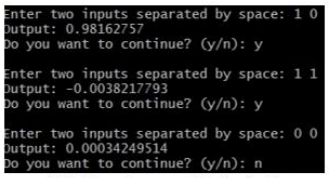


(c) Topology = {2, 4, 4, 4, 1} (d) Topology = {2, 4, 4, 4, 4, 1}

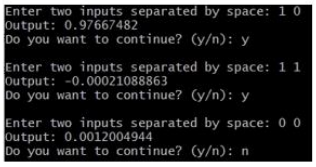
Figure 19. Epoch is set as 500 and the result of different structure.



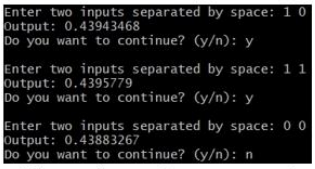
(a) Topology = {2, 4, 1}



(b) Topology = {2, 4, 4, 1}

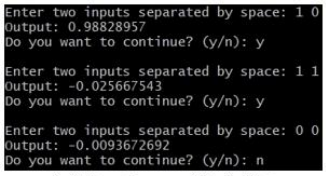


(c) Topology = {2, 4, 4, 4, 1}

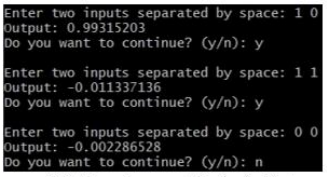


(d) Topology = {2, 4, 4, 4, 4, 1}

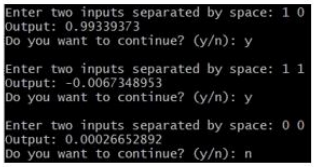
Figure 20. Epoch is set as 2000 and the result of different structure.



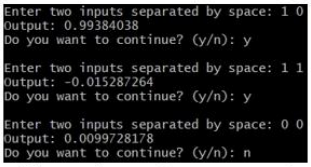
(a) Topology = {2, 4, 1}



(b) Topology = {2, 4, 4, 1}



(c) Topology = {2, 4, 4, 4, 1}



(d) Topology = {2, 4, 4, 4, 4, 1}

Figure 21. Epoch is set as 10000 and the result of different structure.

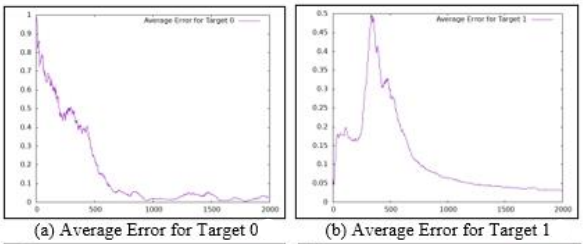
B. Trend

By incorporating an extended package called Gnuplot into the Cygwin C++ compiler, we can visualize the trend of the overall results by plotting graphs generated during the training process.

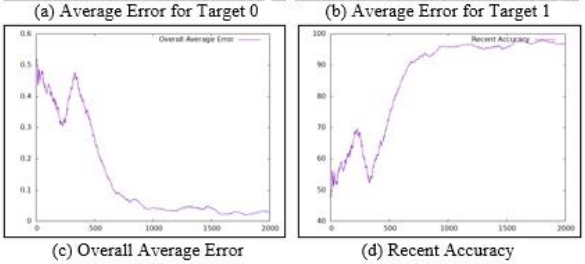
Firstly, it is evident that the average error for target 0 and 1 displays distinct trends. This variance may arise from the diverse sets of input fed into the neural network. Figures 22 and 23 depict two different neural network structures along with their respective average errors for target = 0, average errors for target = 1, and overall average errors. As illustrated in the graphs, they exhibit varying trends for target 0 and 1. However, after a certain number of epochs, the neural network consistently learns and adapts, leading to a decline in the overall average error and an increase in accuracy.

Moreover, it is noteworthy that while increasing the number of hidden layers can enhance accuracy, it also extends the time required for the neural network to comprehend the task. Figure 24 exemplifies this effect, showing that structures with more hidden layers require an extended duration of training to achieve accuracy improvements. Additionally, excessively adding hidden layers may render the neural network unable to discern the appropriate action, even with increased epochs. Figure 25 presents an instance of this scenario, depicting two structures with excessive hidden layers and high epoch settings.

Furthermore, for neural networks with identical structures, there are still some trends that elude explanation within the confines of our current understanding. Despite employing the same structure for training, the accuracy does not uniformly increase over time, as observed in our graphical results. One plausible explanation for this phenomenon could be the impact of changing epochs, which alters the input sets fed into the neural network, thereby affecting the timing of error reduction. Figure 26 provides an illustration of this scenario.

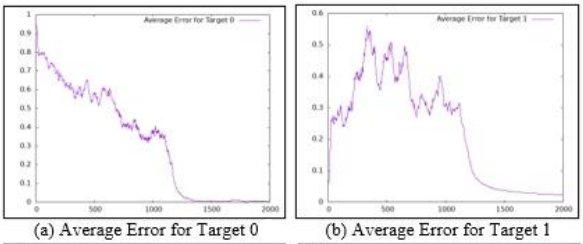


(a) Average Error for Target 0 (b) Average Error for Target 1

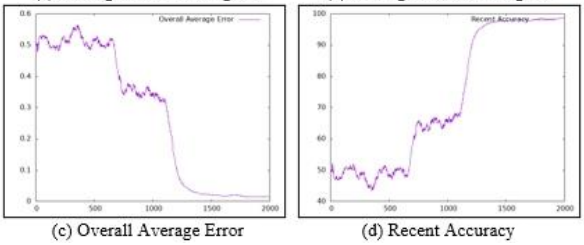


(c) Overall Average Error (d) Recent Accuracy

Figure 22. Topology = {2, 4, 1} and the graphical results for epochs = 2000.

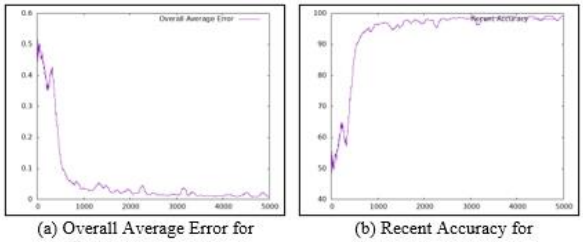


(a) Average Error for Target 0 (b) Average Error for Target 1

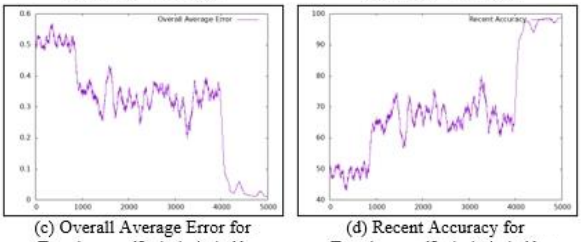


(c) Overall Average Error (d) Recent Accuracy

Figure 23. Topology = {2, 4, 4, 4, 1} and the graphical results for epochs = 2000.



(a) Overall Average Error for Topology = {2, 4, 1} (b) Recent Accuracy for Topology = {2, 4, 1}



(c) Overall Average Error for Topology = {2, 4, 4, 4, 4, 1} (d) Recent Accuracy for Topology = {2, 4, 4, 4, 4, 1}

Figure 24. The graphical results show that the more complex the neural network structure is, the more training time it spends increasing accuracy.

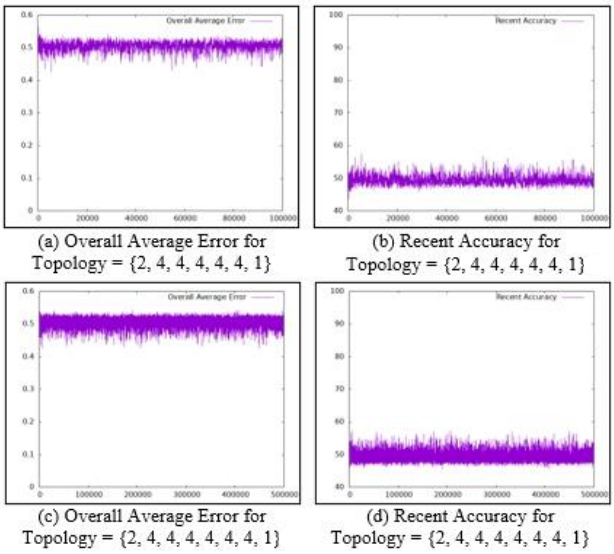


Figure 25. The performance for two structures with excessive hidden layers and high epoch settings. The accuracy did not increased even when we set a huge epoch.

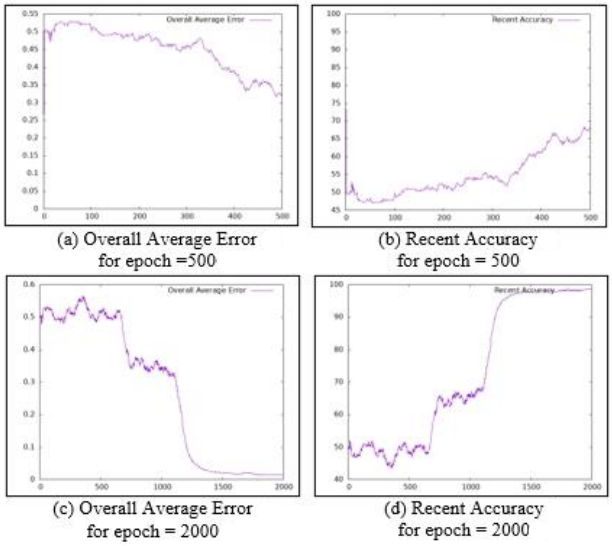


Figure 26. The graphical results for topology {2, 4, 4, 4, 1} with epochs set to 500 and 2000. As depicted in the graphs, with epoch = 500, the accuracy already reaches around 65 percent. However, when epoch is set to 2000, the accuracy observed at epoch = 500 struggles at 50 percent.

C. Pitfall

1. Random cases occur in our code, the basis of which remains unknown as of the time of this report. Possible solutions to this problem include feeding the exact same sets of input and weights for each neuron. These are potential areas for future work to extend this paper and make it more comprehensive.

2. Even when using the same structure to train our neural network, the result will always be the same each time we rerun the code for the same epoch. However, for different epochs, it does not follow the same trend as with lower epochs, as depicted in the figures. The overall average accuracy does not drop simultaneously with the lower epochs; rather, each epoch has its unique dropping time. This phenomenon may also be related to the random sets of input and weights assigned to each neuron.

3. We did not include a part of the code to show the runtime for our training process. However, we observed that it took more time when we set the epoch to a larger value. Including a clock to track the exact time would be necessary if we aim to produce a more comprehensive report in the future.

VI. Future Work

Our exploration into neural network development has unearthed several intriguing avenues for future research and enhancement. Moving forward, we aim to address key limitations and delve deeper into the intricacies of neural network optimization. Here, we outline potential areas for future investigation:

1. Standardizing Input and Weights. The occurrence of random cases in our code poses a significant challenge, with the underlying causes remaining elusive. To overcome this hurdle, we plan to explore methods for standardizing input sets and weights for each neuron. By reducing variability, we can improve the consistency and reliability of our results, laying a foundation for more robust neural network models.

2. Exploring Epoch-Accuracy Dynamics. The complex relationship between the number of epochs and model accuracy warrants further exploration. While higher epochs are often expected to yield superior performance, our findings reveal nuanced trends influenced by various factors. We intend to conduct comprehensive analyses to better understand these dynamics, paving the way for more effective training strategies and enhanced model performance.

3. Measuring Runtime Efficiency. Incorporating runtime analysis into our methodology holds promise for optimizing the computational efficiency of our neural network. By tracking training duration across different epoch settings, we can identify opportunities to streamline processes and optimize resource utilization. This endeavor will contribute to the development of more efficient and scalable neural network architectures, facilitating broader applications in various domains.

VII. Conclusion

In this project, we have embarked on a comprehensive exploration of neural network development in C++ from the ground up. Our journey has been guided by five primary objectives:

1. Implementing Neural Network in C++. Our endeavor to construct a neural network entirely in C++ has provided valuable insights into the fundamental principles of neural network architecture and operation. By coding each component from scratch, we have gained a deeper appreciation for the underlying mechanisms driving neural network behavior.

2. Exploring C++ Techniques and Functions. Through the implementation process, we have familiarized ourselves with a myriad of techniques and functions available in C++. From data structures to algorithms, we have leveraged the versatility of C++ to create a robust and efficient neural network framework.

3. Visualizing Performance through Graphs. An integral aspect of our project has been the visualization of neural network performance using C++ packages. By plotting result

graphs, we have been able to discern trends, patterns, and anomalies in our data, providing valuable insights into the behavior and efficacy of our neural network model.

4. Insights into Training Dynamics. Our analysis has provided valuable insights into the dynamics of neural network training. By exploring the relationship between training parameters such as epochs and hidden layers, we have gained a deeper understanding of how these factors influence model performance.

5. Implications for Real-world Applications. The knowledge and insights gained from our project have far-reaching implications for real-world applications of neural networks. From pattern recognition to predictive analytics, the techniques and methodologies developed in this report can be applied to a wide range of domains, driving innovation and advancing the field of artificial intelligence.

In conclusion, our exploration of neural network development in C++ has been both illuminating and rewarding. By achieving our objectives and uncovering valuable insights along the way, we have laid the foundation for future research and innovation in this exciting field. As we deepen our understanding of neural network principles and techniques, we are better equipped to explore further applications and opportunities in artificial intelligence. This project serves as a stepping stone towards leveraging neural networks for practical solutions and driving advancements in various domains.

VIII. REFERENCE

[1] Coding a Neural Network from Scratch in C: No Libraries Required https://www.youtube.com/watch?v=LA4I3cWkp1E&ab_channel=NicolaiNielsen

[2] Artificial Neural Network From Scratch in C++ https://www.youtube.com/watch?v=k_VdZVJeEyg&ab_channel=CodingMan

[3] Neural Net implementation in C++ https://www.youtube.com/watch?v=sK9AbJ4P8ao&ab_channel=AbhishekPandey

[4] New Video Tutorial: Make a Neural Net Simulator in C++ https://millermattson.com/dave/?p=54

[5] ML – Neural Network Implementation in C++ From Scratch https://www.geeksforgeeks.org/ml-neural-network-implementation-in-c-from-scratch/

[6] Neural Network in C++ From Scratch and Backprop-Free Optimizers https://hyugen-ai.medium.com/neural-network-in-c-from-scratch-and-backprop-free-optimizer-d2a34bb92688

[7] GNUPLOT 4.2 - A Brief Manual and Tutorial, Department of Civil and Environmental Engineering, Edmund T. Pratt School of Engineering, Duke University - Box 90287, Durham, NC 27708-0287 https://people.duke.edu/~hpgavin/gnuplot.html

[8] Using gnuplot from a C++ Program https://warwick.ac.uk/fac/sci/moac/people/students/peter_cock/cygwin/part6/