Logic Learning and Gradient Flow in Neural Networks

Danila Kokin & Haonan Jin & Hang Yao

June 16, 2025

Abstract

This report investigates the foundational principles of neural networks by focusing on two key areas: logic learning and gradient flow visualization. A unified neural network architecture is developed to learn basic logical operations such as AND, OR, and XOR, offering a transparent framework to examine how neural networks handle both linearly and non-linearly separable problems. The study also introduces techniques for visualizing gradient flow to identify potential issues such as vanishing or exploding gradients. All experiments are conducted using the backpropagation algorithm, which enables parameter updates based on gradient information. This integrated approach provides practical insights into the internal mechanics of neural learning and contributes to a more intuitive understanding of how neural networks generalize.

1 Introduction

Artificial Neural Networks (ANNs) have become a cornerstone of modern Machine Learning due to their ability to model complex, non-linear relationships. At the heart of ANN training is the **backpropagation** algorithm, a technique formally introduced and popularized in the 1980s, although its mathematical foundations were laid as early as 1962 by **Frank Rosenblatt** [1]. Backpropagation utilizes the **chain rule** of calculus to efficiently compute gradients of the loss function with respect to each network parameter, enabling effective learning through gradient descent.

The purpose of this project is to explore the internal mechanisms of neural networks by training them to **learn basic logic functions** such as AND, OR, and OR. While these tasks are simple, they effectively illustrate fundamental principles of neural computation—particularly the network's ability to handle **non-linearly** separable problems, as exemplified by the XOR function. In addition, the project introduces **gradient flow visualization** techniques to diagnose common training issues such as **vanishing** and **exploding** gradients.

By combining theoretical foundations with practical implementation and visualization, this project offers a clearer understanding of how neural networks learn and how gradient signals influence training dynamics. This foundation serves as a stepping stone for more advanced studies in neural network architectures.

2 Literature Review

Modern backpropagation was first introduced by **Seppo Linnainmaa** in 1970 as the reverse mode of automatic differentiation for evaluating gradients in computational graphs composed of nested differentiable functions. Later, in 1982, **Paul Werbos** formally applied backpropagation to train **MLPs**, laying the foundation for the algorithm's standard usage in modern neural networks [2].

Once the project context is established, it becomes necessary to introduce several critical foundational concepts for understanding how neural networks learn.

- **Backpropagation:** After the forward pass of the network produces a prediction for each epoch, backpropagation is used to compute the gradients of the loss function with respect to each model parameter. These gradients indicate how the parameters should be adjusted to minimize the loss and are essential for the learning process.
- **Chain Rule:** The backpropagation algorithm relies on the chain rule from calculus to compute gradients layer by layer. For example, the derivative of the loss L with respect to a weight w is given by:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

  where a is the activation function and z is the weighted input to a neuron (e.g., $z = W^T x + b$). This decomposition enables efficient gradient computation across multiple layers.
- **Activation Function:** An activation function is applied to the output of each neuron to introduce non-linearity into the network. Without non-linearity, a neural network—regardless of its depth—would be equivalent to a linear model [3].
- **Optimizer:** An optimizer is an algorithm that updates the weights of a neural network using the gradients calculated during backpropagation. Its goal is to minimize the loss function by adjusting the model parameters efficiently [4].
- **Vanishing/Exploding Gradients:** The *vanishing gradient problem* occurs when gradients become small as they are propagated backward through layers, causing early layers to learn very slowly or not at all. In contrast, the *exploding gradient problem* arises when gradients grow excessively large, leading to unstable updates and divergence during training [5].

3 Theoretical Foundations

In this section, we introduce the key theoretical components used to construct the neural network, focusing on the activation functions and the optimizer.

3.1 Activation Functions

We specifically use `ReLU` and `Sigmoid` as non-linear activation functions in the neural network architecture, where the input to the activation function—referred as the *net input*—is given by $W^T \cdot x + b$.

Sigmoid Activation. The `Sigmoid` function is characterized by an **S-shaped** curve and is mathematically defined as:

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

This function produces a smooth and continuous output, mapping any real-valued input to the range $[0, 1]$.

- Enables the network to capture non-linear patterns.
- Suitable for output layers in binary classification due to its probabilistic output.

ReLU Activation. The Rectified Linear Unit (`ReLU`) is defined as:

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

This means `ReLU` outputs the input directly if it is positive; otherwise, it outputs zero.

- Outputs values in the range $[0, \infty)$, helping to mitigate the vanishing gradient problem.
- Introduces non-linearity while being computationally efficient.
- Encourages sparsity and faster convergence during training.

3.2 Optimizer: Adam

The `Adam` optimizer combines the benefits of `Momentum` (which accelerates convergence) and `RMSProp` (which adapts the learning rate based on past gradients). It provides a smoother update path, adaptive learning rates, and typically achieves faster and more stable convergence.

`Adam` updates the weights using the following equations:

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) \cdot dW$$
$$g_t = \beta_2 g_{t-1} + (1 - \beta_2) \cdot dW^2$$

$$W = W - \frac{\alpha}{\sqrt{g_t + \epsilon}} \cdot v_t$$

Where:

- v_t is the exponentially weighted average of past gradients (`Momentum`).
- g_t is the exponentially weighted average of the squared gradients (`RMSProp`).
- β_1 and β_2 are hyperparameters controlling the decay rates.
- α is the learning rate.
- ϵ is a small constant to avoid dividing by 0.

With these foundational components—non-linear activation functions and the `Adam` optimizer—we are prepared to construct a neural network architecture capable of efficiently learning complex patterns from data.

4 Experiments

4.1 Logic Learning

Logic functions like AND, OR, and XOR can be wrapped as binary classification problems with two binary inputs. The AND ($A \cdot B$) and OR ($A + B$) are linearly separable. This means a single line (a linear decision boundary) can be drawn in the 2D feature space to separate the two output classes. For instance, a simple model like a logistic regression could solve these tasks.

However, the XOR function is non-linearly separable. Defined as $A \oplus B$, it outputs 1 only when the inputs are different [6]. In this case, the data points (0,0) and (1,1) belong to one class (output 0), while (1,0) and (0,1) belong to the other class (output 1). No single line can separate these points without misclassifying at least one, making a simple linear model insufficient.

To address the non-linearity of the XOR function, a neural network with at least one hidden layer is required. The proposed architecture, a simple design highlighting the core concept, is designed to solve this problem effectively, as illustrated in Figure 1. This figure presents a network graph generated using `NetworkX` [7].

- **Input Layer**: Consists of 2 neurons, corresponding to the two binary inputs.
- **Hidden Layer:** This layer contains 2 neurons and uses the `ReLU` activation function, which outputs values in the range $[0, +\infty)$. Each neuron learns a different linear pattern, and together they can form a non-linear decision boundary.
- **Output Layer:** This layer has a single neuron with a `Sigmoid` activation function, which maps the output to a value between 0 and 1.

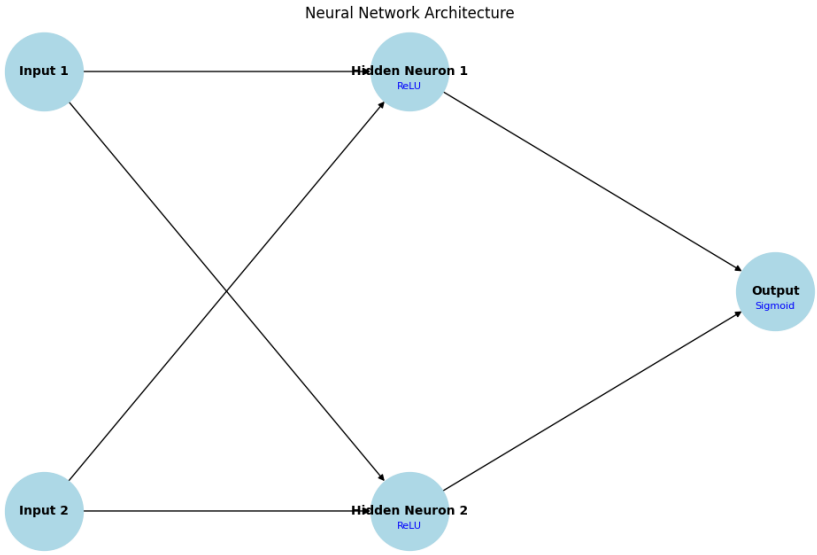


Figure 1: Logic Neural Network Architecture

We have designed a **synthetic dataset** with 10000 samples to simulate logical operations. Each sample consists of two randomly generated input features in the range $[-5, 5]$. A **threshold** of 0 is applied to each feature to determine the binary output labels. For example, if both input values are greater than 0, the output is 1 for AND and OR, and 0 for XOR.

The `Adam` optimizer is used for training, based on the theoretical justification provided earlier. The training process runs for 100 epochs to update the parameters, aiming for optimal accuracy with minimal loss. We balance the class weights to prevent the model from being biased towards the majority class during the training.

4.2 Gradient Flow

To ensure that our network remains trainable—and does not suffer from vanishing or exploding gradients—we instrumented the training loop of the XOR model to capture, record, and visualize the magnitude of the backpropagated signals. Specifically, at the end of each of the 100 epochs, we compute the **Frobenius** norm ($||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$) of the final-batch gradients for both the first-layer weight matrix (`fc1`) and the second-layer weight matrix (`fc2`). These two scalar values summarize the overall strength of the gradient in each layer.

Our Jupyter notebook, `gradient_flow.ipynb`, automates this process. It uses `TensorFlow` for training and `NetworkX` plus `Matplotlib` for visualization. After each epoch's gradients are computed, the notebook (1) updates a two-panel `Matplotlib` figure in memory and (2) appends the norms to time-series arrays. Once training finishes (or the loss falls below a threshold), the notebook stitches these frames into an `MP4` animation that simultaneously shows:

- A network graph in which every edge's color intensity and overlaid label directly encode the absolute gradient on that connection.
- A log-scale line plot of the per-epoch norms for `fc1` (blue) and `fc2` (orange), with exactly one point per epoch.

This dual view makes it easy to see not only how large the gradients are on average, but also where in the network they concentrate and how they fluctuate over time.

5 Results

5.1 Logic Learning

After the training process, we evaluate the model's performance using the test set (20% of the 10000 samples) from **different perspectives**, ensuring that the model learns meaningful patterns.

5.1.1 Accuracy

As illustrated in Table 1, the logic function models do not exhibit overfitting, as indicated by the similar training and test accuracies and losses. For both the AND and OR logic functions, the accuracies are extremely high (greater than 0.99), demonstrating the neural network's capacity to effectively capture linear patterns. However, for the XOR function, the test accuracy is only 0.836, indicating that approximately 16.4% of samples are misclassified. This highlights a limitation of the proposed simple architecture in learning non-linear patterns. A more complex architecture with additional hidden layers and neurons might be necessary perfect classification of XOR.

`Logic Function`	`Train Accuracy`	`Train Loss`	`Test Accuracy`	`Test Loss`
AND	0.997	0.005	0.998	0.004
OR	0.994	0.012	0.991	0.016
XOR	0.828	0.365	0.836	0.358

Table 1: Logic Function Results

5.1.2 Learned Weights and Biases

We now analyze the learned weights and biases for each of the logic function models, examining the correctness of these parameters in achieving the desired logical operations.

AND Function For the AND function (as shown in Figure 2), the network uses its hidden layer to implement NOT logic. `Hidden Neuron 1` acts as a "NOT Input 2" detector, with its output value tending to be positive when `Input 2` is negative, due to the strongly negative weight from `Input 2` to `Hidden Neuron 1` (-9.26) and a nearly negligible weight

from `Input 1` (0.06). Similarly, `Hidden Neuron 2` behaves as a "NOT Input 1" detector, with its output value tending to be positive when `Input 1` is negative, also because of the significant negative weight from that input (−10.97).

The output layer combines these signals with negative weights (−4.70 and −4.47 respectively), but it has a strong positive bias (6.65). This setup means that only when both inputs are positive (i.e., `Input 1` and `Input 2` are both above 0), both "NOT" hidden neurons will produce negative output values. These negative values, when multiplied by the negative hidden-to-output weights, result in a positive contribution to the output neuron. This positive contribution, combined with the output layer's positive bias, then pushes the final result to a '1'.

Conversely, if either `Input 1` or `Input 2` is negative, the corresponding "NOT" hidden neuron will produce a strong positive output. This positive output, when multiplied by its negative weight at the output layer, applies a strong negative influence that overcomes the positive bias, forcing the final result toward '0'. This sophisticated interplay of weights and biases effectively and correctly learns the AND logic.

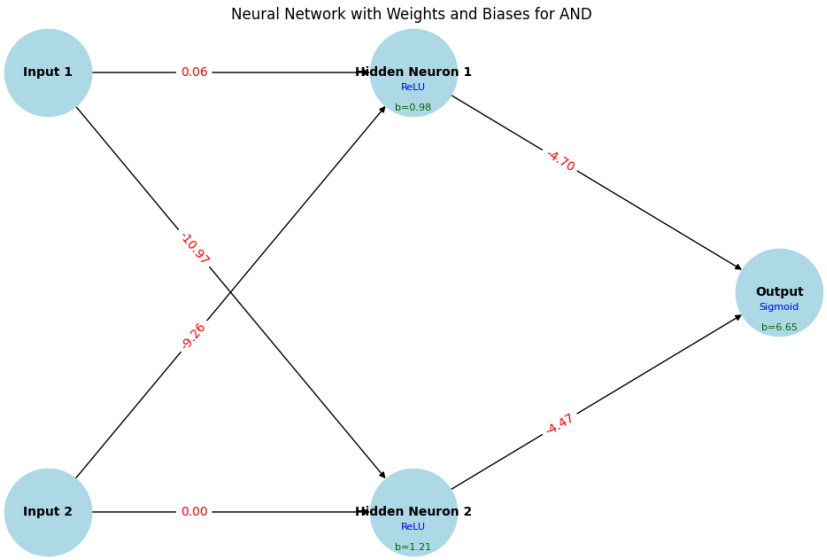


Figure 2: Learned Weights and Biases for AND function

OR Function For the OR function (Figure 3), `Hidden Neuron 1` is strongly influenced by `Input 1`, with a large positive weight of 12.42, while `Input 2` has a negligible weight of 0.00. Combined with its small positive bias of 0.23, this neuron's output value will be significantly positive when `Input 1` is positive (above 0). Conversely, `Hidden Neuron 2` has negative weights from both inputs (−1.93 from `Input 1` and −3.35 from `Input 2`) and a positive bias of 2.06. This neuron will tend to produce a positive output if both inputs are sufficiently negative. The output neuron has a strong positive weight from `Hidden Neuron 1` (5.29) and a negative weight from `Hidden Neuron 2` (−3.13). Crucially, it also has a large positive bias (5.43).

Combining these effects, we can conclude that the output will be positive (close to 1) if at least one of the inputs is positive. Specifically, if `Input 1` is positive, `Hidden Neuron 1`

will produce a strong positive output. Its strong positive connection (5.29) to the output neuron, combined with the output layer's positive bias, will be sufficient to push the final output to "1". If `Input 2` is positive (and `Input 1` is negative), the direct influence from `Hidden Neuron 1` is minimal. However, the overall effect, primarily driven by the strong positive bias of the output layer and potentially a non-suppressive or slightly positive output from `Hidden Neuron 2`, will likely still lead to a '1'. This configuration effectively captures the "at least one positive input" condition required for the OR operation.

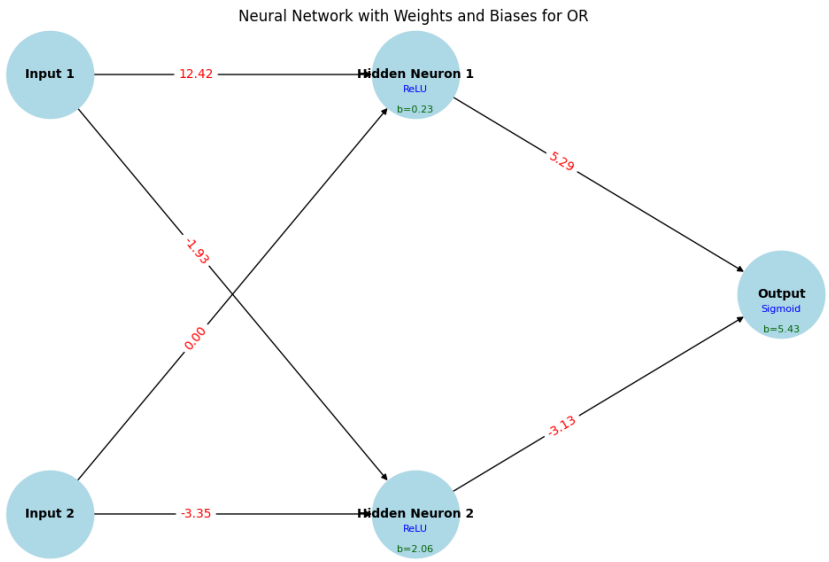


Figure 3: Learned Weights and Biases for OR function

XOR Function For the XOR function (Figure 4), `Hidden Neuron 1` is configured to produce a more positive output when `Input 2` is positive (with a weight of 1.09) and `Input 1` is negative (with a weight of -1.05), aligning with the (0,1) input case.

Similarly, `Hidden Neuron 2` shows almost identical weight patterns, also yielding more positive outputs for the (0,1) input case. This strong similarity between the hidden neurons means they are learning redundant features, which significantly contributes to the network's suboptimal performance. This redundancy is problematic because XOR requires distinct feature extraction by hidden neurons to transform the problem into a linearly separable space.

At the output layer, despite a positive bias (2.78) attempting to push the result towards '1', the network struggles to correctly classify all four XOR input combinations, partly due to the very similar output weights (-1.11 and -1.10) from the two redundant hidden neurons. This struggle arises because the hidden layer's redundant feature extraction prevents it from creating truly distinct representations for the different input patterns. Consequently, the network misclassifies some cases, limiting its accuracy to around 83.6% and demonstrating the current architecture's limitations in fully capturing XOR's non-linear nature.

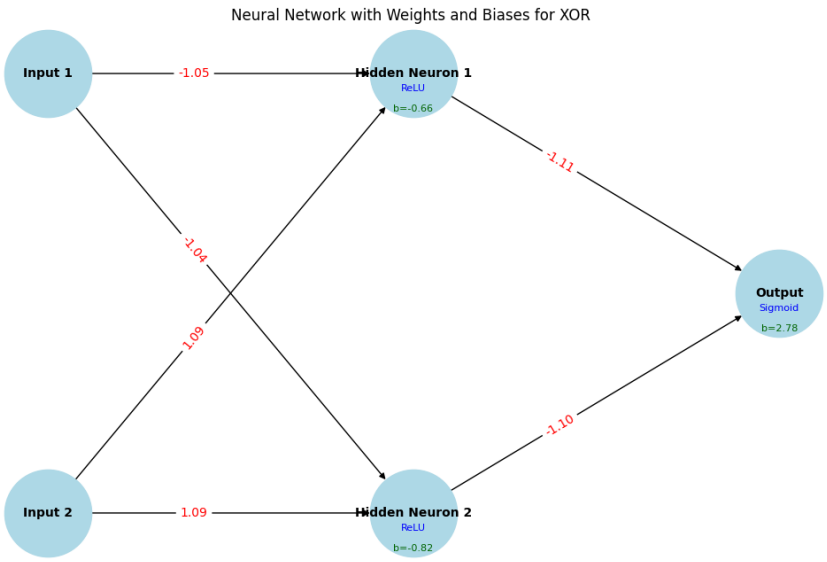
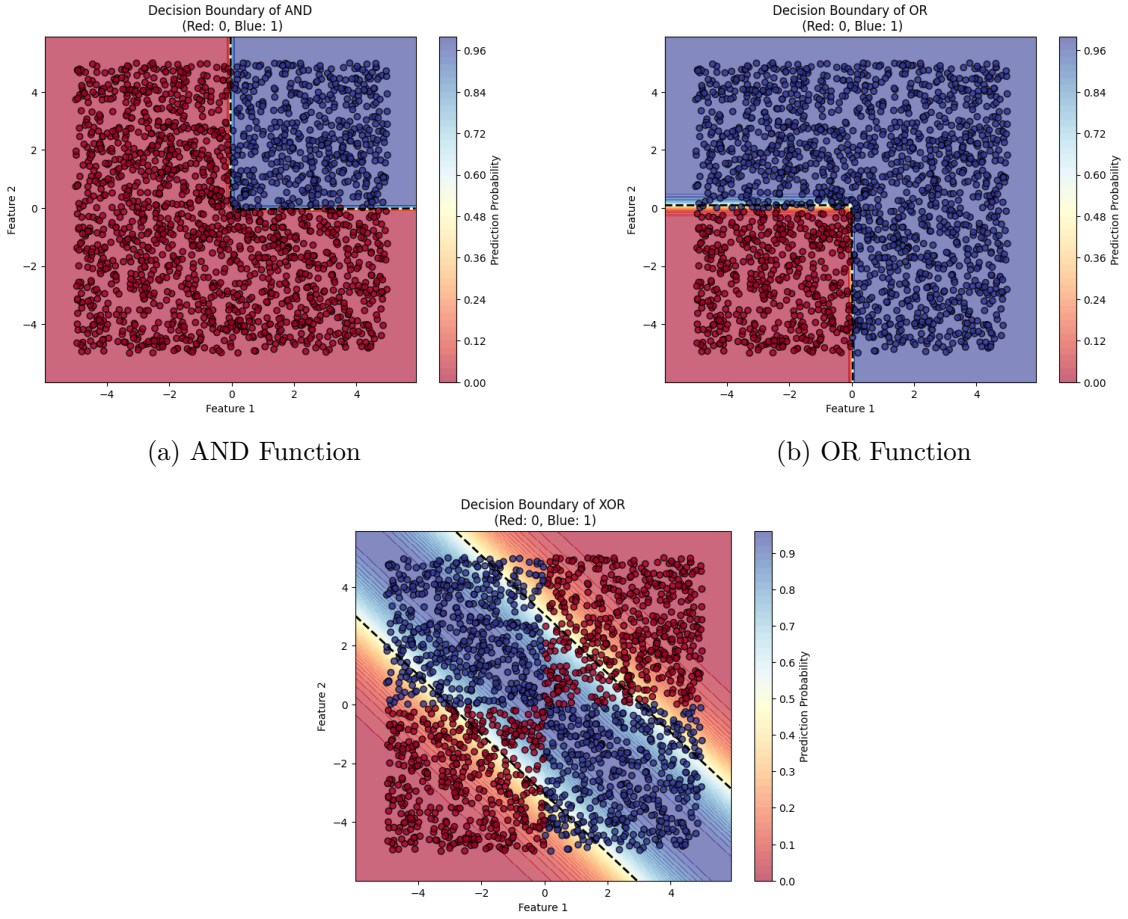


Figure 4: Learned Weights and Biases for XOR function

5.1.3 Decision Boundary

As illustrated in Figure 5, the AND and OR functions are classified with high accuracy. Their decision boundaries are clearly defined along the axes at coordinates (0,0), reflecting the threshold of 0 applied to the input features. For the AND function, only the quadrant where both features have values greater than 0 is classified as Class 1. Oppositely, for the OR function, Class 0 is confined to the quadrant where both features are less than 0, with the remaining three quadrants of the feature space classified as Class 1. These clear delineations demonstrate the network's capacity to learn simple, linearly separable patterns.

However, for the XOR function, we observe two distinct boundary lines that separate the two classes, demonstrating the network's attempt to model a non-linear relationship. While a majority of the points are correctly predicted, a notable portion of samples are still misclassified, often with prediction probabilities close to the decision threshold (0.5). This arises from the limitations of the proposed architecture in fully capturing all the non-linear patterns inherent to XOR, as previously discussed regarding the redundancy in the hidden layer's feature extraction. The network struggles to distinguish perfectly between certain input combinations, contributing to its lower overall accuracy compared to the linearly separable functions.



(a) AND Function

(b) OR Function

(c) XOR Function

Figure 5: Decision Boundaries for Logic Functions

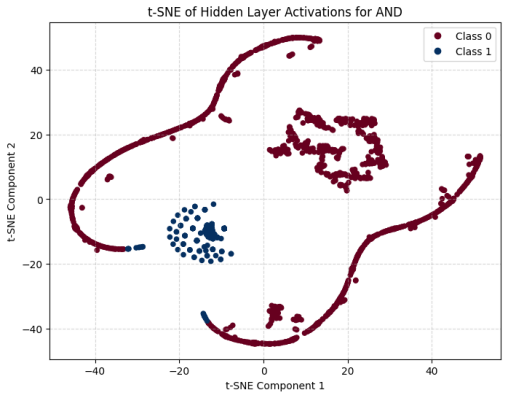
5.1.4 Hidden Layer Activations

We used `t-SNE` [8] to visualize the high-dimensional hidden layer activations. Similar to **Principal Component Analysis** (PCA), `t-SNE` reduces the dimensionality, making it easier to analyze the distinct behaviors of the logic functions. The 2D representation of these activation values—derived from the weight vectors, input feature vectors, and biases—offers insights into how the hidden layer transforms the data.

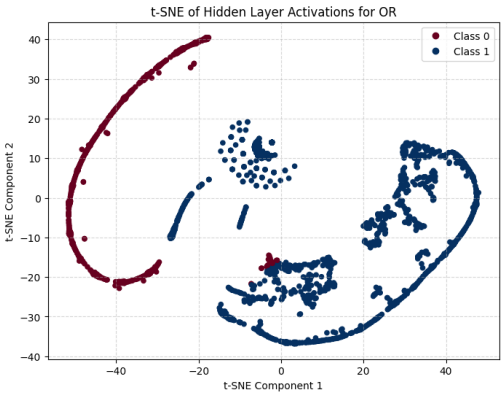
As illustrated in Figure 6, the `t-SNE` results closely match what we observed earlier. For the AND function, points within each class are clearly clustered together, indicating strong separation. Only a few Class 1 points appear to be slightly misclassified, consistent with its high accuracy. The OR function shows a similar pattern, where most of the points within each class are well-grouped, with only some `Class 0` points being misclassified.

In contrast, the XOR function's `t-SNE` plot reveals that while a portion of the data forms distinct clusters, a significant amount of the remaining data points from both classes heavily overlap. This mixing of points means that a clear separation between the classes is not achieved in the hidden layer's activation space, directly contributing to the lower

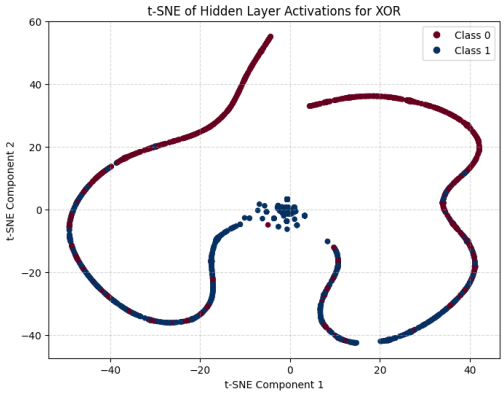
classification accuracy observed for XOR.



(a) AND Function



(b) OR Function



(c) XOR Function

Figure 6: Hidden Layer Activations for Logic Functions

5.2 Gradient Flow

Figure 7 shows the state of our gradient-flow visualization at the final epoch (100). On the left, each edge in the XOR network is labeled with its last-epoch absolute gradient and colored accordingly, revealing that gradient magnitudes remain moderate across all connections. On the right, the solid lines track the per-epoch **Frobenius** norms of `fc1` (blue) and `fc2` (orange) on a logarithmic scale.

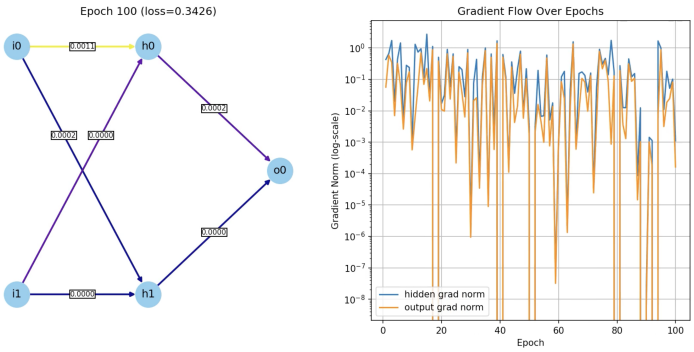


Figure 7: Gradient-flow at epoch 100. *Left:* Network graph showing each edge's absolute gradient value and color-map intensity. *Right:* Log-scale plot of the Frobenius norms of `fc1` (blue) and `fc2` (orange) over 100 epochs.

Although the curves exhibit high-frequency spikes—an expected consequence of training on small minibatches—their overall trajectories stay roughly horizontal, fluctuating between 10^{-1} and 10^{0}. This indicates that, on average, gradients neither decay toward zero (vanishing) nor blow up to extremely large values (exploding). In practice, such stability guarantees that parameter updates remain well-scaled throughout training, avoiding training stalls or numeric overflows.

5.3 Discussion

Logic Learning To summarize the overall performance of the models, the AND and OR functions achieved exceptionally high accuracies. Their learned weights and biases effectively captured the necessary operations, leading to clearly defined decision boundaries that correctly distinguish the classes. Furthermore, the reduced-dimensionality visualizations of their hidden layer activations demonstrated distinct and well-separated clusters for each class. These consistent findings clearly demonstrate the neural network's strong capacity to easily capture linear relationships.

On the other hand, the XOR function—a non-linear separable classification problem—did not achieve the same level of performance as the linearly separable functions. A significant portion of the data points could not be correctly classified, indicating the network's limitations in learning the complex XOR operation with the current architecture. Building a more robust and complex architecture, potentially with additional hidden layers and more neurons, might resolve this issue.

These findings suggest that while neural networks provide an effective deep-learning-based approach for modeling logic functions, the complexity of the architecture must be considered to create robust and sophisticated models that can generalize well to unseen data.

Gradient Flow The gradient-flow animation confirms that our two-layer `MLP` with sigmoid output behaves as expected: hidden-layer gradients stay in a consistent band (no

vanishing or exploding), while the output-layer gradients occasionally hit zero due to sigmoid saturation. These intermittent zero-spikes are not a sign of pathology but rather the network correctly "pausing" updates once it becomes highly confident on a minibatch. In deeper or more complex architectures, one could use similar visual diagnostics to decide whether to adjust activations, learning-rate schedules, or introduce architectural features (e.g. residual links or normalization) to preserve stable gradient signals.

6 Formulated Conclusions from Analysis

Based on the in-depth theoretical analysis of neural network and supported by the empirical findings from our experiments, the following conclusions can be drawn:

- **Effectiveness of Non-linearity in Linear Problems:** The application of non-linear activation functions (`ReLU` and `Sigmoid`) enables the neural network to achieve near-perfect classification accuracies for linearly separable problems like AND (test accuracy ≈ 0.998) and OR (test accuracy ≈ 0.991). This demonstrates that even for linear problems, non-linear activation functions contribute to robust learning and clear decision boundaries, as proved by the distinct clustering in the `t-SNE` visualizations.
- **Efficacy of Backpropagation for Parameter Optimization:** Backpropagation, powered by the chain rule and the `Adam` optimizer, effectively adjusted network parameters. This is evident from the quick and substantial improvements in model accuracy and loss reduction across all logic functions during training, resulting in well-optimized weights and biases.
- **Impact of Architecture Complexity on Non-linear Problems:** The XOR function's test accuracy is approximately 0.836, demonstrates that while our simple two-hidden-neuron architecture learned some non-linear patterns, it was insufficient. The overlapping `t-SNE` clusters and redundant hidden neuron features underscore that complex non-linear problems require more robust architectures, such as additional layers or neurons, for complete correctness.
- **Stability of Gradient Flow:** Gradient flow visualization reveals that the **Frobenius** norms of the gradients for both the hidden and output layers remain within a consistent range (roughly 10^{-1} to 10^{0}) over 100 epochs, confirming the absence of vanishing or exploding gradients and ensuring stable and reliable training dynamics.

7 Recommendations for Future Research

Following the comprehensive analysis of our experiments, we have identified several insightful recommendations for future research:

1. **Architectural Exploration and Optimization:**

- Future work should explore more complex neural network architectures, particularly for handling non-linearly separable data, to improve generalization on unseen samples.
- A comparative study of various activation functions across different architectural configurations would deepen the understanding of their respective properties and suitability for diverse problem types.
- To further enhance model performance, a more careful and organized method is required to tune hyperparameters.

2. **Visualization and Benchmarking:**

   - Broader Benchmarking: Apply the gradient visualization pipeline to larger real-world datasets (e.g., MNIST, CIFAR-10) and complex architectures (e.g., CNNs, RNNs) to generalize insights beyond toy logic tasks.
   - Visualization Enhancements: Develop interactive dashboards (e.g., using Plotly or Bokeh) to allow real-time exploration of gradient flows, activations, and decision boundaries during training.

References

[1] Wikipedia contributors. Neural network (machine learning). https://en.wikipedia.org/wiki/Neural_network_(machine_learning), June 2025. Accessed: 2025-06-13.

[2] Wikipedia contributors. Backpropagation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Backpropagation, 2025. Accessed: 2025-06-13.

[3] GeeksforGeeks. Activation functions in neural networks. https://www.geeksforgeeks.org/machine-learning/activation-functions-neural-networks/, June 2025. Accessed: 2025-06-13.

[4] M. Balawejder. Optimizers in machine learning. https://medium.com/nerd-for-tech/optimizers-in-machine-learning-f1a9c549f8b4, April 2022. Accessed: 2025-06-13.

[5] Kaggle Community. Vanishing and exploding gradients in deep learning. https://www.kaggle.com/discussions/general/290747, 2023. Accessed: 2025-06-13.

[6] Wikipedia contributors. Logic gate. https://en.wikipedia.org/wiki/Logic_gate, June 2025. Accessed: 2025-06-15.

[7] NetworkX Developers. Networkx 1.7 tutorial: Creating a graph. https://networkx.org/documentation/networkx-1.7/tutorial/tutorial.html, 2013. Accessed: 2025-06-15.

[8] Scikit learn Developers. sklearn.manifold.tsne. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html, 2023. Accessed: 2025-06-15.