Development of Neural Network

Learning AND, OR & XOR via Backpropagation and Analysis of Learned Representation

By
Mateja Zatezalo & Pawarit Jamjod
Evaluator: Johannes Weinert

May 19, 2025

User Guide

First clone our repository and navigate to the project root in console to install the requirements (in a virtual environment) using `pip -r install .\requirements.txt`.

To run the program, run `app.py`. Results of the analysis will be shown in the terminal, and the plots used in the analysis will appear.

Abstract

This work presents a from-scratch implementation of a single-layer perceptron and a two-layer multilayer perceptron trained via backpropagation and binary cross-entropy to model the Boolean functions AND, OR, and XOR.

It also presents an in-depth analysis of network's internal representations, including decision boundaries, hidden-layer activations throughout training and Principal Component Analysis for 2D visualization, augmented by animated trajectories of cluster separation.

Additional analyses include single-unit ablation to assess neuron importance, heat-maps of input$\rightarrow$hidden weight matrices, and experiments with input noise and L2 weight-decay to examine representation robustness.

Contents

1 Introduction and Significance

Neural networks have emerged as groundwork of modern maching learning, due to their ability to approximate complex, nonlinear functions and learn hierarchical representations of data. Multilayer perceptrons (*MLPs*) commonly apply the backpropagation algorithm to adjust weights via gradient descent, enabling the network to discover logical rules from the examples. One of the ways for demonstrating these capabilities are Boolean functions *AND, OR* and *XOR*. While single-layer perceptrons are only able to learn the linearly separable *AND* and *OR* gates, they fail on the non-linearly separable *XOR* gate — a significant result that motivated the further development of a more complex architecture.

The significance of this project entails a from-scratch implementation of both a single-layer perceptron and a two-layer MLP using *NumPy*.[1] Refraining from using more advanced frameworks, we derived and coded every step of forward and backward propagation, binary cross-entropy loss, and stochastic gradient updates. This low-level approach enhances the understanding of backpropagation algorithm, mainly weight initialization, convergence and representational capacity.

In addition, our work presents the analysis of how and why the MLP learns these logical functions. Accuracy and predictions are obtained for all gates, while *XOR* required more in-depth analysis by being the minimal benchmark that distinguishes the representational power of the two perceptrons. Among inspection of hidden-layer representations, we recorded activations at regular training intervals and apply *Principal Component Analysis (PCA)* to project the four discrete input patterns into a two-dimensional latent space. Animated trajectories illustrated how initially overlapping clusters separate as learning proceeds. Single-unit ablation experiments quantified the causal importance of each hidden neuron, and heat-maps of the input→hidden weight matrices revealed the logical sub-functions each neuron encodes.

Finally, we extended the study to robustness under input noise and L2 weight-decay, examining how regularization preserves clear latent clustering and generalization in the presence of corrupted inputs. This additional experiment underscored the real-world relevance of the methods: neural networks must not only learn accurately on clean data, but also maintain stable, interpretable representations when faced with uncertainty.

Our project offers a concise yet comprehensive demonstration of the ability of backpropagation,

[1]Numpy developers (2024).

complexity of non-linear tasks, and tools for hidden-layer logic interpretation, forming a solid foundation of neural network research.

2 Model Implementation and Training

In this section we describe our from-scratch implementation of both the single-layer perceptron and the two-layer MLP, the motivating implementation choices behind each, and the utility functions that tie everything together.

2.1 Single-layer Perceptron

We chose to begin with the classic single-layer perceptron as a baseline. The purpose for this was to:

- Demonstrate that a simple linear model can learn *AND* and *OR* perfectly
- Emphasize the inability to learn *XOR* and motivating the more complex architecture

This perceptron was implemented in `models/linear_perceptron.py`, and it defines:

- **Parameters:** Weight matrix W of shape $D \times 1$, where D is the number of dimensions of the Boolean input, and bias b, which is a scalar.
- **Forward pass:** We apply the **logistic sigmoid** element-wise to z (pre-activation):

  $$\hat{p} = \sigma(z), \quad \sigma(z) = \frac{1}{1+e^{-z}}$$

  so $\hat{p} \in (0,1)^N$ is the model's estimated probability of the *"1"* class for each input.
- **Loss function:** We used **binary cross-entropy** to measure the mismatch between true labels $\hat{t} \in (0,1)^N$ and predictions $\hat{p}$:

  $$\mathscr{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[t_i \log \hat{p}_i + (1-t_i)\log(1-\hat{p}_i)\right]$$

  This loss is convex in W, b for the perceptron, which guarantees a unique global minimum for linearly separable data.
- **Backward pass:** We compute the gradient w.r.t z, then gradients w.r.t W and b followed

by the chain rule:

$$\frac{\partial \mathscr{L}}{\partial z} = \frac{\hat{p} - t}{N}, \quad \frac{\partial \mathscr{L}}{\partial W} = X^\top \frac{\partial \mathscr{L}}{\partial z}, \quad \frac{\partial \mathscr{L}}{\partial b} = \sum_{i=1}^{N} \frac{\partial \mathscr{L}}{\partial z_i}.$$

Then the parameters are updated with stochastic gradient descent:

$$W \leftarrow W - \eta \nabla_W, \quad b \leftarrow b - \eta \nabla_b.$$

where η is the learning rate of value 0.1.

Because *AND* and *OR* are linearly separable, this single-layer model converges to approximately zero training loss on those gates, but it cannot solve XOR, which is what motivates our two-layer MLP.

2.2 Two-layer Multilayer Perceptron

Due to the aforementioned limitations of the single-layer perceptron, we extended the model by adding a hidden layer. This hidden layer consists of H sigmoid units, creating a 2-layer network with parameters:

$$W_1 \in \mathbb{R}^{D \times H}, \quad b_1 \in \mathbb{R}^{1 \times H}, \quad W_2 \in \mathbb{R}^{H \times 1}, \quad b_2 \in \mathbb{R}^{1 \times 1}.$$

where $D = 2,\ H = 3$ in our case. We chose a default of 3 hidden units give the network a bit more capacity. This model was implemented in `models/tiny_mlp.py`. It is called *TinyMLP* to emphasize that it is not a full deep-learning library module, but a simple, transparent implementation designed for analysis. Along with the parameters above, it defines:

- **Forward pass:**

  1. **Hidden pre-activation**

  $$Z_1 = XW_1 + b_1, \quad X \in \mathbb{R}^{N \times D},\ Z_1 \in \mathbb{R}^{N \times H}.$$

  2. **Hidden activation**

  $$A_1 = \sigma(Z_1), \quad \sigma(z) = \frac{1}{1 + e^{-z}},\ (A_1 \in (0,1)^{N \times H}).$$

3. **Output pre-activation**

$$Z_2 = A_1 W_2 + b_2, \quad Z_2 \in \mathbb{R}^{N\times 1}.$$

4. **Output probability**

$$\hat{P} = \sigma(Z_2), \quad \hat{P} \in (0,1)^{N\times 1}.$$

- **Loss function:** Here we again used **binary cross-entropy** against true labels $\hat{T} \in (0,1)^{N\times 1}$ as for the single-layer perceptron (subsection 2.1).

- **Backward pass:**

1. **Gradient w.r.t. output pre-activation**

$$\frac{\partial \mathscr{L}}{\partial Z_2} = \frac{\hat{p} - T}{N}, \quad \frac{\partial \mathscr{L}}{\partial Z_2} \in \mathbb{R}^{N\times 1}.$$

2. **Gradients for second layer**

$$\nabla_{W_2}\mathscr{L} = A_1^T \frac{\partial \mathscr{L}}{\partial Z_2}, \quad \nabla_{W_2}\mathscr{L} \in \mathbb{R}^{H\times 1},$$

$$\nabla_{b_2}\mathscr{L} = \sum_{i=1}^{N} \frac{\partial \mathscr{L}}{\partial Z_{2,i}}, \quad \nabla_{b_2}\mathscr{L} \in \mathbb{R}^{1\times 1}.$$

Each weight in W_2 connects one hidden unit to the output. Its update is a sum over all samples of “hidden-activation × output-error.”, while bias is the sum of output errors.

3. **Propagate to hidden layer**

$$\frac{\partial \mathscr{L}}{\partial A_1} = \frac{\partial \mathscr{L}}{\partial Z_2} W_2^T, \quad \frac{\partial \mathscr{L}}{\partial A_1} \in \mathbb{R}^{N\times H},$$

$$\frac{\partial \mathscr{L}}{\partial Z_1} = \frac{\partial \mathscr{L}}{\partial A_1} \odot \sigma'(Z_1), \quad \sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

This shows how much each hidden activation influenced the output, and each hidden neuron’s error is scaled by how "active" it was.

4. **Gradients for first layer**

$$\nabla_{W_1}\mathscr{L} = X^T \frac{\partial \mathscr{L}}{\partial Z_1}, \quad \nabla_{W_1}\mathscr{L} \in \mathbb{R}^{D\times H},$$

$$\nabla_{b_1}\mathscr{L} = \sum_{i=1}^{N} \frac{\partial \mathscr{L}}{\partial Z_{1,i}}, \quad \nabla_{b_1}\mathscr{L} \in \mathbb{R}^{1\times H}.$$

Same pattern as the output layer—each weight update is "input-feature × hidden-error," summed over samples.

- **L2 weight-decay (regularization)**: In later analysis, we add weight-decay to penalize large weights, encouraging smoother, more robust representations:

$$\nabla_{W_k} \leftarrow \nabla_{W_k} + \lambda\, W_k, \quad k = 1, 2.$$

Stochastic gradient descent then updates:

$$W_k \leftarrow W_k - \eta\, \nabla_{W_k}, \quad b_k \leftarrow b_k - \eta\, \nabla_{b_k}, \quad k = 1, 2.$$

where η is the learning rate.

- **Snapshot recording:** To later analyze learning dynamics, we keep track of:
  - **Hidden activations** $A_1 \in \mathbb{R}^{4\times H}$ every snapshot epoch into `self.hist`.
  - **Loss value** $\mathscr{L}$ into `self.loss_hist`.

  These records enable all downstream analyses (PCA animation, ablation, heat-maps).

By deriving from-scratch every gradient step for both layers, we delved into backpropagation mechanics while demonstrating how the hidden layer enables the network the ability to solve the non-linearly separable *XOR* problem.

2.3 Training

2.3.1 Accuracy & Loss Summary

(a) AND gate

Model	Accuracy	Loss
Perceptron	1.00	0.00868
TinyMLP	1.00	0.00156

(b) OR gate

Model	Accuracy	Loss
Perceptron	1.00	0.00460
TinyMLP	1.00	0.00112

(c) XOR gate

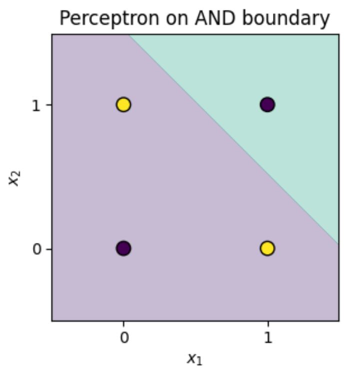
Model	Accuracy	Loss
Perceptron	0.50	0.69315
TinyMLP	1.00	0.19178

Table 1: Accuracy and loss for each Boolean gate.

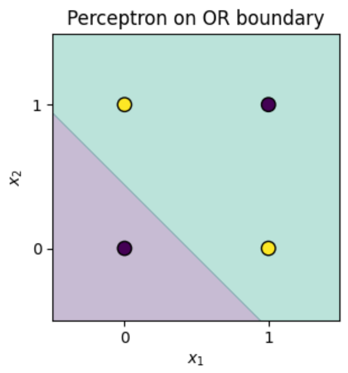
These results confirm that both the Perceptron and our TinyMLP achieve perfect accuracy (and very low loss) on the linearly separable *AND* and *OR* gates. Crucially, however, when tasked with *XOR*, the Perceptron remains at 50% accuracy (loss $\approx$ 0.693), whereas the two-layer TinyMLP converges to 100% accuracy (loss $\approx$ 0.192). For this, we adjusted the learning rate to 0.1, and number of epochs to 20 000. This contrast shows why depth is required: a single linear decision boundary cannot solve the *XOR* pattern, but the MLP's hidden layer enables the non-linear "X"-shaped boundary needed to solve *XOR*.

2.3.2 Decision Boundary

On *AND* and *OR* gates, for both the single-layer perceptron and the TinyMLP, the decision boundary is an identical single line, correctly partitioning the blue ("0") and yellow ("1") points, as shown in Figure 1.



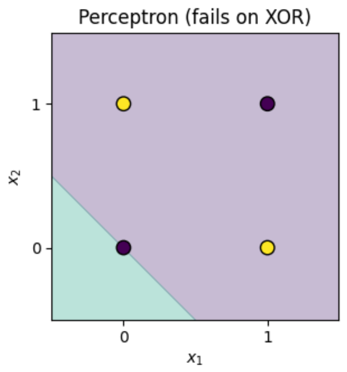
(a) Decision boundary on AND



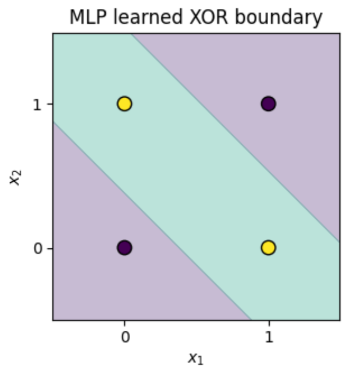
(b) Decision boundary on OR

Figure 1: Linear decision boundaries for the AND and OR gates.

The single-layer perceptron on *XOR* collapses to predicting "0" everywhere, since no single linear boundary can separate the diagonally-opposed positive examples. On the other hand, the two-layer network carves out an X-shaped decision region that hugs the four Boolean inputs, perfectly separating the "1" patterns $(0,1),(1,0)$ from the "0" patterns $(0,0),(1,1)$.



(a) Perceptron - Decision boundary on XOR



(b) TinyMLP - Decision boundary on XOR

Figure 2: Decision boundaries for the XOR gate.

These results confirm that *AND* and *OR*—being linearly separable—pose no challenge to either model, whereas *XOR* can only be learned by the MLP's hidden units. These figures directly illustrate the fundamental result that depth (at least one hidden layer) is required to learn all simple Boolean functions.

3 Representation Analysis

In this section, we dive into the MLP's hidden-layer representations, to reveal how the network organizes the four Boolean inputs and carves out the *XOR* logic. We concentrate our representation analysis on the *XOR* task because, for the linearly separable *AND* and *OR* gates, the hidden-layer simply learns redundant linear features and offers no deeper insight. In contrast, *XOR* is the minimal non-linear problem that forces the network to develop genuinely distinct sub-functions in its hidden units; its evolving clusters, unit-specific ablations, and weight patterns therefore provide the richest window into how backpropagation carves logical structure into the latent space.

3.1 Hidden layer activations

After training the TinyMLP on *XOR* (with $H = 3$ hidden units), the final activations on the four Boolean inputs $[0,0],[0,1],[1,0],[1,1]$ are:

Input	h_1	h_2	h_3
$[0,0]$	0.998	0.556	0.918
$[0,1]$	0.886	0.649	0.023
$[1,0]$	0.888	0.668	0.023
$[1,1]$	0.109	0.748	0.000

Table 2: Final hidden-layer activations h_1, h_2, h_3 on the four XOR input patterns.

From results displayed in Table 2, we discovered that neuron h_1 fires on all inputs except $[1,1]$ (*NAND*-like), h_2 grows with the number of 1's, while h_3 fires only on $[0,0]$ (*NOR*-like). With generated output weights [+8.46, −1.35, −9.09], the network computes:

$$z_{\text{out}} = 8.46 h_1 - 1.35 h_2 - 9.09 h_3 + b_2, \quad \hat{y} = \sigma(z_{\text{out}})$$

so exactly-one-"1" patterns (where h_1 is high and h_3 is low) yield $z_{\text{out}} \gg 0 (\hat{y} \approx 1)$, while $[0,0]$ or $[1,1]$ produce $z_{\text{out}} \ll 0 (\hat{y} \approx 0)$. Backpropagation has automatically discovered classic Boolean primitives (*NAND*, *NOR* and a graded *OR*) and recombined them to implement *XOR*. This demonstrates that the hidden layer does learn interpretable features. Figure 7 in Appendix A.1 further visualizes patterns and results of this analysis.

3.2 PCA of Final Hidden Activations

After extracting the final hidden-layer activations $H_{\text{final}} \in \mathbb{R}^{4\times3}$, we applied PCA to reduce to two dimensions. Figure 3 shows:

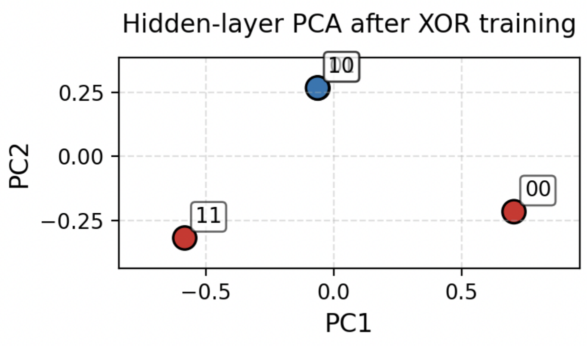
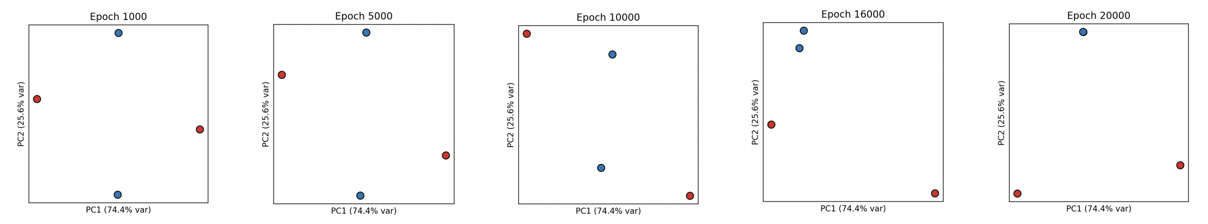


Figure 3: PCA of final hidden-layer activations.

PC1 captures the majority of variance (74%) and separates the *XOR* classes: the two red points (00, 11) lie on one side, the two blue points (01, 10) on the other. **PC2** (26% of variance) shows minimal or no difference in how the hidden units respond to inputs 01 and 10. This analysis shows that the network transforms the *XOR* geometry into a linearly separable layout.

3.3 PCA Clustering Trajectories over Epochs

To capture how the network gradually pulls apart the *XOR* classes in hidden-space, we recorded hidden-layer activations every 1000 epochs and animated their *2D* PCA projection. Figure 4 shows the frames at different epochs, but while running `app.py`, the animation updates automatically every 1000 epochs to reveal the full separation dynamics.



(a) Epoch 1 000 (b) Epoch 5 000 (c) Epoch 10 000 (d) Epoch 16 000 (e) Epoch 20 000

Figure 4: Select frames from the PCA animation of hidden-layer activations on XOR. Red points denote outputs 00/11, blue points 01/10.

At the start, all points remain clustered near the origin—no class structure has emerged. At epoch 5 000, the two "0" patterns (red) begin drifting to opposite sides along PC1, while the "1" patterns (blue) remain the same. At epoch 10 000, separation along PC1 is now clear, red and blue occupy different sides of the axis. Continuing the training, the two clusters tighten, almost perfectly linearly separable along PC1. The final snapshot shows two well-defined clusters. This animation vividly illustrates the non-linear geometry of the Boolean function.

3.4 Single-Unit Ablation

In order to measure the causal role of each hidden neuron in implementing *XOR*, we performed a simple ablation test: for each hidden unit i, we zeroed out its outgoing weight $W_{2,i}$, re-computed the network's predictions on the four *XOR* inputs, and recorded the resulting accuracy.

Ablated Unit	**Accuracy**
Neuron 0 (NAND)	0.50
Neuron 1 (OR count)	1.00
Neuron 2 (NOR)	0.75

Table 3: Accuracy after zeroing one hidden-to-output weight at a time.

Along with our study in Subsection 3.1, ablation confirms that neurons 0 (*NAND*-like) and 2 (*NOR*-like) contribute significantly, while neuron 1 is redundant for correctness, and may serve to adjust the decision boundary.

3.5 Weight and Activation Heatmaps

To round out our representation analysis, we visualized both the learned input$\rightarrow$hidden weights and the final hidden-unit activations, and also compared how these maps change under noisy inputs and L2 regularization.

3.5.1 Input→Hidden Weight Heatmap

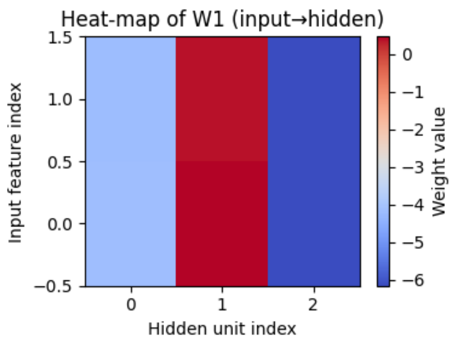


Figure 5: Heat-map of the trained input→hidden weight matrix W_1. Rows correspond to the two input features, columns to hidden units.

Column 0 shows large negative weights on both x_1 and x_2, so it fires unless both inputs are 1. Column 1 shows moderate positive weights, while Column 2 displays large positive weights ensure activation only when both inputs are zero.

3.5.2 Clean vs. Noisy & Weight-Decay Heatmaps

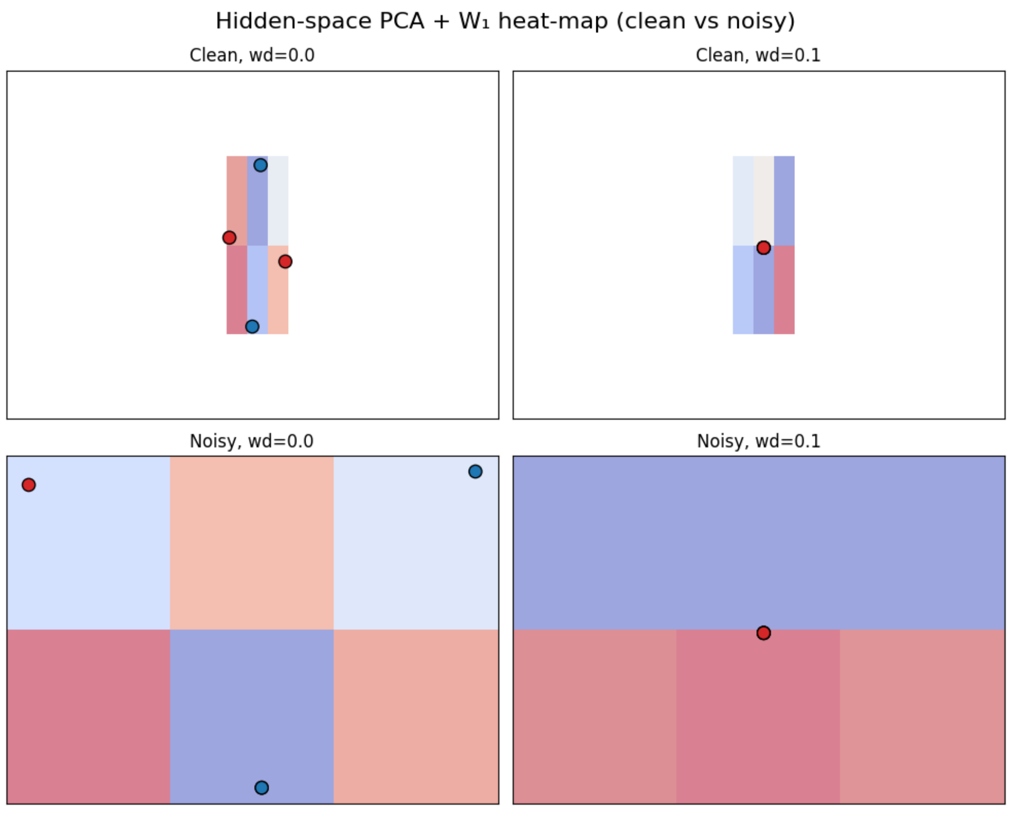


Figure 6: Hidden-space PCA plus W_1 heatmap under four conditions: (a) clean data, no weight-decay; (b) clean data, $\lambda = 0.1$ L2; (c) noisy data, no weight-decay; (d) noisy data, $\lambda = 0.1$ L2.

To test how stable the learned *XOR* representation is under input noise, we retrained four TinyMLP models on clean vs. 10% noisy inputs, and with vs. without L2 weight-decay $\lambda = 0.1$. Figure 6 shows each model's final PCA clustering (points) overlaid with W_1 heatmap:

- (a) **Clean**, $\lambda = 0$: Four points form two well-separated clusters along PC1, identical to our earlier scatter plot 3.2. Clean data with no L2 allows a small amount of variance in hidden activations but still yields perfectly separable clusters.
- (b) **Clean**, $\lambda = 0.1$: The four PCA points collapse into a single overlapping spot, indicating that the regularized model has driven hidden-space variance to near zero on clean data, a sign of an invariant encoding.
- (c) **Noisy**, $\lambda = 0$: Clusters become smeared, points drift toward noise and no longer form clearly separable clusters. Weight map is uneven, with some hidden units "over-reacting" to corrupted inputs, indicating overfitting. Without regularization, noise in the inputs produces a high-variance latent space that still classifies correctly but lacks interpretability.
- (d) **Noisy**, $\lambda = 0.1$: L2 weight-decay has over-regularized the network on noisy inputs, collapsing all hidden activations to the same point and erasing the *XOR* structure. Hence, the model regresses to the trivial 50% predictor.

To conclude with, a moderate amount of regularization on clean data preserves a tight, interpretable hidden-space, but under noise it either underfits (too much λ) or overfits (too little λ), so finding the right balance is key to robust representations.

4 Future Work

We believe that our from-scratch implementation provides a fully sufficient platform for not only fitting the Boolean gates, but also for carrying out the representation analyses.

Nevertheless, future work regarding this topic may include re-implementing the TinyMLP in PyTorch[2] or TensorFlow[3] to leverage automatic differentiation, GPU acceleration, and richer tooling. This approach would not be from-scratch, but would allow scaling to larger networks and datasets.

Another addition would include exploration of how adding more hidden layers or units (along

[2]Paszke et al. (2019).
[3]Abadi et al. (2016).

with modern techniques like dropout or batch normalization) affects the speed and stability of learning logical functions.

Utilization of alternative activations (*ReLU* or *tanh*) activations and corresponding losses could yield an insight on how the choice of nonlinearity shapes the emergence of logical primitives.

5 Conclusion

This project demonstrated a from-scratch implementation and analysis with the aim to show how backpropagation enables neural networks to learn both linearly separable (*AND*, *OR*) and non-linear (*XOR*) Boolean functions. Our single-layer Perceptron served as a clear baseline, perfect on *AND/OR* but hopeless on *XOR*, while our two-layer TinyMLP brought in the hidden-layer capacity needed to solve *XOR* with perfect accuracy.

By recording hidden-layer activations, applying PCA, animating their trajectories, performing single-unit ablations, and visualizing weight and activation heat-maps, we displayed exactly how the network carves logical sub-functions into distinct neurons, and how these primitives recombine to implement *XOR*. Our robustness experiments further revealed the delicate bias–variance trade-off: unregularized models overfit noisy inputs, while overly strong L2 underfit and collapse to trivial predictors.

From manual derivation of forward/backward passes to rich representation analyses, we confirmed that even a “tiny” MLP can yield interpretable hidden-layer features. The techniques and insights here lay a solid foundation for scaling up to deeper networks, more complex logic, or real-world data, and underline the value of introspection tools like PCA and ablation in understanding neural computation.

Appendix

A.1 Final Hidden Activations Heatmap

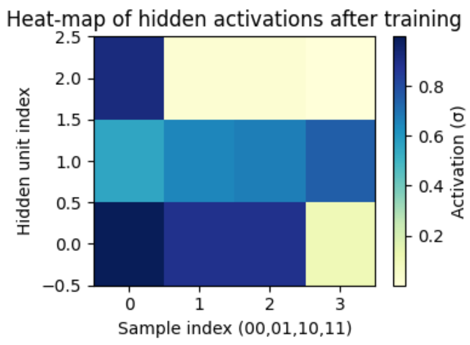


Figure 7: Heat-map of the final hidden-layer activations A_1, with rows as hidden units and columns as the four input patterns.

This heatmap visualizes the results and the pattern obtained in Subsection 3.1, showing that each neuron has specialized into a clear logical primitive, which the output layer then recombines to implement *XOR*.

References

Abadi et al. (2016): TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, Software available at https://www.tensorflow.org.

Numpy developers (2024): Numpy, https://numpy.org/ (23.04.2025).

Paszke et al. (2019): PyTorch: An Imperative Style, High-Performance Deep Learning Library, Proceedings of NeurIPS 32 (2019), Software available at https://pytorch.org.