VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 5: Practice with Pytorch #2

Course: Digital Image and Video Processing

Supervisors:

Prof. Dr. Ly Quoc Ngoc

MSc. Nguyen Manh Hung

MSc. Pham Thanh Tung

Student:

23127266 - Nguyen Anh Thu

January 1, 2026

Contents

List of Figures

List of Tables

Chapter I: Introduction

1. Overview

In this assignment, I implement a feed-forward neural network using a modular design approach in PyTorch. Instead of using predefined network structures, each component of the network is constructed and organized into separate modules.

The implementation includes fully connected layers, activation functions, loss functions, and the training procedure based on forward and backward propagation. This modular approach allows flexible network configuration and provides a clear understanding of how individual components interact within a feed-forward neural network.

2. Definitions

2.1. PyTorch

PyTorch is an open-source machine learning framework for building and training neural networks. It supports efficient tensor computation, automatic differentiation, and flexible model construction on both CPU and GPU, making it suitable for research and education. [4]

2.2. Tensor

A tensor is the core data structure in PyTorch, representing a multi-dimensional array. It generalizes scalars, vectors, and matrices and is used to store both input data and model parameters. [5]

2.3. Tensors in PyTorch

A PyTorch tensor is similar to a NumPy array in that it represents an n-dimensional numerical array without inherent knowledge of neural networks or gradients. The key difference is that PyTorch tensors can be computed on either CPU or GPU, enabling hardware acceleration. [1]

PyTorch tensors can share memory with NumPy arrays when created using `torch.from_numpy()`, allowing changes in one to affect the other. This mechanism improves efficiency and avoids redundant memory usage.

Common tensor operations include element-wise computation, matrix multiplication, reshaping, normalization, and reduction, which together form the computational foundation of neural network training and inference.

2.4. Neural Network

A neural network is a computational model inspired by biological neural systems. In this laboratory, a Feed Forward Neural Network (FFNN) is implemented, consisting of an input layer, one or more hidden layers, and an output layer. [3]

Each layer is formed by neurons connected through weighted edges. The network processes data through a sequence of linear transformations followed by nonlinear activation functions. During training, the network learns by iteratively updating its weights to minimize the error between predicted outputs and target values.

2.4.1 Fully Connected Layer

A fully connected layer is a neural network layer in which each neuron is connected to all neurons in the preceding layer. In this implementation, fully connected layers are used to connect consecutive layers within a feed-forward neural network. [2]

Given an input vector $x \in \mathbb{R}^{1\times n}$, the output of a fully connected layer is computed as:

$$y = xW + b$$

where $W \in \mathbb{R}^{n\times m}$ is the weight matrix and $b \in \mathbb{R}^{1\times m}$ is the bias vector.

The resulting output $y \in \mathbb{R}^{1\times m}$ is then passed to an activation layer or directly forwarded to the next layer in the network.

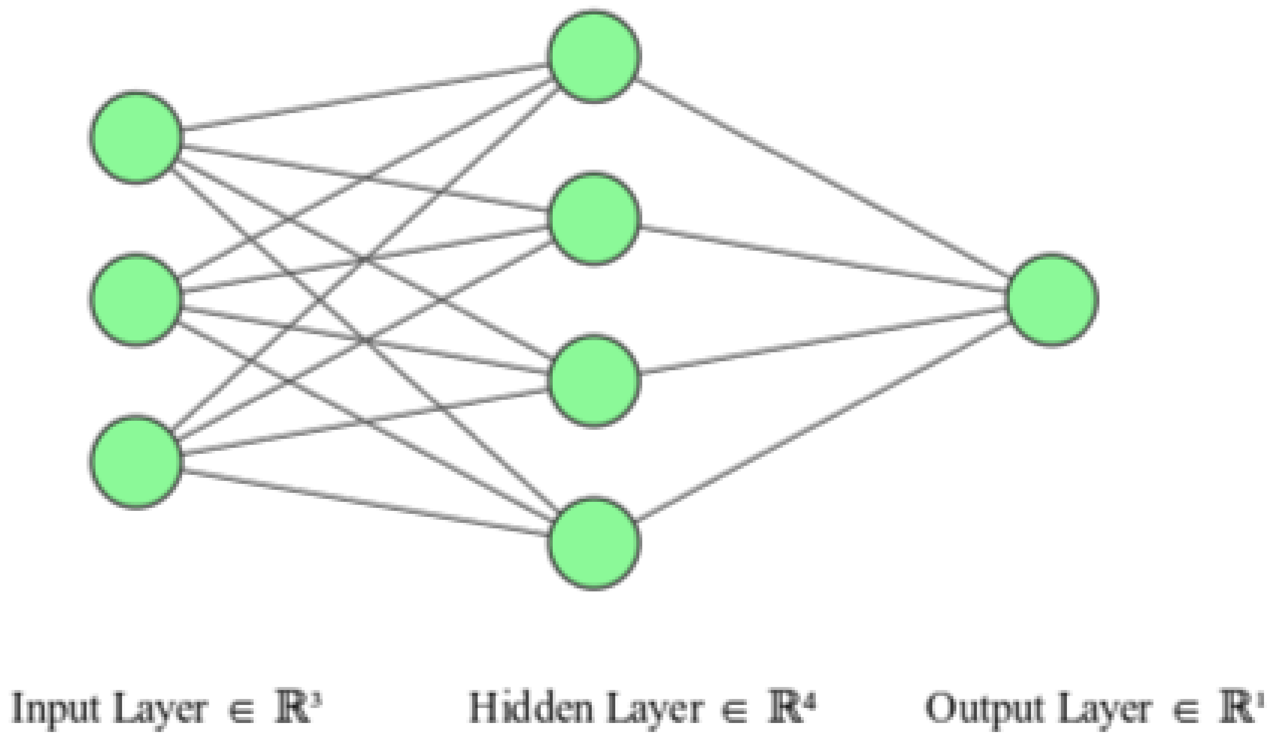

Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^4$
Output Layer $\in \mathbb{R}^1$

Figure I.1: Feed Forward Neural Network Architecture

2.4.2 Core concepts involved in the implemented network include:

- **Weight matrices and bias vectors**: Trainable parameters stored in fully connected layers (`FCLayer`), representing connections between consecutive layers
- **Activation functions**: Nonlinear functions (`tanh` or `sigmoid`) and their corresponding derivatives applied through activation layers
- **Loss function**: Mean Squared Error (MSE) used to measure the difference between predicted outputs and target values
- **Forward propagation**: Sequential data flow through network layers to compute output predictions
- **Backpropagation**: Layer-wise gradient computation and parameter updates performed in reverse order
- **Gradient-based optimization**: Sample-wise stochastic gradient descent controlled

by the learning rate

3. Objectives

No.	Implementation Task
1	Implement a Feed Forward Neural Network with modular layers.
2	Train the network using forward propagation, backpropagation, and sample-wise SGD.
3	Analyze training behavior by varying key hyperparameters.

Table I.1: Objectives of the laboratory

4. Program Structure

Input

- Training data $x_{\text{train}} \in \mathbb{R}^{N\times 1\times 2}$ and target values $y_{\text{train}} \in \mathbb{R}^{N\times 1\times 1}$, provided as tensors for supervised learning
- Experimental configuration parameters including hidden layer size, activation function, learning rate, and number of training epochs

Output

- Predicted output values y_{pred} produced by the trained feed-forward neural network
- Saved network parameters (weights and biases) stored in `xor_model.pkl`

Chapter II: Implementation

1. Network Construction

The network is organized as a sequential structure, where layers are dynamically added and executed in order during training and inference. This design supports flexible configuration of network architecture and provides a clear framework for conducting implementation-based experiments.

1.1. Class Structure

The program is organized in an object-oriented manner with the following main classes:

- **BaseLayer**: an abstract base class that defines a common interface for all layers through two methods `forward()` and `backward()`.
- **FCLayer**: a fully-connected (linear) layer that performs linear transformation between input and output, including both weights and bias.
- **ActivationLayer**: an activation layer that applies a nonlinear function element-wise and computes the corresponding gradient in the backward propagation phase.
- **Network**: a class that manages the entire network, responsible for adding layers, configuring the loss function, training the model, and making predictions.

2. Activation Functions

To introduce nonlinearity to the model, this practice implements two common activation functions: **sigmoid** and **tanh**. These functions are defined in the `Activation` class, while their corresponding derivatives are implemented in the `ActivationPrime` class for backpropagation.

2.1. Sigmoid Function

The sigmoid function is defined as follows:

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

This function maps any real value to the range $(0, 1)$, commonly used in binary classification problems. In the program, sigmoid is directly implemented using PyTorch tensor operators to ensure computational efficiency.

2.2. Tanh Function

The hyperbolic tangent (tanh) function is defined by:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Unlike sigmoid, the tanh function has a range of $(-1, 1)$ and is zero-centered, which helps stabilize gradient propagation in many cases. Therefore, tanh is commonly used in the experiments of this practice.

3. Forward Propagation

Forward propagation is the process of computing the network output based on input data. In this model, data is passed sequentially through each layer in the layer list of the `Network` class.

For each fully-connected layer, the linear transformation is performed according to the formula:

$$Z = XW + b$$

where X is the input tensor, W is the weight matrix, and b is the bias vector.

Then, the output Z is passed through the activation layer to produce nonlinear output. These intermediate values are stored in each layer to serve the backpropagation process.

3.1. Forward Pass Steps

Forward propagation is performed sequentially according to the following steps:

1. Receive input data from the training set.
2. Pass data through each fully-connected layer.
3. Apply the corresponding activation function at each activation layer.
4. Obtain the final output of the network.

This implementation allows the network to support any number of hidden layers, not limited to a single hidden layer as in simple models.

3.2. Implementation Details

The `forward()` method of each layer is implemented independently and called sequentially in the `Network` class. PyTorch tensors are used throughout the computation process to ensure efficiency and consistency in data dimensions.

4. Backward Propagation

Backward propagation is used to compute the gradients of the loss function with respect to all trainable parameters in the network and to update these parameters in order to minimize the loss.

The network in this practice is trained using the Mean Squared Error (MSE) loss function, defined as:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2,$$

where y denotes the ground truth, $\hat{y}$ is the predicted output, and N is the number of elements in the output tensor.

4.1. Gradient Computation

The backpropagation process is implemented manually and follows the reverse order of the forward pass. For each training sample, gradients are propagated from the output layer back to the input layer using the chain rule.

Gradient from the loss function. The derivative of the MSE loss with respect to the predicted output is given by:

$$\frac{\partial L}{\partial \hat{y}} = \frac{2(\hat{y} - y)}{N}.$$

This gradient serves as the initial error signal for the backward propagation process.

Backpropagation through the activation layer. Each activation layer applies a non-linear function element-wise. During backpropagation, the gradient with respect to the pre-activation input z is computed by:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \odot f'(z),$$

where $a = f(z)$ is the activation output and $\odot$ denotes element-wise multiplication.

For the activation functions used in this project, the derivatives are:

$$\tanh'(z) = 1 - \tanh^2(z),$$

$$\text{sigmoid}'(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

The activation layer does not update any parameters; it only propagates the gradient backward.

Backpropagation through the fully-connected layer. A fully-connected layer performs the linear transformation:

$$Z = XW + b,$$

where X is the input vector, W is the weight matrix, and b is the bias. Given the gradient with respect to the output Z, $\delta = \partial L/\partial Z$, the gradients are computed as:

$$\frac{\partial L}{\partial W} = X^\top \delta,$$

$$\frac{\partial L}{\partial b} = \sum \delta,$$

$$\frac{\partial L}{\partial X} = \delta W^\top.$$

Parameter update using SGD. The network parameters are updated using Stochastic Gradient Descent (SGD). With a learning rate α, the update rules are:

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W},$$
$$b \leftarrow b - \alpha \frac{\partial L}{\partial b}.$$

Since training is performed in a sample-wise manner, no batch averaging is required, and the bias gradient is computed as a direct sum over the output error tensor.

Backward propagation order. During training, the backward pass iterates through all layers in reverse order compared to the forward pass. At each layer, gradients are computed and propagated to the preceding layer until gradients with respect to the network input are obtained.

5. Training Process

5.1. Input Data

The training dataset used in this practice is the XOR problem. It consists of four input samples and their corresponding target labels. Each input sample contains two features, and each target output contains a single value.

The input–output pairs are defined as:

$$(0,0) \to 0,$$
$$(0,1) \to 1,$$
$$(1,0) \to 1,$$
$$(1,1) \to 0.$$

In the implementation, the input data is stored as a tensor of shape $(4,1,2)$, and the target labels are stored as a tensor of shape $(4,1,1)$. All values are represented using floating-point tensors. The input data is used directly as defined, without any preprocessing or normalization steps.

5.2. Training Procedure

Training is performed using a sample-wise Stochastic Gradient Descent (SGD) approach. During training, each sample in the training dataset is processed independently within each epoch.

For each epoch, the network iterates through all training samples and performs the following steps for each sample:

1. Forward propagation through all layers to compute the predicted output.
2. Loss computation using the Mean Squared Error (MSE) loss function.
3. Backward propagation through all layers to update the weights and bias.

After all samples have been processed in an epoch, the loss values are averaged and printed periodically to monitor the training progress.

6. Model saving and loading

6.1. State Dictionary

The model persistence is implemented using a custom `state_dict` mechanism that stores all learnable parameters (weights and biases) from fully connected layers in the network. The `Network` class provides two methods for managing state:

- `state_dict()`: Returns a dictionary containing weights and biases for all fully connected layers. Keys follow the format `layer_{i}_weights` and `layer_{i}_bias` where `i` is the layer index.
- `load_state_dict(state_dict)`: Loads weights and biases from a state dictionary into the network, with validation to ensure the structure matches.

At initialization, weights are randomly sampled from a normal distribution. When a saved `state_dict` is loaded, these initial random values are completely replaced by the trained weights, ensuring consistent model restoration.

6.2. Saving Model

The model is saved using the `save(filepath)` method of the `Network` class. This method:

1. Extracts the state dictionary via `self.state_dict()`
2. Serializes the dictionary to disk using Python's `pickle` module

3. Saves the file with the specified path (`xor_model.pkl`)

The implementation uses `pickle.dump()` to write the state dictionary to a binary file. This approach stores only the model parameters (weights and biases), not the entire model object, making the saved file lightweight and architecture-independent.

6.3. Loading Model

To load a saved model, the static method `Network.load(filepath, network)` is used:

1. A new `Network` instance with the same architecture as the saved model must be created first
2. The saved state dictionary is loaded from the file using `pickle.load()`
3. The state dictionary is applied to the network instance via `network.load_state_dict()`
4. The method validates that the state dictionary structure matches the network architecture

The `load_state_dict()` method performs validation to ensure:

- The number of items in the state dictionary matches the number of fully connected layers (2 items per layer: weights and bias)
- All required keys (`layer_{i}_weights` and `layer_{i}_bias`) are present for each layer
- Raises `ValueError` if there is a mismatch

6.4. Usage Example

The typical workflow for saving and loading a model is:

1. **Train the model**: Create and train a network with desired architecture and hyperparameters
2. **Save the model**: Call `net_train.save("xor_model.pkl")` to persist the trained weights
3. **Create new instance**: Build a new network with the same architecture

4. **Load the model**: Call `Network.load("xor_model.pkl", net_loaded)` to restore weights

After loading, the model can be used for inference through forward propagation. Given the same input data and network architecture, the loaded model is expected to produce the same predictions as those obtained before saving.

Chapter III: Installation and Usage

This section describes how to set up the environment, install required libraries, and execute the program. It also provides a comprehensive guide to the implementation structure and execution steps.

1. Environment Setup

1.1. System Requirements

This PyTorch Feed Forward Neural Network project requires the following system components:

- Python 3.6 or higher (Python 3.12.9 tested);
- PyTorch library (version 2.7.1+cpu or compatible);
- NumPy library (version 2.2.3 or compatible);
- Jupyter Notebook or Python script environment for execution;
- RAM: Minimum 2GB (recommended 4GB);
- Disk space: Approximately 500MB for dependencies and saved model files.

1.2. Python and Library Installation

To set up the development environment, perform the following steps:

a) Create a Virtual Environment.

A virtual environment is recommended to isolate project dependencies.

```
python -m venv .venv
```

Activate the virtual environment:

```
# Windows:
.venv\Scripts\activate
```

```

# Linux/Mac:
source .venv/bin/activate
```

b) Install Required Libraries.

Install PyTorch for CPU-only execution:

```
pip install torch
```

Install NumPy and additional supporting libraries:

```
pip install numpy matplotlib pandas
```

For interactive execution using notebooks:

```
pip install jupyter notebook
```

1.3. Main Libraries

The project uses the following Python libraries:

- `torch`: Core library for tensor computation and manual implementation of neural network layers;
- `numpy`: Numerical computing library used for data handling and array manipulation;
- `pickle`: Built-in Python module used to serialize and save trained model parameters;
- `matplotlib`: Visualization library used for plotting experimental results;
- `pandas`: Data analysis library used to summarize and compare experimental outcomes.

1.4. Installation Check

To verify that the installation was successful, run Python and import the required libraries:

```
python
>>> import torch
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> print(torch.__version__)
>>> print(np.__version__)
```

If the installation is correct, the commands will execute without errors and display the corresponding version numbers.

2. How to Run the Program

2.1. Running the Notebook

The implementation is provided as a Jupyter Notebook (`23127266_p05.ipynb`) that can be executed interactively. The notebook is organized into 9 sequential sections that build upon each other.

a) Start Jupyter Notebook.

Navigate to the project directory and launch Jupyter Notebook:

```
jupyter notebook
```

This will open Jupyter in your web browser. Navigate to and open `23127266_p05.ipynb`.

b) Execute Cells Sequentially.

Execute cells in order by clicking "Run" or pressing Shift+Enter. The notebook follows this structure:

1. **Import and Setup** - Import required libraries (`torch`, `numpy`, `matplotlib`), set random seed for reproducibility

2. **Activation Functions** - Define `Activation` class with sigmoid, tanh, ReLU and their derivatives
3. **Loss Functions** - Define `Loss` and `LossPrime` classes with MSE and its derivative
4. **Layers** - Implement `FCLayer` and `ActivationLayer` classes with forward and backward methods
5. **Network Class** - Implement `Network` class as a sequential container with training, prediction, and persistence methods
6. **Training Data** - Prepare XOR problem data and demonstrate basic network training
7. **Network Builder Utilities** - Provide helper functions (`build_network`, `build_net_with_depth`, `calculate_accuracy`, `plot_decision_boundary`)
8. **Training and Testing** - Perform training experiments with different architectures and hyperparameters, visualize decision boundaries
9. **Save and Load Model** - Demonstrate model persistence: train model, save to file, load from file, verify predictions match

2.2. Model File

After training and saving in Section 9, a file named `xor_model.pkl` will be created in the current directory. This file contains the model's state dictionary (weights and biases for all fully connected layers) serialized using Python's pickle module. The file can be loaded in subsequent runs without retraining, as long as the network architecture matches the saved model.

3. Project Directory Structure

3.1. Project Structure

The implementation is organized as a single Jupyter Notebook (`pytorch-basic-neural-networks.ipynb`) containing all components. The notebook includes sections for environment setup, tensor operations demonstration, activation functions, neural network implementation, data preparation, training, model persistence, prediction, and experiments.

3.2. Class Structure

The program consists of two main classes:

3.2.1 ActivationFunction Class

A utility class providing static methods for activation functions and their derivatives. This class uses PyTorch operations exclusively and operates on tensors.

- `sigmoid(s)`: Computes sigmoid activation function $\sigma(s) = \frac{1}{1+e^{-s}}$, output range [0, 1]
- `sigmoid_derivative(s)`: Computes derivative of sigmoid function $\sigma'(s) = s \cdot (1 - s)$ where s is typically the sigmoid output

Both methods are static and can be called without instantiating the class.

3.2.2 FFNeuralNetwork Class

The main neural network class inheriting from `nn.Module`:

Initialization (___init___):

- Sets architecture parameters: `input_size` (default: 3), `hidden_size` (default: 4), `output_size` (default: 1)
- Initializes weight matrices `W1` (inputSize $\times$ hiddenSize) and `W2` (hiddenSize $\times$ outputSize) with random values from normal distribution using `nn.Parameter`
- Allocates storage for intermediate variables: `z`, `z2`, `z3` for forward pass; `z_activation`, `z_activation_derivative`, `out_error`, `out_delta`, `z2_error`, `z2_delta` for backward pass

Table III.1: Summary of core and static methods

Method	**Input**	**Output**	**Notes**
`activation(z)`	z	Activated tensor	Sigmoid activation
`activation_derivative(z)`	z	Derivative tensor	Sigmoid derivative
`forward(X)`	Input tensor X	Network output	Input $\rightarrow$ Hidden $\rightarrow$ Output
`backward(X, y, output, lr)`	X, y, $output$, lr	Updated weights	Manual backpropagation
`train(X, y, lr)`	X, y, lr	Updated model	One training step
`predict(x_predict)`	Input sample	Predicted output	Inference only
`save_weights(model, path)`	Model, file path	Saved file	Uses state dictionary
`load_weights(path, in, hid, out)`	File path, layer sizes	New model instance	Reconstructs model

4. Implementation Structure and Execution Steps

4.1. Implementation Sections

No.	**Section Name**	**Summary (Inputs / Outputs / Properties)**
1	Import and Setup	Inputs: none. Outputs: initialized environment, fixed random seed.
2	Activation	Class properties: activation function, derivative function. Inputs: tensor x. Outputs: activated tensor and gradient tensor.
3	Loss	Class properties: loss function, loss derivative. Inputs: predicted output y_{pred}, target y. Outputs: scalar loss value and loss gradient.
4	Layers	`FCLayer` properties: weights, bias, input data, output data. `ActivationLayer` properties: activation function, activation derivative. Inputs: input tensor. Outputs: transformed tensor and propagated gradient.

No.	**Section Name**	**Summary (Inputs / Outputs / Properties)**
5	Network	Class properties: list of layers, loss function, learning rate. Inputs: training data and target values. Outputs: trained network and prediction results.
6	Training Data	Inputs: XOR input samples and corresponding labels. Outputs: formatted tensors for training and evaluation.
7	Builder Utilities	Inputs: layer sizes, activation types, input data. Outputs: constructed network instances, accuracy values, visualization figures.
8	Training and Testing	Inputs: network instance, hyperparameters. Outputs: accuracy metrics and decision boundary visualizations.
9	Save and Load Model	Inputs: trained network instance, file path. Outputs: saved model file (`xor_model.pkl`) and loaded network instance.

Table III.2: Implementation sections in the notebook

4.2. Key Implementation Details

Model Persistence: Model parameters are serialized using Python's `pickle` module. The saved state dictionary stores weights and bias values with keys following the format `layer_{i}_weights` and `layer_{i}_bias`, ensuring compatibility with the reconstructed network architecture.

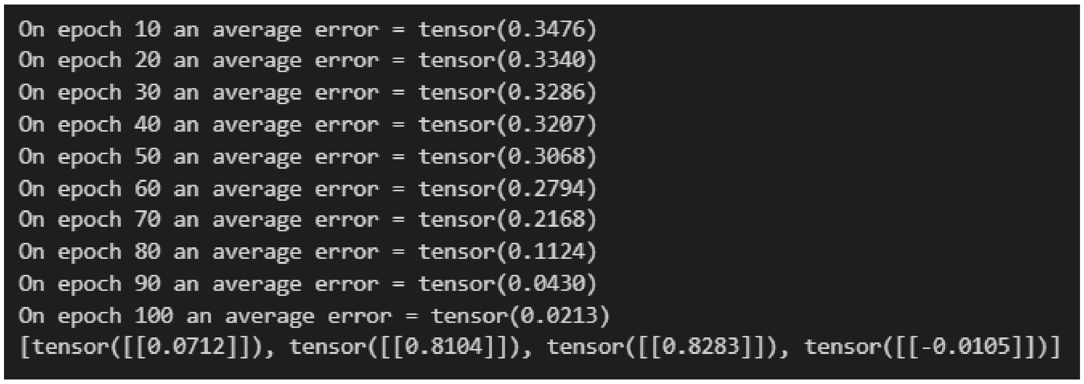
Training Process: Training is performed using manual forward and backward propagation across layers. Gradient descent is applied with a configurable learning rate, and loss values are monitored during training iterations.

Network Architecture: The network supports flexible architectures with multiple hidden layers. Layers are composed sequentially using the `add()` method, allowing different activation functions and layer depths to be evaluated experimentally.

Chapter IV: Experimental Evaluation

1. Execution Results

1.1. Training Results

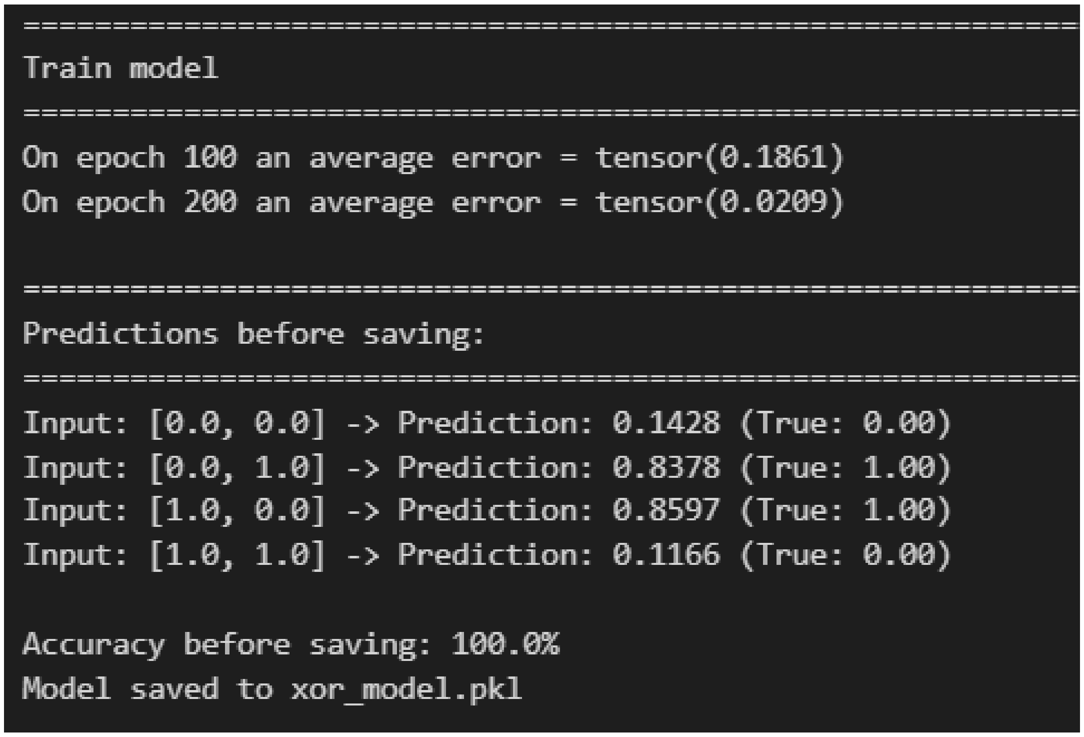


```
On epoch 10 an average error = tensor(0.3476)
On epoch 20 an average error = tensor(0.3340)
On epoch 30 an average error = tensor(0.3286)
On epoch 40 an average error = tensor(0.3207)
On epoch 50 an average error = tensor(0.3068)
On epoch 60 an average error = tensor(0.2794)
On epoch 70 an average error = tensor(0.2168)
On epoch 80 an average error = tensor(0.1124)
On epoch 90 an average error = tensor(0.0430)
On epoch 100 an average error = tensor(0.0213)
[tensor([[0.0712]]), tensor([[0.8104]]), tensor([[0.8283]]), tensor([[-0.0105]])]
```

Figure IV.1: Training loss reported during the learning process

Figure IV.1 shows that after 100 training epochs, the average training loss decreases to 0.0213. The trained network produces final outputs of approximately $[0.0712,\ 0.8104,\ 0.8283,\ -0.0105]$ for the XOR input samples, indicating successful convergence on the dataset.

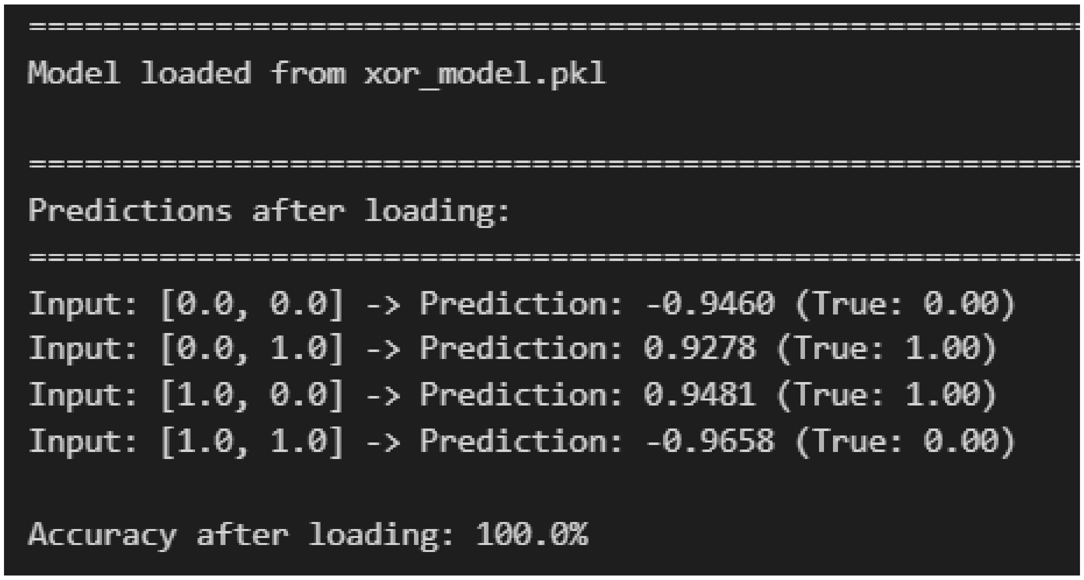
1.2. Model Saving and Loading Results



```
============================================================
Train model
============================================================
On epoch 100 an average error = tensor(0.1861)
On epoch 200 an average error = tensor(0.0209)

============================================================
Predictions before saving:
============================================================
Input: [0.0, 0.0] -> Prediction: 0.1428 (True: 0.00)
Input: [0.0, 1.0] -> Prediction: 0.8378 (True: 1.00)
Input: [1.0, 0.0] -> Prediction: 0.8597 (True: 1.00)
Input: [1.0, 1.0] -> Prediction: 0.1166 (True: 0.00)

Accuracy before saving: 100.0%
Model saved to xor_model.pkl
```

Figure IV.2: Prediction results before saving the trained model

Figure IV.2 shows the prediction outputs obtained immediately after training and before saving the model. All input samples are classified correctly, indicating that the trained parameters successfully capture the XOR decision pattern at this stage.

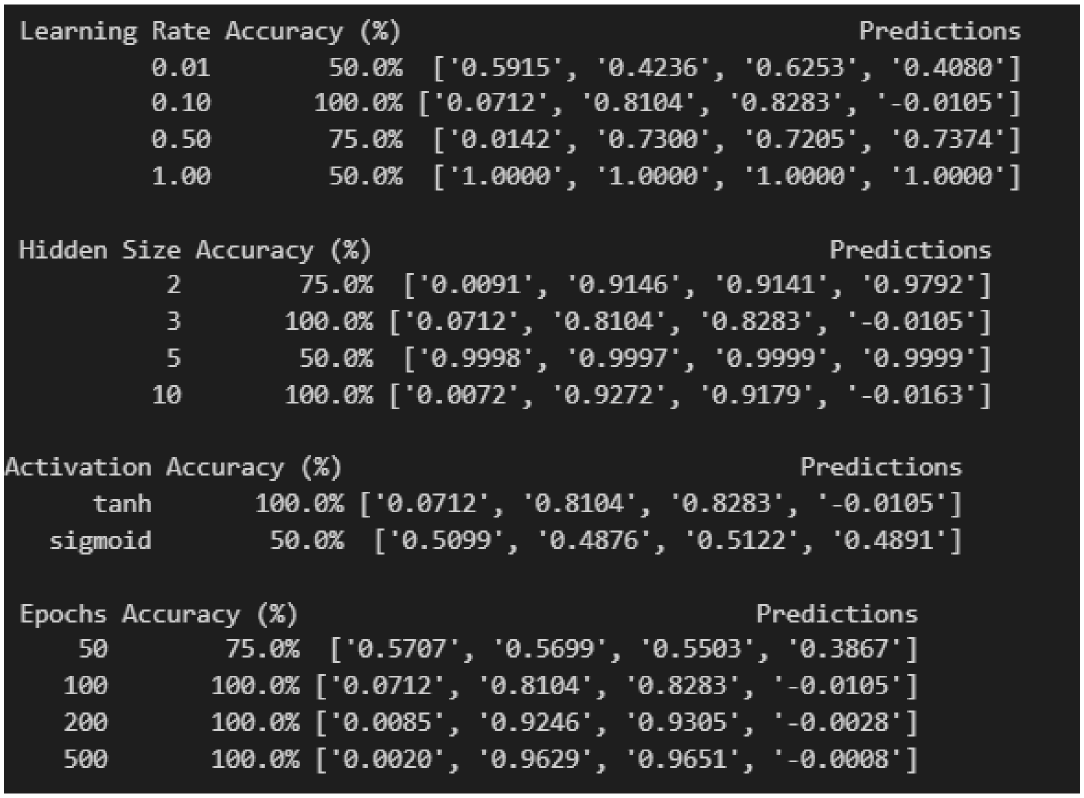


```
============================================================
Model loaded from xor_model.pkl

============================================================
Predictions after loading:
============================================================
Input: [0.0, 0.0] -> Prediction: -0.9460 (True: 0.00)
Input: [0.0, 1.0] -> Prediction: 0.9278 (True: 1.00)
Input: [1.0, 0.0] -> Prediction: 0.9481 (True: 1.00)
Input: [1.0, 1.0] -> Prediction: -0.9658 (True: 0.00)

Accuracy after loading: 100.0%
```

Figure IV.3: Prediction results after loading the trained model

Figure IV.3 presents the prediction results after reloading the saved model. The predictions remain unchanged compared to the results before saving, demonstrating that the model saving and loading mechanism preserves the learned parameters correctly and enables reliable reuse of trained models.

1.3. Hyperparameter Experiment Results

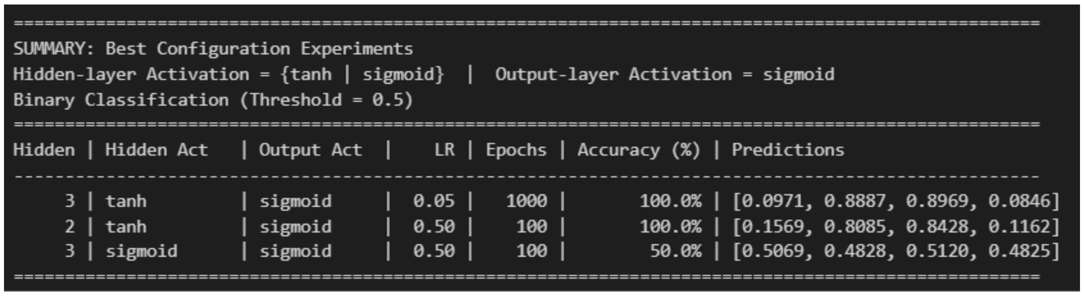


```
 Learning Rate Accuracy (%)                              Predictions
         0.01       50.0%  ['0.5915', '0.4236', '0.6253', '0.4080']
         0.10      100.0% ['0.0712', '0.8104', '0.8283', '-0.0105']
         0.50       75.0%  ['0.0142', '0.7300', '0.7205', '0.7374']
         1.00       50.0%  ['1.0000', '1.0000', '1.0000', '1.0000']

 Hidden Size Accuracy (%)                              Predictions
          2       75.0%  ['0.0091', '0.9146', '0.9141', '0.9792']
          3      100.0% ['0.0712', '0.8104', '0.8283', '-0.0105']
          5       50.0%  ['0.9998', '0.9997', '0.9999', '0.9999']
         10      100.0% ['0.0072', '0.9272', '0.9179', '-0.0163']

Activation Accuracy (%)                              Predictions
     tanh      100.0% ['0.0712', '0.8104', '0.8283', '-0.0105']
  sigmoid       50.0%  ['0.5099', '0.4876', '0.5122', '0.4891']

 Epochs Accuracy (%)                              Predictions
    50       75.0%  ['0.5707', '0.5699', '0.5503', '0.3867']
   100      100.0% ['0.0712', '0.8104', '0.8283', '-0.0105']
   200      100.0% ['0.0085', '0.9246', '0.9305', '-0.0028']
   500      100.0% ['0.0020', '0.9629', '0.9651', '-0.0008']
```

Figure IV.4: Experimental results for different learning rates, hidden sizes, activation functions, and epochs

Figure IV.4 summarizes the experimental results obtained by varying key hyperparameters, including learning rate, hidden layer size, activation function, and number of training epochs. The results show that moderate learning rates and sufficient training epochs are necessary to achieve correct classification on the XOR problem. In contrast, extreme learning rates or unsuitable activation choices lead to unstable training behavior or suboptimal accuracy.

1.4. Best Configuration Analysis

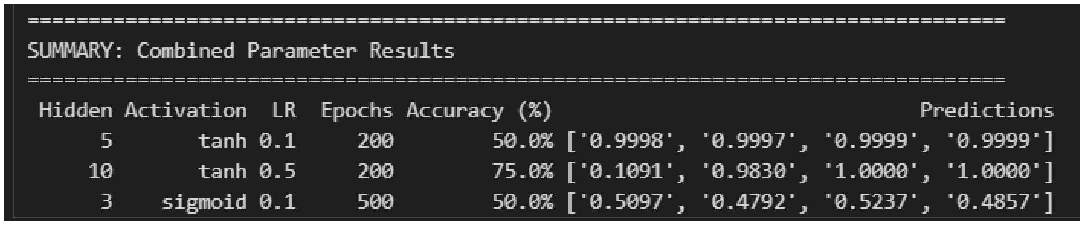


```
==========================================================================================
SUMMARY: Best Configuration Experiments
Hidden-layer Activation = {tanh | sigmoid}  |  Output-layer Activation = sigmoid
Binary Classification (Threshold = 0.5)
==========================================================================================
Hidden | Hidden Act   | Output Act  |    LR | Epochs | Accuracy (%) | Predictions
------------------------------------------------------------------------------------------
     3 | tanh         | sigmoid     |  0.05 |   1000 |       100.0% | [0.0971, 0.8887, 0.8969, 0.0846]
     2 | tanh         | sigmoid     |  0.50 |    100 |       100.0% | [0.1569, 0.8085, 0.8428, 0.1162]
     3 | sigmoid      | sigmoid     |  0.50 |    100 |        50.0% | [0.5069, 0.4828, 0.5120, 0.4825]
==========================================================================================
```

Figure IV.5: Results of selected best-performing configurations

Figure IV.5 presents the performance of selected configurations that achieve the highest classification accuracy. The results indicate that networks with a `tanh` hidden activation and a `sigmoid` output activation consistently reach perfect accuracy when trained with appropriate learning rates and sufficient epochs. This observation highlights the importance of coordinated choices between architecture and training parameters.

1.5. Combined Parameter Evaluation



```
================================================================================
SUMMARY: Combined Parameter Results
================================================================================
 Hidden Activation  LR  Epochs Accuracy (%)                             Predictions
      5       tanh 0.1     200        50.0% ['0.9998', '0.9997', '0.9999', '0.9999']
     10       tanh 0.5     200        75.0% ['0.1091', '0.9830', '1.0000', '1.0000']
      3    sigmoid 0.1     500        50.0% ['0.5097', '0.4792', '0.5237', '0.4857']
```

Figure IV.6: Performance comparison for combined hyperparameter settings

Figure IV.6 compares several combined hyperparameter settings to evaluate their joint influence on model performance. The results show that increasing model capacity or learning rate alone does not guarantee improved accuracy. Instead, balanced configurations yield more reliable convergence behavior, while overly aggressive settings may result in unstable or degenerate predictions.

2. Evaluation and Discussion

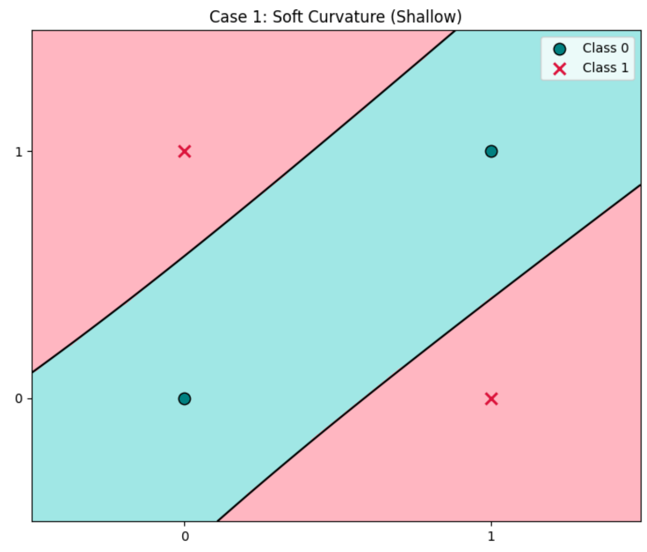
2.1. Experimental Setup

All experiments were conducted on the XOR dataset consisting of four two-dimensional input samples and their corresponding binary labels. The network was trained using sample-wise stochastic gradient descent with Mean Squared Error (MSE) as the loss function.

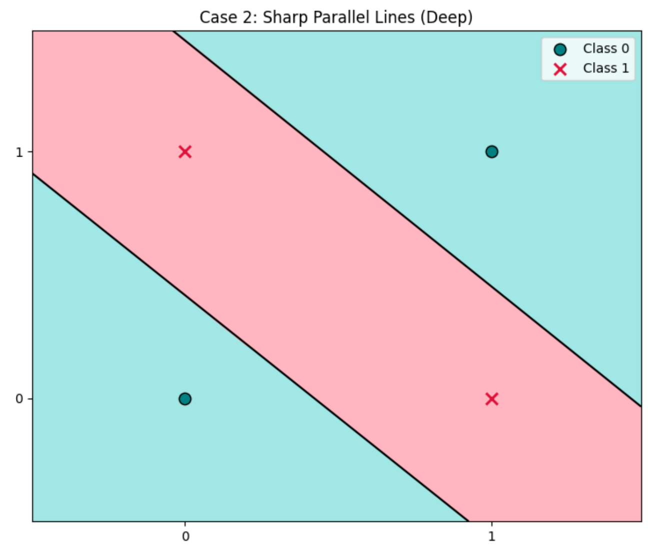
To ensure reproducibility, a fixed random seed was used for each experimental configuration. Performance was evaluated primarily through classification accuracy and qualitative inspection of decision boundaries.

2.2. Decision Boundary Analysis

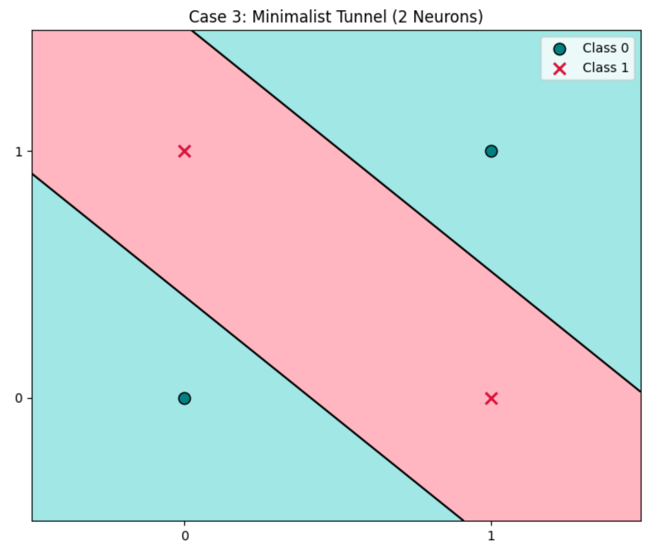
To better understand how different network configurations affect the learned representation, decision boundaries were visualized for multiple architectural cases. The following analysis examines four distinct configurations, each demonstrating unique characteristics in how the network partitions the input space.



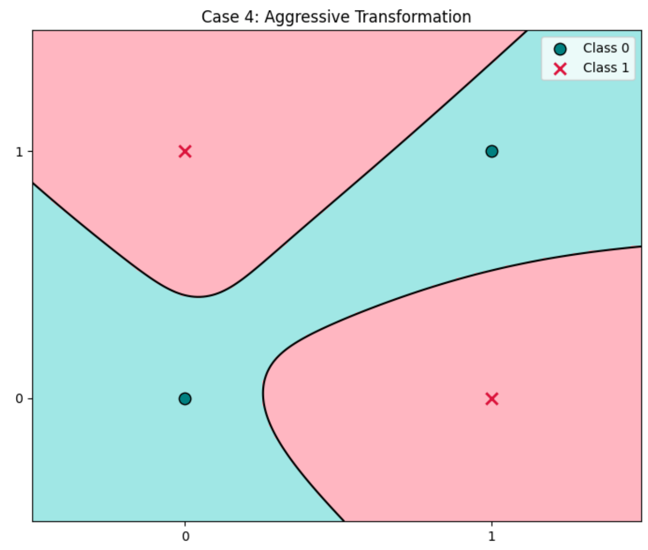
(a) Case 1: Soft Curvature (Shallow Network)
Hidden: [3], Activation: tanh, Epochs: 300, LR: 0.1



(b) Case 2: Sharp Parallel Lines (Deep Network)
Hidden: [2, 3, 3], Activation: tanh, Epochs: 1000, LR: 0.1



(c) Case 3: Minimalist Tunnel (Two Hidden Neurons)
Hidden: [2], Activation: tanh, Epochs: 2000, LR: 0.05



(d) Case 4: Aggressive Transformation
Hidden: [4, 4], Activation: tanh, Epochs: 500, LR: 0.8

Figure IV.7: Decision boundaries for four different network architectures on the XOR problem

Case 1 demonstrates a shallow network configuration with a single hidden layer of 3 neurons. The resulting decision boundary exhibits smooth, gently curved characteristics, indicating limited expressive capacity. The model successfully separates the XOR classes but relies on a broad nonlinear transformation, which may limit robustness to variations in the data.

Case 2 employs a deeper architecture with three hidden layers [2, 3, 3]. The increased depth results in a more complex transformation of the input space, producing sharper decision boundaries that form approximately parallel linear regions. This behavior indicates that deeper architectures can approximate piecewise linear decision surfaces more effectively, improving classification precision.

Case 3 investigates the minimum architectural requirement to solve the XOR problem, using only two hidden neurons in a single layer. The network forms a narrow decision tunnel separating the two classes. Although the model achieves correct classification, the margin between classes is small, making the solution sensitive to noise and parameter perturbations.

Case 4 applies a more aggressive nonlinear transformation with two hidden layers of 4 neurons each and a high learning rate of 0.8. The resulting decision boundary is highly curved and asymmetric. While the model perfectly fits the XOR data, the complexity of the boundary suggests potential overfitting and reduced generalization capability.

The experimental results reveal how architectural choices fundamentally shape decision boundaries. Shallow networks produce smooth but limited transformations, while deeper architectures enable more precise piecewise linear separations. Even minimal configurations can solve XOR, though with reduced robustness. Conversely, excessive capacity combined with high learning rates leads to complex boundaries that may compromise generalization, highlighting the importance of balanced architecture design.

2.3. Effect of Output Activation Function

In addition to architectural variations, the effect of the output activation function was examined by comparing `tanh` and `sigmoid` activations at the output layer.

2.4. Learning Behavior Analysis

To further analyze training dynamics, the mean classification accuracy was evaluated across multiple random seeds under varying numbers of epochs and learning rates.

2.4.1 Output Activation vs. Epochs

This experiment analyzes how training duration influences performance for different output activations. Sigmoid-based outputs converge faster and achieve higher final accuracy compared to tanh-based outputs.

This suggests that sigmoid activation is more suitable for binary classification tasks when paired with MSE loss in this implementation.

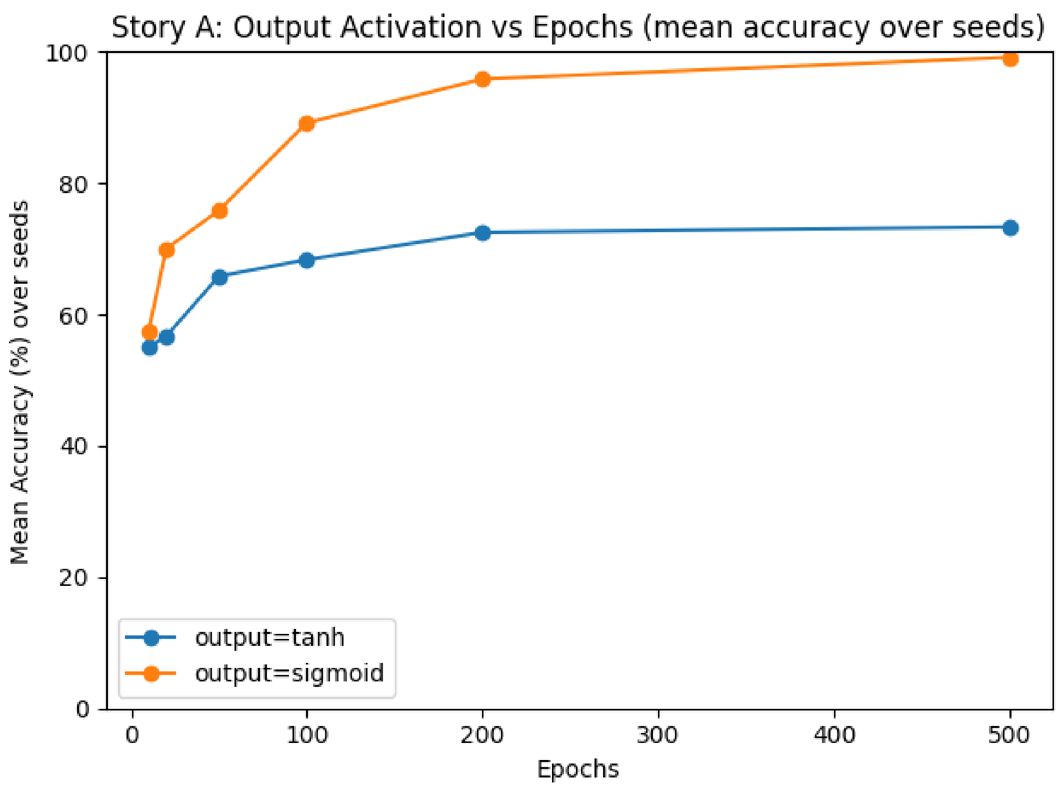


Figure IV.8: Mean Accuracy vs. Epochs for Different Output Activations

2.4.2 Output Activation vs. Learning Rate

The effect of learning rate was also evaluated. Sigmoid output activation demonstrates greater stability across a wide range of learning rates, while tanh performance degrades at higher learning rates due to gradient saturation and oscillation.

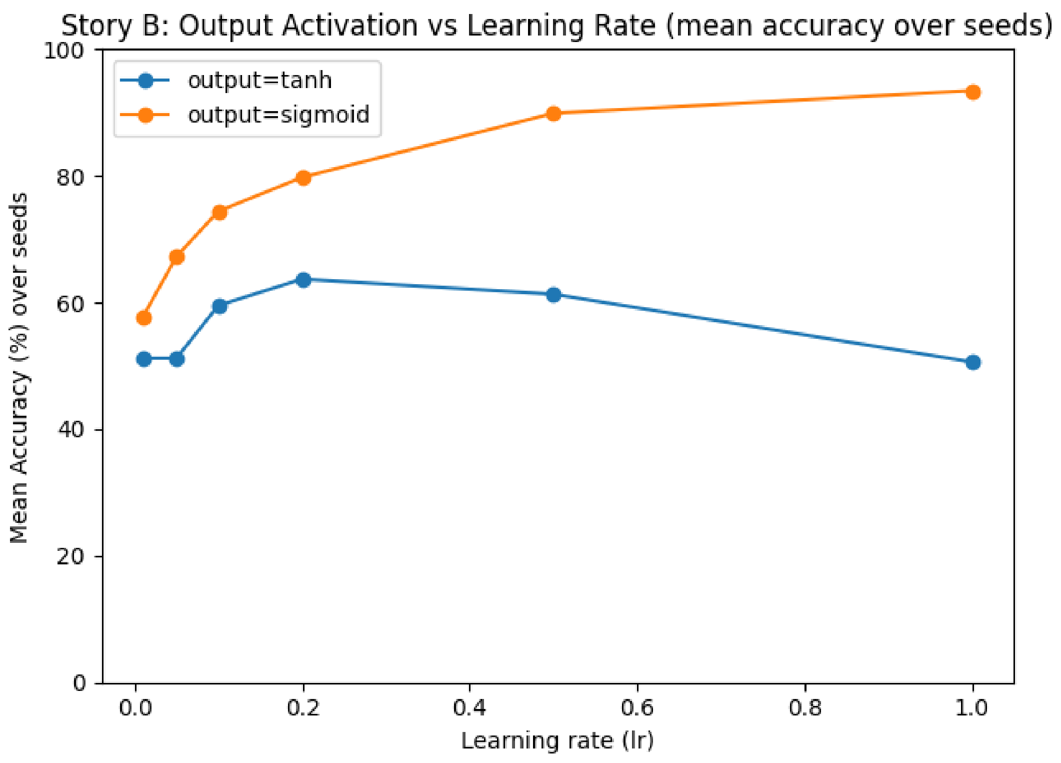


Figure IV.9: Mean Accuracy vs. Learning Rate for Different Output Activations

2.5. Overall Discussion

The experimental results highlight several key observations. First, network depth and hidden layer size significantly influence the shape and robustness of the decision boundary. Second, even minimal architectures are capable of solving the XOR problem, though with reduced margins. Finally, the choice of output activation function plays a crucial role in convergence behavior and classification stability.

These findings confirm that the manually implemented network successfully captures essential learning dynamics and provides an interpretable framework for studying neural network behavior.

Chapter V: Conclusion

1. Task Completion Summary

No.	Task	Completion
1	Implement a modular Feed Forward Neural Network architecture	100%
2	Implement manual forward propagation and backpropagation	100%
3	Train the network using sample-wise SGD with MSE loss	100%
4	Implement model saving and loading mechanisms	100%
5	Analyze decision boundaries under different architectures	100%
6	Evaluate learning behavior under different hyperparameters	100%

Table V.1: Task completion summary

2. Conclusion

This laboratory successfully implemented a Feed Forward Neural Network from basic components, including linear layers, activation layers, and loss functions, without relying on automatic differentiation. The manual implementation of forward propagation and backpropagation provides direct insight into the internal learning mechanism of neural networks.

The training process was carried out using sample-wise Stochastic Gradient Descent with Mean Squared Error loss. In addition, model saving and loading were implemented to preserve trained parameters and enable reuse of trained models for further evaluation and experimentation.

Through extensive experiments on the XOR problem, the results demonstrate that architectural choices such as network depth and hidden layer size significantly influence the learned decision boundaries. Shallow networks produce smooth but limited nonlinear transformations, while deeper architectures enable more precise partitioning of the input space.

Furthermore, the experimental analysis shows that the choice of output activation function affects learning stability and convergence behavior. Sigmoid output activation exhibits more stable performance across different training durations and learning rates compared to tanh in this implementation.

Overall, the experimental results confirm that the manually implemented network is capable of learning nonlinear decision boundaries and serves as an effective and interpretable framework for studying the impact of network architecture and training hyperparameters on learning behavior.

Reference

[1] GeeksforGeeks. Tensors in pytorch. https://www.geeksforgeeks.org/python/tensors-in-pytorch/, 2021. Retrieved: 2025-12-17.

[2] GeeksforGeeks. What is fully connected layer in deep learning, 2025. Accessed: 2025-01-01.

[3] Ly Quoc Ngoc. Pytorch #2 – practice introduction, 2023. Lecture notes and laboratory practice material.

[4] PyTorch. Pytorch: An open source deep learning framework. https://pytorch.org/projects/pytorch/, 2025. Retrieved: 2025-12-17.

[5] PyTorch Documentation Contributors. Tensors. https://docs.pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html, 2025. Retrieved: 2025-12-17.

Acknowledgment

I would like to express sincere gratitude to Prof. Dr. Ly Quoc Ngoc for his guidance and support throughout the course. The author also wishes to thank MSc. Nguyen Manh Hung and MSc. Pham Thanh Tung for their valuable assistance and support during the completion of this assignment.