VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 4: Practice with Pytorch #1

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

December 17, 2025

Contents

List of Figures

List of Tables

Chapter I: Introduction

1. Overview of the Laboratory

This laboratory introduces the fundamental concepts and basic implementation of a Feed Forward Neural Network using the PyTorch framework.

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. [3]

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. [4]

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 considered, consisting of an input layer, one hidden layer, and an output layer. [2]

Each layer is composed of neurons connected through weighted edges. The network processes information by applying linear transformations followed by nonlinear activation functions. Learning occurs by adjusting the weights to minimize the difference between predicted outputs and target values.

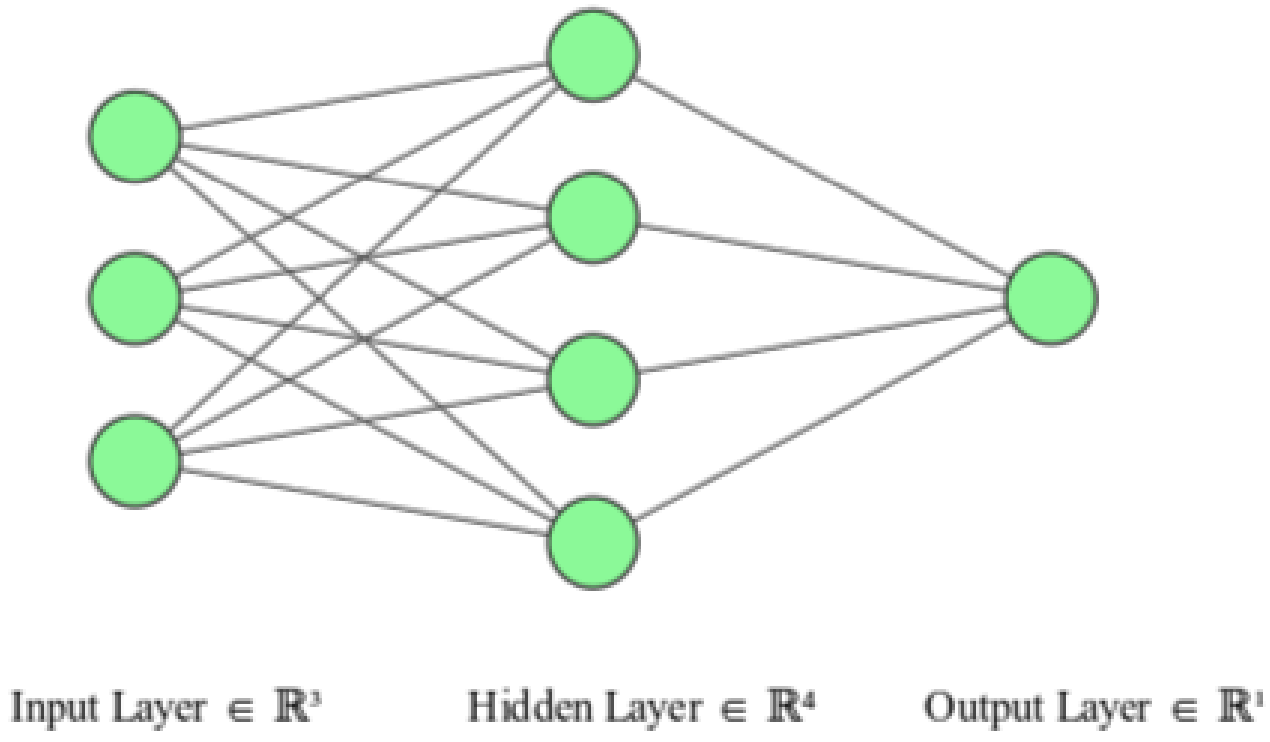

Figure I.1: Feed Forward Neural Network Architecture

Core concepts involved in the implemented network include:

- **Weight matrices**: `W1` connecting input to hidden layer, and `W2` connecting hidden to output layer

- **Activation functions**: Sigmoid function and its derivative
- **Loss function**: Mean Squared Error (MSE)
- **Forward propagation**: Computing predictions from inputs
- **Backpropagation**: Computing gradients and updating weights
- **Gradient-based optimization**: Adjusting weights using learning rate

3. Objectives

No.	Implementation Task
1	Implement the core Feed Forward Neural Network architecture.
2	Train the network and implement model saving and loading mechanisms using PyTorch's state_dict mechanism.
3	Experiment with learning rate and hidden layer size to analyze their effects on training behavior.

Table I.1: Objectives of the laboratory

4. Program Structure

Input

- Training data $X \in \mathbb{R}^{N\times 3}$ and target values $y \in \mathbb{R}^{N\times 1}$
- Prediction input $x_{\text{predict}} \in \mathbb{R}^{1\times 3}$ (normalized using training statistics)

Output

- Trained model parameters saved as `state_dict.pth`
- Predicted output value from the trained neural network

Chapter II: Implementation

1. Network Construction

The implementation uses object-oriented programming principles, with the main network class `FFNeuralNetwork` inheriting from PyTorch's `nn.Module`.

1.1. Class Structure

The `FFNeuralNetwork` class encapsulates all network components:

- **Architecture parameters**: `inputSize`, `hiddenSize`, `outputSize`
- **Weight matrices**: `W1` (input to hidden, shape `inputSize × hiddenSize`) and `W2` (hidden to output, shape `hiddenSize × outputSize`)
- **Intermediate variables**: Storage for forward pass activations (`z`, `z2`, `z3`) and backward pass gradients (`out_error`, `out_delta`, `z2_error`, `z2_delta`)

1.2. Weight Initialization

Weights are initialized using random values drawn from a normal distribution via `torch.randn()`. This provides initial random weights that are approximately centered around zero with unit variance. Bias terms are omitted in this simplified implementation to focus on the core learning mechanism.

The default architecture configuration is:

- Input size: 3 features
- Hidden layer: 4 neurons
- Output size: 1 neuron

All parameters are configurable through the class constructor, allowing experimental flexibility.

2. Activation Functions

Using the sigmoid activation function to introduce nonlinearity into the model. A utility class `ActivationFunction` provides static methods for both the activation function and its derivative.

2.1. Sigmoid Function

The sigmoid function is defined as:

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

This function maps any real-valued input to a value in the range (0, 1), making it suitable for normalization and probability estimation. In PyTorch, this is implemented using:

```
1 / (1 + torch.exp(-z))
```

2.2. Sigmoid Derivative

The derivative of the sigmoid function, required for backpropagation, is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

This derivative has a convenient property: if we already have computed the sigmoid output, we can compute its derivative using only that output value. This is computationally efficient and is used during backward propagation.

Both functions are implemented as static methods using PyTorch operations exclusively, ensuring compatibility with tensor operations and automatic differentiation.

3. Forward Propagation

Forward propagation computes the network's output given an input tensor. The process flows sequentially through the layers:

3.1. Forward Pass Steps

- **Input to Hidden Layer**:

$$z = X \cdot W_1$$

where X is the input tensor (batch_size $\times$ inputSize), W_1 is the weight matrix (inputSize $\times$ hiddenSize), and z is the intermediate result before activation.

- **Hidden Layer Activation**:

$$z_2 = \sigma(z) = \text{sigmoid}(z) \tag{II.1}$$

The sigmoid activation function is applied element-wise to z, producing the hidden layer output z_2.

- **Hidden to Output Layer**:

$$z_3 = z_2 \cdot W_2 \tag{II.2}$$

where W_2 is the weight matrix (hiddenSize $\times$ outputSize).

- **Output Activation**:

$$\text{output} = \sigma(z_3) = \text{sigmoid}(z_3) \tag{II.3}$$

The final sigmoid activation produces the network's prediction.

3.2. Implementation Details

The `forward()` method implements these steps using `torch.matmul()` for matrix multiplication. All intermediate values are stored as instance variables to facilitate backward propagation. The method returns a tensor of shape `(batch_size, outputSize)` containing predictions for all samples in the batch.

4. Backward Propagation

Backward propagation computes gradients of the loss function with respect to each weight and updates the weights accordingly. The implementation uses the mean squared error (MSE) loss

function and manually computes gradients using the chain rule.

4.1. Loss Function

The loss function used is Mean Squared Error:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{II.4}$$

where y_i is the target value and $\hat{y}_i$ is the predicted value for sample i, and N is the batch size.

4.2. Gradient Computation

The backward pass computes errors and deltas (gradients) at each layer:

- **Output Layer Error and Delta**:

$$\text{error}_{\text{out}} = y - \text{output} \tag{II.5}$$

$$\delta_{\text{out}} = \text{error}_{\text{out}} \cdot \sigma'(\text{output}) \tag{II.6}$$

- **Hidden Layer Error and Delta**:

$$\text{error}_{z2} = \delta_{\text{out}} \cdot W_2^T \tag{II.7}$$

$$\delta_{z2} = \text{error}_{z2} \cdot \sigma'(z_2) \tag{II.8}$$

- **Weight Updates**:

$$W_1 \leftarrow W_1 + \alpha \cdot X^T \cdot \delta_{z2} \tag{II.9}$$

$$W_2 \leftarrow W_2 + \alpha \cdot z_2^T \cdot \delta_{\text{out}} \tag{II.10}$$

where α is the learning rate.

4.3. Implementation Details

The `backward()` method receives the input X, target y, predicted output, and learning rate. It computes all errors and deltas, then updates both weight matrices using gradient ascent (since we're maximizing the negative loss, or equivalently, minimizing the loss). Matrix transposition is performed using `torch.t()` to align dimensions for matrix multiplication.

5. Training Process

5.1. Data Preparation

5.1.1 Training and Test Data

A small sample dataset is manually constructed to illustrate the learning behavior of a Feed Forward Neural Network. The training data consists of three input–output pairs:

$$X = \begin{bmatrix} 2 & 9 & 0 \\ 1 & 5 & 1 \\ 3 & 6 & 2 \end{bmatrix}, \quad y = \begin{bmatrix} 90 \\ 100 \\ 88 \end{bmatrix}$$

Each row of X represents one training sample, while each column corresponds to a feature. The target vector y contains the desired outputs.

A separate input sample $x_{\text{predict}} \in \mathbb{R}^{1\times 3}$ is reserved for inference after training and is not included in the training set.

5.1.2 Data Normalization

To improve numerical stability and training convergence, both input features and target values are normalized prior to training.

Input normalization. Each feature of X is normalized independently using its maximum value:

$$\max(X) = [3,\ 9,\ 2]$$

$$X_{\text{norm}} = \begin{bmatrix} 2/3 & 9/9 & 0/2 \\ 1/3 & 5/9 & 1/2 \\ 3/3 & 6/9 & 2/2 \end{bmatrix} = \begin{bmatrix} 0.67 & 1.00 & 0.00 \\ 0.33 & 0.56 & 0.50 \\ 1.00 & 0.67 & 1.00 \end{bmatrix}$$

This feature-wise normalization ensures comparable numerical ranges across input dimensions.

Target normalization. Target values are scaled by dividing by 100, assuming 100 is the maximum possible score:

$$y_{\text{norm}} = \begin{bmatrix} 0.90 \\ 1.00 \\ 0.88 \end{bmatrix}$$

5.1.3 Normalization of Prediction Input

Unlike the reference implementation, where the prediction input is normalized independently, this implementation applies a consistent strategy.

The prediction sample

$$x_{\text{predict}} = [3,\ 8,\ 4]$$

is normalized using the feature-wise maximum values derived from the training data:

$$x_{\text{predict}}^{\text{norm}} = \left[\frac{3}{3},\ \frac{8}{9},\ \frac{4}{2}\right] = [1.0,\ 0.89,\ 2.0]$$

This approach preserves consistency between training and inference. Values greater than 1 indicate that a feature exceeds the observed training range, reflecting out-of-distribution behavior.

5.2. Training Mechanism

One training epoch is encapsulated in the `train()` method, which consists of:

- Forward propagation to compute predictions
- Backward propagation to compute gradients and update weights

Training is performed over a fixed number of epochs (default: 1000). The learning rate controls

the magnitude of weight updates and is set to 0.1.

5.3. Training Loop

The training loop repeatedly executes the following steps:

- Forward pass
- Loss computation using Mean Squared Error (MSE)
- Backward pass and weight update
- Loss monitoring for convergence analysis

The process terminates when the specified number of epochs is reached or when convergence is observed.

6. Model saving and loading

6.1. State Dictionary

The model is persisted using PyTorch's `state_dict` mechanism, which stores only the learnable parameters instead of the full model object. In this implementation, the network contains two learnable weight matrices, `W1` and `W2`, defined as `nn.Parameter` objects.

At initialization, these weights are randomly sampled from a normal distribution using `torch.randn`. This provides a valid starting point for training. When a saved `state_dict` is loaded, the initial random values are fully replaced by the trained weights, ensuring consistent model restoration.

6.2. Saving Model

After training, the model parameters are extracted via `model.state_dict()`, which contains only `W1` and `W2`. This dictionary is saved to disk using `torch.save()`.

The trained weights are stored in the file `ffnn_model.pth`. Saving only the `state_dict` avoids pickle-related security risks and is compatible with PyTorch 2.6+ default settings.

6.3. Loading Model

To load the saved model, a new `FFNeuralNetwork` instance with the same architecture parameters is first created. The saved state dictionary is then loaded from `ffnn_model.pth` using `torch.load(..., weights_only=True)` and applied via `load_state_dict()`.

This approach ensures that the model architecture is explicitly defined and that only trained parameters are restored.

6.4. Prediction

After loading, the model can be used for inference through forward propagation. Input data must be normalized using the same statistics as the training data to ensure consistency between training and prediction.

Chapter III: Installation and Usage

This section describes how to set up the environment, install required libraries, and execute the program in all three supported run modes.

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 for interactive execution;
- RAM: Minimum 2GB (recommended 4GB);
- Disk space: Approximately 500MB for dependencies and model files.

1.2. Python and Library Installation

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

a) Create a Virtual Environment.

Create a virtual environment to isolate the project's 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 according to your system configuration. For CPU-only installation:

```
pip install torch torchvision torchaudio
```

For GPU support with CUDA, visit `https://pytorch.org/get-started/locally/` for the appropriate installation command.

Install NumPy:

```
pip install numpy
```

For Jupyter Notebook support:

```
pip install jupyter notebook
```

1.3. Main Libraries

The project uses the following Python libraries:

- `torch`: PyTorch core library for tensor operations and neural network construction;
- `torch.nn`: PyTorch neural network module providing base classes like `nn.Module`;
- `numpy`: NumPy library for array operations and conversion between NumPy arrays and PyTorch tensors.

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
>>> print(torch.__version__)
>>> print(np.__version__)
```

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

2. How to Run the Program

2.1. Running the Notebook

The implementation is provided as a Jupyter Notebook (`pytorch-basic-neural-networks.ipynb`) that can be executed interactively. The notebook is organized into sequential steps 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 `pytorch-basic-neural-networks.ipynb`.

b) Execute Cells Sequentially.

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

- Environment Setup - Import required libraries (`torch`, `numpy`, `torch.nn`)
- Working with Tensors - Demonstrate NumPy-PyTorch Tensor conversion and shared memory mechanism
- Define Activation Functions - Define `ActivationFunction` class with sigmoid and its derivative

- Define Feed Forward Neural Network - Implement `FFNeuralNetwork` class inheriting from `nn.Module`
- Prepare Sample Data - Create and normalize training data (X, y) and test data (x_predict)
- Training Loop - Train the network for 1000 epochs with loss printing every 100 epochs
- Save Model - Save trained model's state_dict (weights only) to file
- Loading model and Prediction - Load saved model and perform prediction on new data
- Experiments - Test different hyperparameters (learning rates, hidden layer sizes, model configurations)

2.2. Alternative: Running as Python Script

If you prefer to run the code as a Python script, you can extract the code cells from the notebook into a `.py` file. Ensure all imports are at the top and cells are executed in sequential order.

2.3. Model File

After training and saving, a file named `ffnn_model.pth` will be created in the current directory. This file contains the model's state dictionary (weights only) and can be loaded in subsequent runs without retraining.

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 × hiddenSize) and `W2` (hiddenSize × 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

Core Methods:

- `activation(z)`: Applies sigmoid activation using `ActivationFunction.sigmoid`, stores result in `self.z_activation`

- `activation_derivative(z)`: Computes sigmoid derivative using `ActivationFunction.sigmoid_derivative`, stores result in `self.z_activation_derivative`
- `forward(X)`: Performs forward propagation through two layers: Input → Hidden Layer (matrix multiplication with W1, then sigmoid activation) → Output Layer (matrix multiplication with W2, then sigmoid activation)
- `backward(X, y, output, learning_rate)`: Performs backpropagation manually: computes output layer error and delta, propagates to hidden layer, then updates weights W1 and W2 using gradient descent with specified learning rate
- `train(X, y, learning_rate)`: Training step combining forward and backward passes for one epoch
- `predict(x_predict)`: Makes predictions on new input data, prints input and output, returns predicted tensor

Static Methods:

- `save_weights(model, path)`: Saves model state_dict to file
- `load_weights(path, input_size, hidden_size, output_size)`: Loads state_dict into new model instance

Chapter IV: Experimental Evaluation

1. Execution Results

1.1. Training Results

```
Starting training...
Number of epochs: 1000
Learning rate: 0.1

Epoch #1 - Loss: 0.374021
Epoch #100 - Loss: 0.030377
Epoch #200 - Loss: 0.009770
Epoch #300 - Loss: 0.005774
Epoch #400 - Loss: 0.004424
Epoch #500 - Loss: 0.003831
Epoch #600 - Loss: 0.003528
Epoch #700 - Loss: 0.003356
Epoch #800 - Loss: 0.003249
Epoch #900 - Loss: 0.003179
Epoch #1000 - Loss: 0.003129
```

Figure IV.1: Training loss over 1000 epochs

Figure IV.1 illustrates the training process of the network over 1000 epochs with a learning rate of 0.1. The loss decreases steadily from an initial value of approximately 0.37 at epoch 1 to 0.0031 at epoch 1000, indicating stable convergence.

Loss values are reported every 100 epochs to monitor training progress, showing that the network successfully minimizes the mean squared error through iterative forward and backward

propagation.

1.2. Loading model and Prediction Results

```
Model state_dict loaded from ffnn_model.pth
Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9303]], grad_fn=<MulBackward0>)

Predicted value (scaled): 0.930308
Predicted value (original scale): 93.03
```

Figure IV.2: Prediction results after loading the trained model

Figure IV.2 shows the prediction output obtained after loading the trained model from `ffnn_model.pth`. The normalized input `[1.0000, 0.8889, 2.0000]` produces a predicted output of `0.9303`. When converted back to the original scale, the predicted value is approximately `93.03`.

This result lies within the range of the training targets and indicates that the model can generalize from the learned weights to unseen input data with similar feature distributions.

2. Evaluation and Discussion

2.1. Effect of Learning Rate

```
============================================================
Experiment 1: Testing different learning rates
============================================================
Learning rate: 0.01 - Final Loss: 0.063629
Learning rate: 0.10 - Final Loss: 0.004101
Learning rate: 0.50 - Final Loss: 0.002876
Learning rate: 1.00 - Final Loss: 0.003301
```

Figure IV.3: Final training loss for different learning rates

Figure IV.3 presents the experimental results obtained with different learning rates. When the learning rate is set to 0.01, the network converges slowly and yields a relatively high final loss (0.0636). Increasing the learning rate to 0.10 significantly improves convergence, reducing the loss to 0.0041. The best performance in this experiment is achieved with a learning rate of 0.50, resulting in the lowest final loss of 0.0029. However, further increasing the learning rate to 1.00 slightly degrades performance, suggesting that excessively large updates may reduce optimization stability.

These results indicate that an appropriately chosen learning rate is critical for efficient training. In this setup, moderate-to-high learning rates improve convergence, but overly large values provide diminishing returns.

2.2. Effect of Hidden Layer Size

```
============================================================
Experiment 2: Testing different hidden layer sizes
============================================================
Hidden size:  2 - Final Loss: 0.003681
Hidden size:  4 - Final Loss: 0.002525
Hidden size:  6 - Final Loss: 0.003753
Hidden size:  8 - Final Loss: 0.003287
Hidden size: 16 - Final Loss: 0.004180
```

Figure IV.4: Final training loss for different hidden layer sizes

Figure IV.4 shows the impact of varying the number of hidden neurons. A small hidden layer (2 neurons) results in higher loss compared to larger configurations, indicating limited representational capacity. Increasing the hidden size to 4 yields the lowest loss (0.0025), while further increasing the size to 6, 8, and 16 does not consistently improve performance.

This behavior suggests that, given the small training dataset, moderate model capacity is sufficient. Larger hidden layers do not provide clear benefits and may introduce unnecessary complexity.

2.3. Results with different configurations

Experiments are conducted with a fixed number of training epochs (10000) while varying the hidden layer size and learning rate. Table results show the effects on training loss, prediction output,

and training time.

```
================================================================
Experiment 3: Predictions with Different Model Configurations
================================================================

Test Input (normalized): [1.0, 0.8888888955116272, 2.0]
Test Input (original): [3.0, 8.0, 4.0]

================================================================
SUMMARY COMPARISON TABLE
================================================================
Hidden | LR    | Loss (MSE)   | Prediction   | Time (s)
----------------------------------------------------------------
4      | 0.10  | 0.002576     | 90.92        | 3.7827
4      | 0.50  | 0.000532     | 92.92        | 3.7331
4      | 1.00  | 0.000317     | 93.92        | 3.5541
8      | 0.10  | 0.003627     | 97.77        | 4.2277
8      | 0.50  | 0.000394     | 89.99        | 3.4562
8      | 1.00  | 0.000099     | 97.47        | 5.4964
16     | 0.10  | 0.000977     | 92.45        | 5.9478
16     | 0.50  | 0.000135     | 92.99        | 6.1445
16     | 1.00  | 0.000062     | 98.75        | 5.9753
================================================================
```

Figure IV.5: Prediction comparison across different model configurations

2.3.1 Hidden layer size = 4.

With a small hidden layer, the model achieves moderate training loss across different learning rates. Increasing the learning rate from 0.10 to 1.00 consistently reduces the loss, with the lowest loss of 0.000317 obtained at learning rate 1.00. Training time remains relatively low (approximately 3.5–3.8 seconds), indicating low computational cost for small model capacity.

2.3.2 Hidden layer size = 8.

For a larger hidden layer, the model benefits more clearly from higher learning rates. The loss decreases significantly from 0.003627 (LR = 0.10) to 0.000099 (LR = 1.00). Compared to the hidden size of 4, training time increases to around 4.2–5.5 seconds, reflecting higher computational cost due to increased model parameters.

2.3.3 Hidden layer size = 16.

When the hidden layer size is further increased to 16, the model achieves the lowest observed loss of 0.000062 at learning rate 1.00. However, this improvement comes with a noticeable increase in training time, ranging from approximately 5.9 to 6.1 seconds. This confirms that larger hidden layers lead to higher computational cost when the number of epochs is fixed.

2.4. Overall Evaluation

The configuration with 16 hidden units and a learning rate of 1.0 achieves the lowest MSE, its predicted value deviates more from the expected target.

In contrast, the configuration with 4 hidden units and a learning rate of 0.5 produces a more reasonable and stable prediction for the small test sample.

This result indicates that a lower training loss does not necessarily correspond to better practical prediction performance, especially when using a limited dataset.

Chapter V: Conclusion

1. Task Completion Summary

No.	Task	Completion
1	Implement Feed Forward Neural Network	100%
2	Manual forward and backward propagation	100%
3	Model training with loss monitoring	100%
4	Model saving and loading (`state_dict`)	100%
5	Hyperparameter configuration (Hidden layer, Learning rate)	100%
6	Result comparison (loss, prediction, training time)	100%

Table V.1: Task completion summary

2. Conclusion

This laboratory successfully completes all required tasks, including neural network implementation, training, model persistence, configuration experiments, and result comparison. The experiments show that model performance depends on the joint selection of hidden layer size and learning rate.

Due to the very small dataset, the model has a high risk of overfitting, and generalization performance cannot be reliably evaluated. In practical applications with larger datasets, data should be split into training, validation, and test sets, where validation loss is used to monitor convergence and prevent overfitting.

Reference

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

[2] Ly Quoc Ngoc. Pytorch #1 – practice introduction, 2023. Lecture notes and laboratory practice material.

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

[4] 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.