VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 6.2: PyTorch - Images Input & Visualization - Version #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 8, 2026

Contents

List of Figures

List of Tables

Chapter I: Introduction

1. Overview

In this laboratory assignment, a Feed Forward Neural Network (FFNN) is implemented using PyTorch to perform handwritten digit classification on an MNIST-like image dataset. Unlike high-level neural network abstractions, the network is constructed manually with explicit forward and backward propagation steps, allowing detailed inspection of internal computations.

The implementation uses a modular design approach where each component of the network is constructed and organized into separate modules, including fully connected layers, activation functions, loss functions, and the training procedure based on forward and backward propagation. The implementation supports image-based input, data preprocessing from raw image folders, training visualization, and model persistence through state dictionaries. Two experimental settings are considered: a reduced training sample set and the full training dataset, enabling comparison between small-scale and large-scale learning behavior.

2. Problem Statement

The problem addressed in this laboratory is the automatic prediction of handwritten digits from grayscale image data. Given an input image representing a single handwritten digit, the system is required to infer the most probable digit class among ten possible categories (0-9).

This task can be formulated as a supervised multi-class classification problem, where the model learns a discriminative mapping from high-dimensional pixel space to a discrete label space. The primary challenge lies in handling variations in writing style, stroke thickness, and local pixel intensity while preserving class-discriminative features.

Figure I.1: Sample of handwritten digit images from the dataset. [2]

2.1. Input

Let the input dataset be defined as

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N},$$

where each input sample

$$x_i \in \mathbb{R}^{784}$$

represents a flattened grayscale image of size 28×28, and

$$y_i \in \{0, 1, \ldots, 9\}$$

denotes the ground-truth digit label.

Each input vector x_i is obtained by converting an image to grayscale, resizing it to 28×28 pixels, flattening it into a one-dimensional vector, and normalizing pixel intensities using standardization (mean subtraction and variance scaling). The dataset is stored in image folder format, where images are organized in subdirectories labeled by their digit classes.

2.2. Output

The desired output of the network is a probability distribution over ten digit classes. Formally, the network produces

$$\hat{y}_i = f(x_i; \theta) \in \mathbb{R}^{10},$$

where $f(\cdot)$ denotes the Feed Forward Neural Network parameterized by θ, and

$$\sum_{k=1}^{10} \hat{y}_{i,k} = 1, \quad \hat{y}_{i,k} \geq 0.$$

The predicted class label is obtained using the argmax operator:

$$\hat{c}_i = \arg\max_k \hat{y}_{i,k}.$$

2.3. Framework

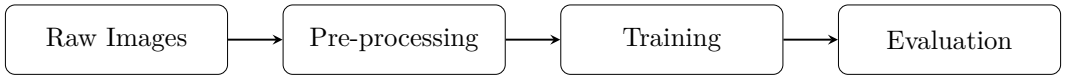

Figure I.2: High-level processing pipeline of the handwritten digit classification system.

Figure I.2 presents the overall framework of the proposed system as a linear processing pipeline.

Raw Images stage provides handwritten digit samples as grayscale images stored in folder structure. These images serve as the original data source for the learning process.

Pre-processing stage converts raw images into numerical vectors by loading images from folders, converting to grayscale, resizing to 28×28 pixels, flattening into 784-dimensional vectors, and normalizing pixel values using standardization.

Training stage, model parameters are optimized using labeled data through supervised learning with mini-batch stochastic gradient descent, with the objective of minimizing cross-entropy loss.

Evaluation stage measures the predictive performance of the trained model on unseen data using quantitative metrics such as accuracy and loss, along with visualization tools including

confusion matrices and prediction samples.

3. Objectives

No.	Objective
1	Implement a Feed Forward Neural Network with manual backpropagation in PyTorch using modular design.
2	Visualize Image and Network Architecture.
3	Load and preprocess image datasets from folder structure for training.
4	Train and evaluate the model on both sample and full datasets.
5	Visualize training loss, accuracy, and confusion matrices.
6	Save and reload trained model parameters for validation.
7	Compare the results with different hidden layer sizes, batch sizes, and normalization modes.

Table I.1: Laboratory Objectives

Chapter II: Implementation

1. Overview

The handwritten digit classification problem is formulated as a supervised multi-class classification task, where the objective is to learn a discriminative mapping from high-dimensional pixel space to discrete class labels. The input space consists of grayscale images of size 28×28 pixels, which are flattened into latent representations of dimension 784. Each latent vector $\mathbf{x} \in \mathbb{R}^{784}$ encodes the spatial intensity distribution of a handwritten digit, where pixel values are normalized using standardization (mean subtraction and variance scaling).

The classification problem aims to assign each input latent vector $\mathbf{x}_i$ to one of ten digit classes $\mathcal{C} = \{0, 1, 2, \ldots, 9\}$. Given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ where $y_i \in \mathcal{C}$ denotes the ground-truth label, the model learns a function $f : \mathbb{R}^{784} \to \mathbb{R}^{10}$ that maps input features to a probability distribution over classes. The predicted class is obtained via the argmax operation: $\hat{y}_i = \arg\max_k f(\mathbf{x}_i)_k$.

The implementation uses a modular design approach where each component of the network is constructed and organized into separate classes, enabling flexible network configuration and clear understanding of how individual components interact within a feed-forward neural network.

2. Network Construction

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

2.1. 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()`. This abstraction allows uniform treatment of different layer types in the network.

- **FCLayer**: A fully-connected (linear) layer that performs linear transformation between input and output, including both weights and bias. Each FCLayer stores weight matrix W and bias vector b, initialized with small random values from a normal distribution.
- **ActivationLayer**: An activation layer that applies a nonlinear function element-wise and computes the corresponding gradient in the backward propagation phase. This layer wraps activation functions and their derivatives, supporting sigmoid, tanh, ReLU, and softmax.
- **Network**: A class that manages the entire network, responsible for adding layers, configuring the loss function, training the model, and making predictions. It maintains a list of layers and coordinates forward and backward passes.
- **Activation**: A utility class providing static methods for activation functions (sigmoid, tanh, ReLU, softmax).
- **ActivationPrime**: A utility class providing static methods for activation function derivatives used in backpropagation.
- **Loss**: A utility class providing static methods for loss functions (MSE, cross-entropy with logits, cross-entropy with probabilities).
- **LossPrime**: A utility class providing static methods for loss function derivatives.

2.2. Network Architecture

The implemented feed-forward neural network consists of multiple layers: an input layer, one or more hidden layers, and an output layer. The architecture transforms the input latent representation through a series of linear transformations and nonlinear activations to produce class probability estimates.

For the MNIST digit classification task, the default architecture is:

- **Input layer**: Receives flattened image vectors of dimension $d_{\text{in}} = 784$, corresponding to 28×28 pixel images.

- **Hidden layers**: Contains configurable numbers of neurons. The default configuration uses two hidden layers with sizes $d_{h1} = 128$ and $d_{h2} = 64$, where each neuron applies a linear transformation followed by the ReLU activation function.
- **Output layer**: Contains $d_{\text{out}} = 10$ neurons, one for each digit class, with the Softmax activation function applied to produce normalized probability distributions.

The network architecture can be represented as: $784 \to 128 \to 64 \to 10$, where arrows indicate fully-connected layers with activation functions applied between layers.

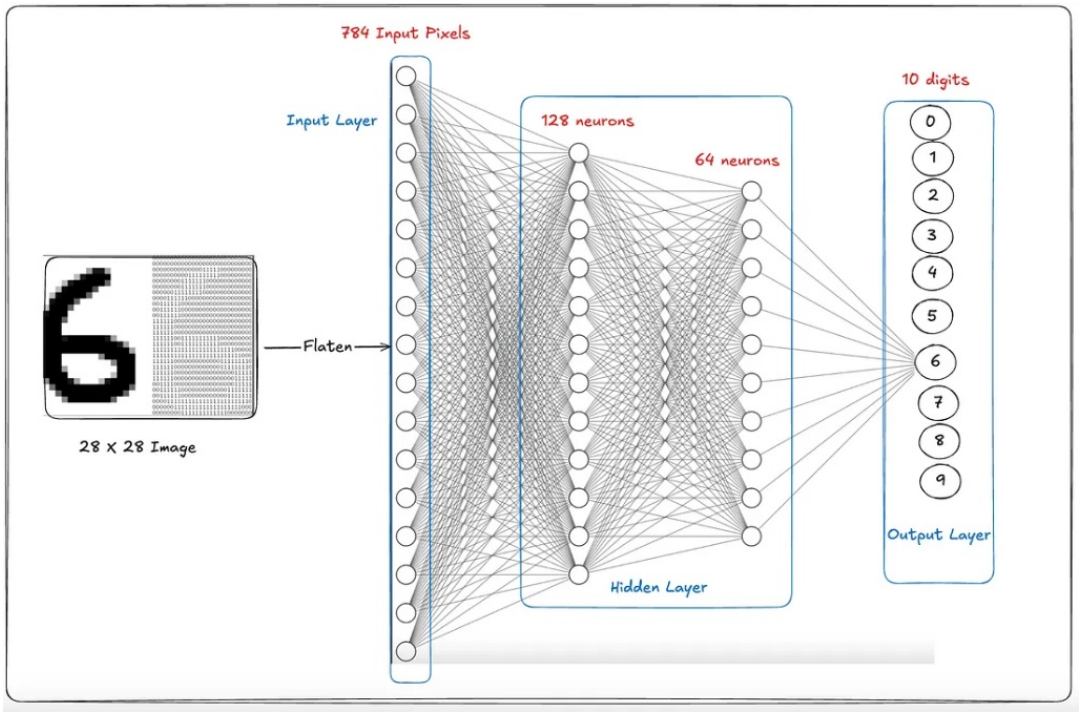

Figure II.1: FFNN architecture with 2 hidden layers for digit classification.

2.3. Parameter Initialization

The network parameters consist of weight matrices and bias vectors for each fully-connected layer. For a network with architecture $784 \to 128 \to 64 \to 10$, the parameters are:

- $\mathbf{W}_1 \in \mathbb{R}^{784\times128}$: Weight matrix connecting input to first hidden layer
- $\mathbf{b}_1 \in \mathbb{R}^{128}$: Bias vector for first hidden layer
- $\mathbf{W}_2 \in \mathbb{R}^{128\times64}$: Weight matrix connecting first to second hidden layer

- $\mathbf{b}_2 \in \mathbb{R}^{64}$: Bias vector for second hidden layer
- $\mathbf{W}_3 \in \mathbb{R}^{64\times10}$: Weight matrix connecting second hidden layer to output
- $\mathbf{b}_3 \in \mathbb{R}^{10}$: Bias vector for output layer

Initialization follows a small-variance normal distribution strategy:

$$\mathbf{W}_i \sim \mathcal{N}(0, \sigma^2), \quad \sigma = 0.01 \tag{II.1}$$

$$\mathbf{b}_i = \mathbf{0} \tag{II.2}$$

This initialization scheme ensures that initial activations remain in the linear region of the ReLU function, promoting stable gradient flow during early training stages.

3. Activation Functions

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

3.1. 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.

The derivative of sigmoid with respect to its input is:

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

3.2. 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.

The derivative of tanh is:

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

3.3. ReLU Activation Function

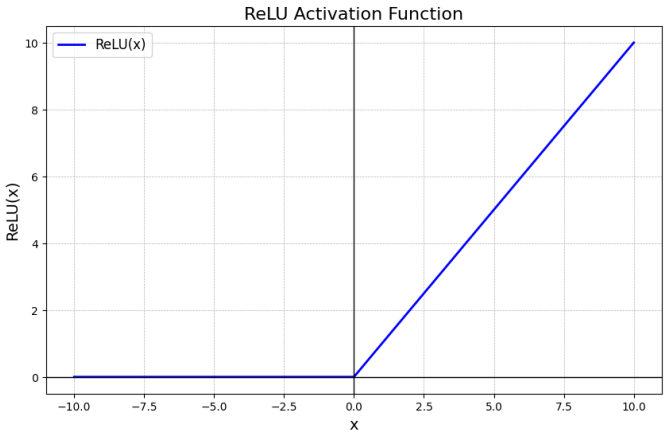

Figure II.2: ReLU activation function and its derivative. [1]

The Rectified Linear Unit (ReLU) activation function is applied element-wise to the hidden layer pre-activations. The function is defined as:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{II.3}$$

The derivative of ReLU with respect to its input is:

$$\frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{II.4}$$

3.3.1 Rationale for ReLU

ReLU is chosen for the hidden layer activation due to several advantageous properties:

- **Computational efficiency**: The function and its derivative are computationally inexpensive, involving only thresholding operations.
- **Sparsity**: ReLU naturally induces sparsity by zeroing out negative activations, effectively reducing the effective network capacity and promoting feature selectivity.
- **Gradient flow**: Unlike saturating activations such as sigmoid or tanh, ReLU avoids vanishing gradients for positive inputs, allowing deeper networks to train effectively. The gradient remains constant (equal to 1) for positive values, facilitating stable backpropagation.
- **Nonlinearity**: Despite its piecewise linear nature, the combination of multiple ReLU units enables the network to approximate complex nonlinear decision boundaries.

3.4. Softmax Activation Function

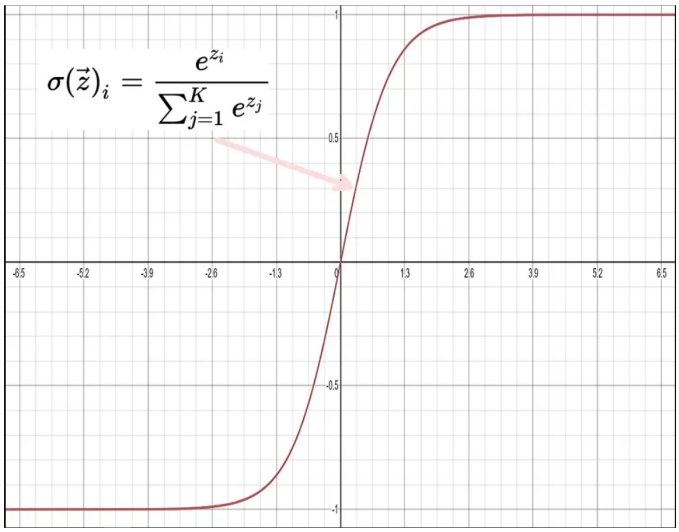

Figure II.3: Softmax activation function mapping logits to probability distribution.

The Softmax function is applied to the output layer pre-activations to produce a valid probability distribution over the ten digit classes. Given a vector $\mathbf{z} \in \mathbb{R}^{10}$ of logits, the Softmax function computes:

$$\text{Softmax}(\mathbf{z})_k = \frac{\exp(z_k - \max_j z_j)}{\sum_{j=1}^{10} \exp(z_j - \max_j z_j)} \tag{II.5}$$

where the subtraction of the maximum value ($\max_j z_j$) is performed for numerical stability, preventing overflow in the exponential computation.

The Softmax output satisfies the probability axioms:

$$\sum_{k=1}^{10} \text{Softmax}(\mathbf{z})_k = 1 \tag{II.6}$$

$$\text{Softmax}(\mathbf{z})_k \geq 0 \quad \forall k \in \{1, \ldots, 10\} \tag{II.7}$$

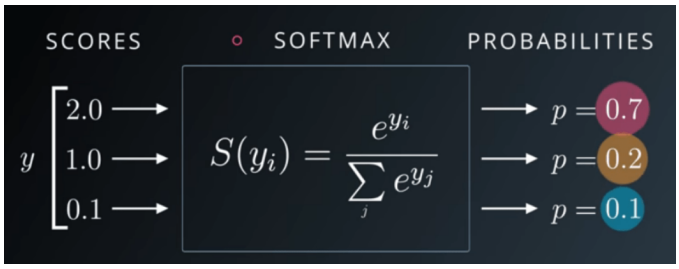

Figure II.4: Softmax example visualization. [4]

3.4.1 Rationale for Softmax

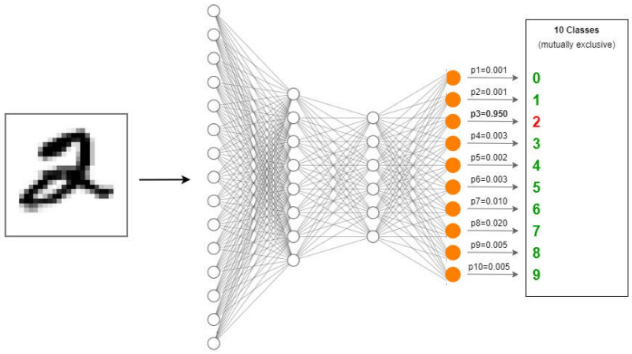

Figure II.5: Softmax architecture in multi-class classification. [3]

Softmax is the appropriate choice for the output layer in multi-class classification problems for the following reasons:

- **Probability interpretation**: The output directly represents class probabilities, enabling intuitive interpretation of model confidence. The argmax operation on Softmax outputs yields the most probable class prediction.
- **Compatibility with cross-entropy loss**: Softmax pairs naturally with the cross-entropy loss function, resulting in a simplified gradient computation during backpropagation. The gradient of the cross-entropy loss with respect to the logits reduces to the difference between predicted probabilities and one-hot encoded targets.
- **Competitive normalization**: The exponential function in Softmax creates a competitive normalization effect, where larger logit values receive exponentially higher probabilities, effectively amplifying differences between classes.
- **Differentiability**: Softmax is smooth and differentiable everywhere, ensuring stable gradient-based optimization.

3.5. Gradient Computation for Softmax

The partial derivative of Softmax with respect to its input logits is required for backpropagation. For a Softmax output $\mathbf{p} = \text{Softmax}(\mathbf{z})$, when receiving gradient $\frac{\partial L}{\partial \mathbf{p}}$ from the loss function, the gradient with respect to logits is computed as:

$$\frac{\partial L}{\partial z_k} = \sum_j \frac{\partial L}{\partial p_j} \frac{\partial p_j}{\partial z_k} = p_k \left(\frac{\partial L}{\partial p_k} - \sum_j \frac{\partial L}{\partial p_j} p_j \right) \tag{II.8}$$

This formulation is implemented in the `ActivationPrime.softmax_derivative()` method, which takes both the pre-activation logits and the output error gradient as arguments.

4. 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. Each layer stores intermediate values (pre-activations and activations) for use in the backward propagation

process.

4.1. Forward Pass Computation

Given an input batch $\mathbf{X} \in \mathbb{R}^{N \times 784}$ containing N samples, the forward propagation proceeds through the following steps for a network with architecture $784 \to 128 \to 64 \to 10$:

4.1.1 Input to First Hidden Layer

The first linear transformation computes pre-activations for the first hidden layer:

$$\mathbf{Z}_1 = \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1 \tag{II.9}$$

where $\mathbf{Z}_1 \in \mathbb{R}^{N \times 128}$ contains the pre-activation values. The ReLU activation is then applied element-wise:

$$\mathbf{H}_1 = \text{ReLU}(\mathbf{Z}_1) = \max(\mathbf{0}, \mathbf{Z}_1) \tag{II.10}$$

yielding the first hidden layer activations $\mathbf{H}_1 \in \mathbb{R}^{N \times 128}$.

4.1.2 First to Second Hidden Layer

The first hidden activations are transformed through the second linear layer:

$$\mathbf{Z}_2 = \mathbf{H}_1\mathbf{W}_2 + \mathbf{b}_2 \tag{II.11}$$

where $\mathbf{Z}_2 \in \mathbb{R}^{N \times 64}$ contains the pre-activation values. ReLU activation is applied again:

$$\mathbf{H}_2 = \text{ReLU}(\mathbf{Z}_2) = \max(\mathbf{0}, \mathbf{Z}_2) \tag{II.12}$$

yielding the second hidden layer activations $\mathbf{H}_2 \in \mathbb{R}^{N \times 64}$.

4.1.3 Second Hidden to Output Layer

The second hidden activations are transformed through the output linear layer:

$$\mathbf{Z}_3 = \mathbf{H}_2\mathbf{W}_3 + \mathbf{b}_3 \tag{II.13}$$

where $\mathbf{Z}_3 \in \mathbb{R}^{N\times 10}$ contains the output logits. The Softmax function is applied row-wise to produce the final probability distribution:

$$\hat{\mathbf{Y}} = \text{Softmax}(\mathbf{Z}_3) \tag{II.14}$$

where $\hat{\mathbf{Y}} \in \mathbb{R}^{N\times 10}$ and each row sums to unity.

4.2. Forward Pass Steps

Forward propagation is performed sequentially according to the following steps:

1. Receive input data from the training set (batched or individual samples).
2. Pass data through each fully-connected layer, computing linear transformation $Z = XW + b$.
3. Apply the corresponding activation function at each activation layer (ReLU for hidden layers, Softmax for output layer).
4. Store intermediate values (pre-activations and activations) in each layer for use in backpropagation.
5. Obtain the final output of the network as probability distributions over classes.

This implementation allows the network to support any number of hidden layers, not limited to a fixed architecture. The modular design enables flexible network configuration through the `Network.add()` method.

4.3. 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.

For the `FCLayer`, the forward pass computes:

$$\text{out\_data} = \text{in\_data} \times \text{weights} + \text{bias} \tag{II.15}$$

For the `ActivationLayer`, the forward pass applies the activation function element-wise:

$$\text{out\_data} = \text{activation}(\text{in\_data}) \tag{II.16}$$

The `Network.predict()` method performs forward propagation for inference, while the training loop calls forward propagation for each batch during the training process.

5. 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 cross-entropy loss function (with softmax probabilities), defined as:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i,c_i}),$$

where $c_i \in \{0, 1, \ldots, 9\}$ denotes the ground-truth integer class label for sample i, $\hat{y}_{i,c_i}$ is the predicted probability for the true class, and N is the batch size.

5.1. Gradient Computation

The backpropagation process is implemented manually and follows the reverse order of the forward pass. For each training batch, 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 cross-entropy loss with respect to the predicted probabilities is given by:

$$\frac{\partial L}{\partial \hat{y}_{i,k}} = \begin{cases} -\frac{1}{N\cdot\hat{y}_{i,c_i}} & \text{if } k = c_i \\ 0 & \text{otherwise} \end{cases}$$

However, when using softmax with cross-entropy, it is more efficient to compute the gradient

directly with respect to the logits. The gradient with respect to logits $\mathbf{z}_3$ is:

$$\frac{\partial L}{\partial \mathbf{z}_3} = \frac{1}{N}(\hat{\mathbf{Y}} - \mathbf{Y}_{\text{one-hot}})$$

where $\mathbf{Y}_{\text{one-hot}}$ is the one-hot encoded version of the integer labels. This elegant form arises from the combination of cross-entropy loss and Softmax activation.

Backpropagation through the softmax activation layer. The softmax derivative computation is handled specially in the `ActivationLayer.backward()` method. When the activation is softmax, the derivative function receives both the pre-activation logits and the output error gradient:

$$\frac{\partial L}{\partial \mathbf{z}_3} = \text{softmax\_derivative}(\mathbf{z}_3, \frac{\partial L}{\partial \hat{\mathbf{Y}}})$$

The softmax derivative computes:

$$\frac{\partial L}{\partial z_{3,k}} = p_k \left(\frac{\partial L}{\partial \hat{y}_k} - \sum_j \frac{\partial L}{\partial \hat{y}_j} p_j \right)$$

where $p_k = \text{Softmax}(\mathbf{z}_3)_k$.

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

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

where $a = \text{ReLU}(z)$ is the activation output, $\odot$ denotes element-wise multiplication, and $\text{ReLU}'(z) = 1$ if $z > 0$, else 0.

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:

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

$$\frac{\partial L}{\partial b} = \sum \delta \quad \text{(sum over batch dimension)},$$

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

Parameter update using mini-batch SGD. The network parameters are updated using Stochastic Gradient Descent (SGD) with mini-batches. 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 mini-batches, the gradients are averaged over the batch size N, and the bias gradient is computed as a sum over the batch dimension.

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. The `Network.fit()` method coordinates this process by calling `backward()` on each layer in reverse order.

5.2. Implementation Details

The `backward()` method of each layer receives the output error gradient and learning rate as arguments. For `FCLayer`, it computes parameter gradients, updates weights and biases, and returns the input error gradient. For `ActivationLayer`, it computes the gradient through the activation function and returns the pre-activation error gradient.

The special handling for softmax derivative requires passing both the pre-activation logits and the output error gradient, which is implemented in the `ActivationPrime.softmax_derivative()`

method.

6. Training Process

6.1. Input Data

The training dataset used in this practice is the MNIST-like handwritten digit dataset. Images are stored in folder structure, where each subdirectory is labeled by digit class (0-9). Each image is converted to grayscale, resized to 28×28 pixels, and flattened into a 784-dimensional vector.

The input–output pairs are defined as:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N},$$

where $x_i \in \mathbb{R}^{784}$ represents a flattened and normalized image, and $y_i \in \{0, 1, \ldots, 9\}$ denotes the ground-truth digit label.

In the implementation, the input data is stored as a tensor of shape $(N, 784)$, and the target labels are stored as a tensor of shape $(N,)$ with integer class labels. All values are represented using floating-point tensors for inputs and integer tensors for labels.

6.2. Data Preprocessing

6.2.1 Image Loading and Preprocessing

Raw grayscale images are loaded from folder structure and preprocessed through the following steps:

1. **Image loading**: Images are loaded from subdirectories organized by digit class.
2. **Grayscale conversion**: Images are converted to grayscale if they are in color format.
3. **Resizing**: Images are resized to 28×28 pixels to match MNIST format.
4. **Flattening**: Each 28×28 image is flattened into a 784-dimensional vector.

5. **Normalization**: Pixel values are normalized using standardization:

$$x_{i,j}^{\text{std}} = \frac{x_{i,j} - \mu_j}{\sigma_j}$$

where μ_j and σ_j are the mean and standard deviation of pixel j computed over the training set.

6.2.2 Train-Validation Split

The dataset is split into training and validation sets. A fixed 20% of the samples is randomly selected as the validation set, while the remaining data is used for training. The split is controlled by a fixed random seed to ensure reproducibility across experiments.

6.3. Training Procedure

Training is performed using mini-batch Stochastic Gradient Descent (SGD). During training, the dataset is divided into mini-batches of configurable size (default: 64 samples per batch).

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

1. Forward propagation through all layers to compute the predicted probability distributions.
2. Loss computation using the cross-entropy loss function.
3. Backward propagation through all layers to compute gradients.
4. Parameter update using SGD with the specified learning rate.

After all batches have been processed in an epoch, the model is evaluated on both training and validation sets. Loss values and accuracy metrics are computed and stored in the training history.

6.4. Training Configuration

The training process is controlled by several hyperparameters:

- **Learning rate** (α): Controls the step size in parameter updates (default: 0.05).

- **Batch size**: Number of samples processed together in each update (default: 64).
- **Number of epochs**: Total number of complete passes through the training dataset (default: 200 for sample set, 100 for full set).
- **Hidden layer sizes**: Configurable architecture, default $[128, 64]$.
- **Activation function**: ReLU for hidden layers, Softmax for output layer.
- **Normalization mode**: Standardization is applied to input features.

6.5. Validation and Monitoring

After each epoch (or at specified intervals), the model performance is evaluated on the validation set. Key metrics tracked include:

- **Training loss**: Cross-entropy loss on training set.
- **Validation loss**: Cross-entropy loss on validation set.
- **Training accuracy**: Percentage of correctly classified samples in training set.
- **Validation accuracy**: Percentage of correctly classified samples in validation set.

These metrics are plotted over epochs to visualize training progress and detect overfitting. A large gap between training and validation accuracy may indicate overfitting, while consistent improvement in both metrics indicates effective learning.

6.6. Training History

The training history is stored in a dictionary containing lists of metrics at each evaluation point:

- `epochs`: List of epoch numbers where metrics were recorded.
- `train_loss`: List of training loss values.
- `val_loss`: List of validation loss values.
- `train_acc`: List of training accuracy values.

- `val_acc`: List of validation accuracy values.

Metrics are recorded at the first epoch, last epoch, and at regular intervals (e.g., every 10 epochs) to reduce storage requirements while maintaining visibility into training progress.

7. Model Persistence

Model persistence enables saving trained network parameters to disk and reloading them for inference or continued training. This capability is essential for deploying models and reproducing experimental results.

7.1. State Dictionary Representation

The model state is represented as a dictionary containing all learnable parameters from fully-connected layers. 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 index of the FCLayer in the network (counting only FC layers, not activation layers).
- `load_state_dict(state_dict)`: Loads weights and biases from a state dictionary into the network, with validation to ensure the structure matches. Raises `ValueError` if the number of layers or keys don't match.

For a network with architecture $784 \to 128 \to 64 \to 10$, the state dictionary includes:

- `layer_0_weights`: $\mathbf{W}_1 \in \mathbb{R}^{784\times128}$ (input-to-first-hidden weight matrix)
- `layer_0_bias`: $\mathbf{b}_1 \in \mathbb{R}^{128}$ (first hidden layer bias vector)
- `layer_1_weights`: $\mathbf{W}_2 \in \mathbb{R}^{128\times64}$ (first-to-second-hidden weight matrix)
- `layer_1_bias`: $\mathbf{b}_2 \in \mathbb{R}^{64}$ (second hidden layer bias vector)
- `layer_2_weights`: $\mathbf{W}_3 \in \mathbb{R}^{64\times10}$ (second-hidden-to-output weight matrix)
- `layer_2_bias`: $\mathbf{b}_3 \in \mathbb{R}^{10}$ (output layer bias vector)

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.

7.2. 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. Extracts the network architecture information from FCLayers (list of layer sizes: $[input\_size, hidden1, hidden2, \ldots, output\_size]$)
3. Combines architecture and state dictionary into a single dictionary:
   - `architecture`: Network structure information (e.g., $[784, 128, 64, 10]$)
   - `state_dict`: Trained weights and biases
4. Serializes the dictionary to disk using Python's `pickle` module
5. Saves the file with the specified path (e.g., `custom_mnist_net.pkl`)

The implementation uses `pickle.dump()` to write the combined dictionary to a binary file. This approach stores both the model parameters and architecture information, enabling automatic network reconstruction during loading.

7.3. Loading Model

To load a saved model, the static method `Network.load(filepath, network=None)` is used:

1. The saved data is loaded from the file using `pickle.load()`
2. If the saved format includes `architecture` (new format), a new network is automatically created:
   - Network architecture is reconstructed from the saved architecture list
   - FCLayers and ActivationLayers are added in the correct sequence

- Default activation (tanh) and loss function (MSE) are configured

3. If the saved format only contains `state_dict` (old format), a network instance must be provided with matching architecture
4. The state dictionary is applied to the network instance via `network.load_state_dict()`
5. 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

After loading, the network produces identical predictions to those obtained before saving, given the same input data and architecture configuration.

7.4. Checkpoint Saving and Loading

For training workflows, a checkpoint system is implemented that saves both model parameters and configuration:

- `save_checkpoint(path, network, cfg)`: Saves network state dictionary together with configuration dictionary to a checkpoint file.
- `load_checkpoint(path)`: Loads checkpoint and automatically rebuilds the network using the saved configuration, returning both the network and configuration.

This approach enables complete model restoration including hyperparameters and training configuration, facilitating experiment reproducibility and model deployment.

7.5. 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 using `Network.fit()`
2. **Save the model**: Call `net.save("custom_mnist_net.pkl")` to persist the trained weights and architecture
3. **Load the model**: Call `Network.load("custom_mnist_net.pkl")` to automatically reconstruct and restore the network
4. **Use for inference**: Call `net.predict(X)` to make predictions on new data

Alternatively, for checkpoint-based workflows:

1. **Save checkpoint**: Call `save_checkpoint("checkpoint.pkl", net, config)` after training
2. **Load checkpoint**: Call `loaded = load_checkpoint("checkpoint.pkl")` to get network and config
3. **Verify**: Test the loaded model on validation data to confirm identical predictions

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 chapter describes the implementation of a custom feed-forward neural network for MNIST digit classification using PyTorch. The implementation includes activation functions, loss functions, layer architectures, and a complete training pipeline. This section covers environment setup, dataset preparation, implementation structure, and usage instructions.

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. Dataset Download

The MNIST dataset in JPG format is required for training and evaluation. The dataset is available at https://www.kaggle.com/datasets/scolianni/mnistasjpg.

2.1. Method 1: Direct Download from Kaggle

1. Navigate to the dataset page on Kaggle
2. Click the "Download" button to download the dataset archive
3. Extract the archive to the project directory
4. Rename the extracted folder to `mnistasjpg_data`

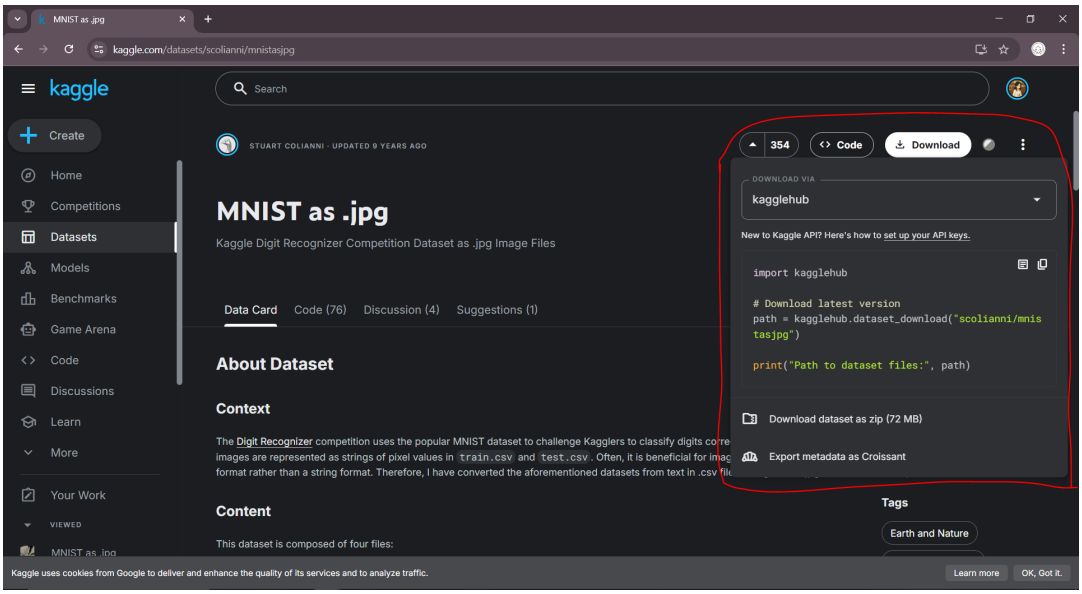

Figure III.1: Kaggle dataset download page for MNIST as JPG.

2.2. Method 2: Programmatic Download via Kaggle API

Alternatively, uncomment and execute the following code cells in the notebook:

Step 1: Install kagglehub library

```
# %pip install kagglehub
```

Step 2: Download dataset

```
# import kagglehub
# path = kagglehub.dataset_download("scolianni/mnistasjpg")
# print("Path to dataset files:", path)
```

Step 3: Move dataset to target directory

```
# import shutil
# import os
# target_path = os.path.join(os.getcwd(), "mnistasjpg_data")
# if not os.path.exists(target_path):
#     shutil.move(path, target_path)
#     print(f"Dataset moved to: {target_path}")
```

2.3. Dataset Structure

After download, the dataset directory `mnistasjpg_data` contains the following structure:

- `trainingSample/`: Reduced training set (600 samples)
- `trainingSet/`: Full training set (42,000 samples)

Each split folder contains subdirectories labeled 0-9, where each subdirectory contains JPG images of the corresponding digit class.

3. Implementation Structure

This section provides an overview of the core components implemented in the neural network framework.

3.1. Core Classes

The implementation consists of several key classes organized into functional modules.

3.1.1 Activation Functions

The `Activation` class provides static methods for non-linear activation functions used in neural network layers.

Table III.1: Activation Class Methods

Method	Description	Output
`relu(x)`	Computes Rectified Linear Unit activation: $\max(0, x)$	`torch.Tensor`
`softmax(x)`	Computes stable softmax activation for multi-class classification. Input must be 2D tensor of shape (N, C)	`torch.Tensor`

3.1.2 Activation Derivatives

The `ActivationPrime` class provides derivatives of activation functions required for back-propagation.

Table III.2: ActivationPrime Class Methods

Method	Description	Output
`relu_derivative(z)`	Computes derivative of ReLU w.r.t. pre-activation z	`torch.Tensor`
`softmax_derivative(z, out_error)`	Computes gradient through softmax layer. Requires pre-activation z and output error gradient	`torch.Tensor`

3.1.3 Loss Functions

The `Loss` class implements loss functions for training neural networks.

Table III.3: Loss Class Methods

Method	Description	Input	Output
`cross_entropy_with_logits (y_true_int, logits)`	Computes cross-entropy loss from raw logits using stable formulation	`y_true_int: torch.Tensor (N,), logits: torch.Tensor (N, C)`	`torch.Tensor` (scalar)
`cross_entropy (y_true_int, probs)`	Computes cross-entropy loss from probability tensor	`y_true_int: torch.Tensor (N,), probs: torch.Tensor (N, C)`	`torch.Tensor` (scalar)

3.1.4 Loss Derivatives

The `LossPrime` class provides derivatives of loss functions for gradient computation.

Table III.4: LossPrime Class Methods

Method	Description	Input	Output
`cross_entropy_with_logits_prime (y_true_int, logits)`	Computes derivative of cross-entropy w.r.t. logits	`y_true_int: torch.Tensor (N,), logits: torch.Tensor (N, C)`	`torch.Tensor` (N, C)
`cross_entropy_prime (y_true_int, probs)`	Computes derivative of cross-entropy w.r.t. probabilities	`y_true_int: torch.Tensor (N,), probs: torch.Tensor (N, C)`	`torch.Tensor` (N, C)

3.1.5 Base Layer

The `BaseLayer` abstract class defines the interface for all neural network layers.

Table III.5: BaseLayer Class Properties and Methods

Property/Method	Description
`forward(in_data)`	Abstract method for forward propagation. Must be implemented by subclasses
`backward(out_error, rate)`	Abstract method for backward propagation and parameter updates. Must be implemented by subclasses

3.1.6 Fully-Connected Layer

The `FCLayer` class implements a fully-connected (linear) layer with weights and bias.

Table III.6: FCLayer Class Properties and Methods

Property/Method	**Description**
`weights`	Weight matrix of shape (in_size, out_size), initialized with small random values
`bias`	Bias vector of shape (1, out_size), initialized to zeros
`in_data`	Cached input tensor from forward pass
`out_data`	Cached output tensor from forward pass
`__init__(in_size, out_size, init_std)`	Initializes layer with input/output dimensions and weight initialization standard deviation
`forward(in_data)`	Computes linear transformation: $out = in \times W + b$
`backward(out_error, rate)`	Performs backpropagation, computes gradients, and updates weights/bias using SGD

3.1.7 Activation Layer

The `ActivationLayer` class applies non-linear activation functions element-wise.

Table III.7: ActivationLayer Class Properties and Methods

Property/Method	**Description**
`activation`	Activation function callable (e.g., sigmoid, tanh, relu, softmax)
`activation_derivative`	Derivative function callable for backpropagation
`in_data`	Cached pre-activation tensor from forward pass
`out_data`	Cached activated tensor from forward pass
`__init__(activation, activation_derivative)`	Initializes layer with activation function and its derivative
`forward(in_data)`	Applies activation function to input tensor
`backward(out_error, rate)`	Computes gradient through activation function (rate parameter unused)

3.1.8 Network Class

The `Network` class is a sequential container for neural network layers with training and inference capabilities.

Table III.8: Network Class Properties and Methods

Property/Method	**Description**
`layers`	List of layer objects in sequential order
`loss`	Loss function callable
`loss_prime`	Loss function derivative callable
`__init__()`	Initializes empty network with no layers
`add(layer)`	Adds a layer to the network
`use(loss, loss_prime)`	Sets loss function and its derivative for training
`predict(data)`	Forward propagates a single sample through all layers
`predicts(data)`	Forward propagates multiple samples, returns list of outputs
`fit(x_train, y_train, epochs, alpha, print_every)`	Trains network using sample-wise SGD with specified epochs and learning rate
`state_dict()`	Returns dictionary containing all trainable parameters (weights and biases)
`load_state_dict(state_dict)`	Loads weights and biases from state dictionary
`save(filepath)`	Saves network state (architecture and weights) to file using pickle
`load(filepath, network)`	Static method that loads network from file and automatically reconstructs architecture

3.2. Utility Functions

The implementation includes utility functions for data loading, network building, training, and evaluation.

Table III.9: Data Loading and Network Building Functions

Function	Description	Input	Output
`load_mnist_from_image_folders(cfg)`	Loads MNIST images from folder structure, performs train/val split, normalization, and returns tensors	`cfg: dict` with dataset config	`dict` with X_train, y_train, X_val, y_val, mean, std, paths
`build_network_mnist(cfg)`	Builds MNIST classifier network with configurable hidden layers and activations	`cfg: dict` with model config	`Network` instance
`get_activation_functions(name)`	Maps activation name to (activation, derivative) tuple	`name: str` ("relu", "tanh", "sigmoid", "softmax")	`tuple` of callables

Table III.10: Training and Evaluation Functions

Function	Description	Input	Output
`train_digit_classifier(cfg)`	Complete training pipeline for digit classification using mini-batch SGD	`cfg: dict` with training config	`dict` with network, history, data
`train_and_evaluate(cfg, checkpoint_path, single_image_path)`	Complete pipeline: load data, train, evaluate, visualize, and save checkpoint	`cfg: dict`, optional checkpoint/image paths	`dict` with network, history, data, results

Continued on next page

Table III.10 – continued from previous page

Function	Description	Input	Output
`accuracy_from_probs(probs, y_true_int)`	Computes classification accuracy from probability tensor and integer labels	`probs: torch.Tensor (N, C), y_true_int: torch.Tensor (N,)`	`float` accuracy
`plot_history(history)`	Plots training curves (loss and accuracy) over epochs	`history: dict` with training metrics	`None` (displays plots)
`confusion_matrix_np(y_true, y_pred, num_classes)`	Computes confusion matrix for classification evaluation	`y_true, y_pred: np.ndarray (N,), num_classes: int`	`np.ndarray` (C, C)
`show_confusion_matrix(cm, title)`	Visualizes confusion matrix as heatmap	`cm: np.ndarray (C, C), title: str`	`None` (displays plot)
`describe_network(net)`	Extracts and prints FC-layer architecture	`net: Network`	`list[int]` architecture
`visualize_network_structure(net, title)`	Visualizes network architecture as flow diagram	`net: Network, title: str`	`None` (displays plot)

Continued on next page

Table III.10 – continued from previous page

Function	Description	Input	Output
`visualize_samples(paths, y_true, num_show, title)`	Displays grid of sample images with labels	`paths: list[str]`, `y_true: np.ndarray`, `num_show: int`, `title: str`	`None` (displays plot)
`predict_single_image(path, net, mean, std, device)`	Predicts digit class for a single image file	`path: str`, `net: Network`, `mean/std: np.ndarray`, `device: str`	`tuple[int, np.ndarray]` (pred, probs)
`visualize_predictions(net, paths, y_true, mean, std, device, num_show)`	Visualizes predictions on validation images with true/predicted labels	`net: Network`, `paths: list[str]`, `y_true: np.ndarray`, `mean/std: np.ndarray`, `device: str`, `num_show: int`	`None` (displays plot)
`save_checkpoint(path, network, cfg)`	Saves network weights and config to file	`path: str`, `network: Network`, `cfg: dict`	`None`
`load_checkpoint(path)`	Loads checkpoint and rebuilds network	`path: str`	`dict` with network and config

4. Implementation Structure and Execution Steps

5. Training Configuration Parameters

This section lists the key configuration parameters used for training the neural network on different datasets.

5.1. Training Configuration

The full training set configuration uses the complete dataset for final model evaluation:

- **Input size**: 784
- **Hidden layers**: [128, 64]
- **Output size**: 10 classes
- **Epochs**: 100
- **Batch size**: 32
- **Learning rate**: 0.01
- **Validation split**: 0.2
- **Weight initialization**: Standard deviation 0.01
- **Normalization mode**: Standardize
- **Hidden activation**: ReLU
- **Seed**: 42

Chapter IV: Experimental Evaluation

1. Visualize Input Images and Model Architecture

1.1. Input Images

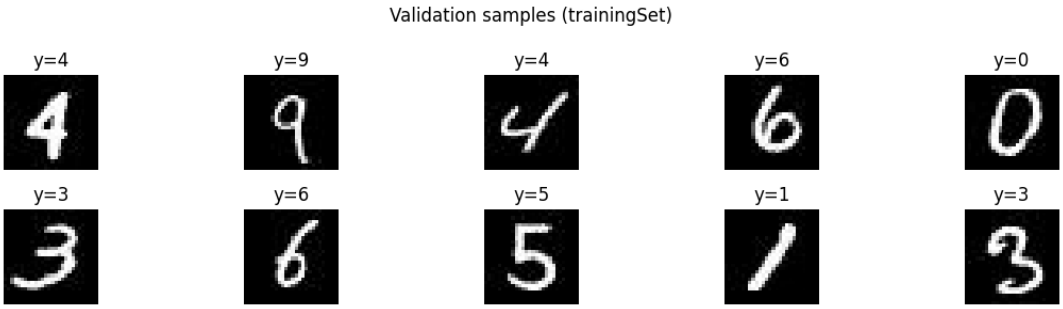

Figure IV.1: Sample of input images from the Validation set

1.2. Model Architecture

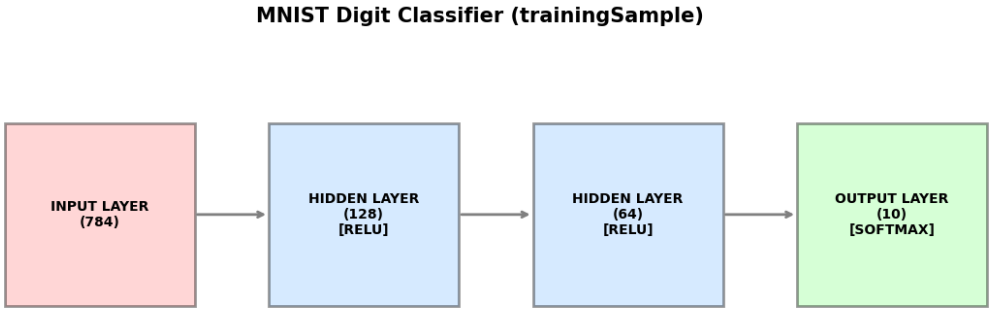

Figure IV.2: Feedforward Neural Network Architecture

2. Execution Results

2.1. Sample

Figure IV.3 shows the prediction results on the training sample dataset.

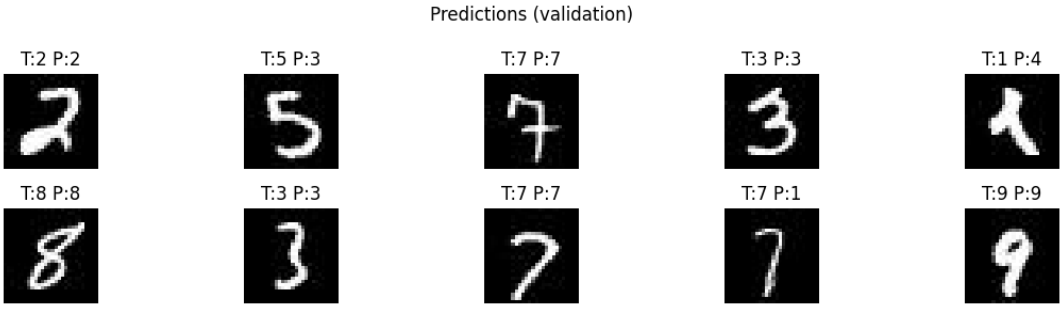

Figure IV.3: Prediction results on training sample

2.2. Set

Figure IV.4 shows the prediction results on the training set dataset.

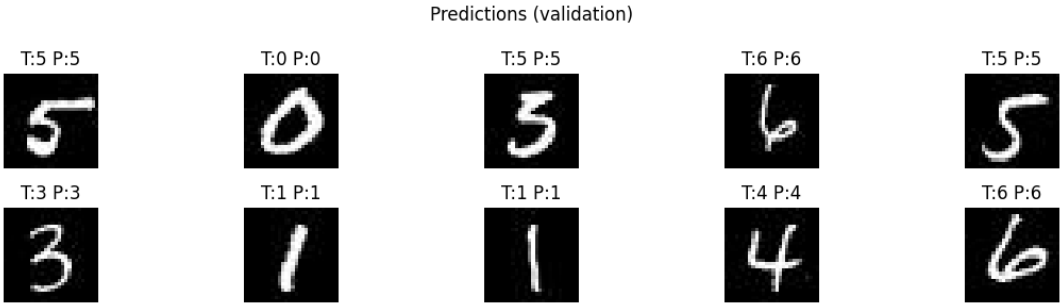

Figure IV.4: Prediction results on training set

3. Evaluation and Discussion

3.1. Training Performance

The training performance is evaluated using loss, accuracy, and confusion matrix metrics for both training sample and training set configurations.

3.1.1 Training Sample Results

Figures IV.5, IV.6, and IV.7 show the loss curve, accuracy curve, and confusion matrix for the training sample configuration, respectively.

As shown in Figures IV.5 and IV.6, the training sample configuration demonstrates rapid learning in the initial epochs. The training loss decreases sharply from approximately 2.2 to near zero (around 0.01–0.02) by epoch 20, while the validation loss stabilizes at a higher value between 0.15 and 0.20.

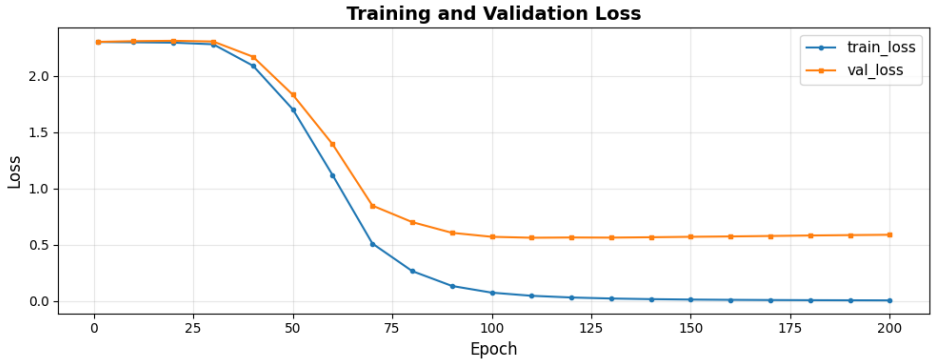

Figure IV.5: Training and validation loss curve for sample configuration

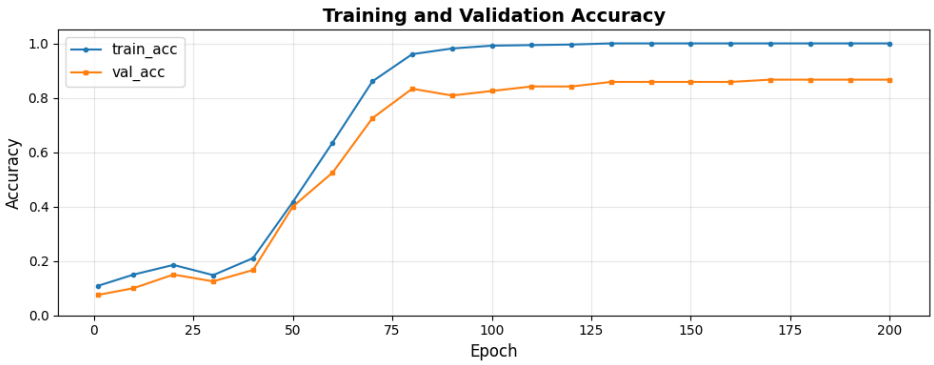

Figure IV.6: Training and validation accuracy curve for sample configuration

Similarly, the training accuracy reaches 1.0 by epoch 75 and remains at perfect accuracy, whereas the validation accuracy plateaus at approximately 0.86–0.87 after epoch 100.

The consistent gap between training and validation metrics indicates overfitting, which is expected given the limited dataset size of the training sample.

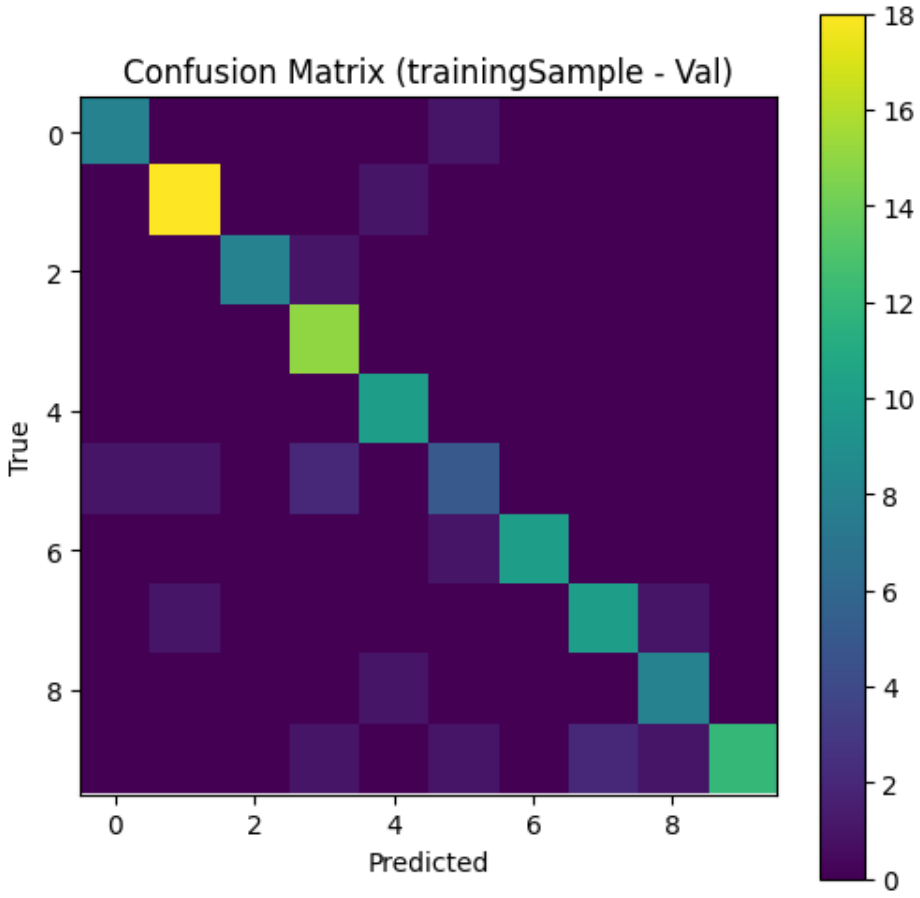

Figure IV.7: Confusion matrix for sample configuration

3.1.2 Training Set Results

Figures IV.8 and IV.9 show the loss curve and accuracy curve for the training set configuration, respectively.

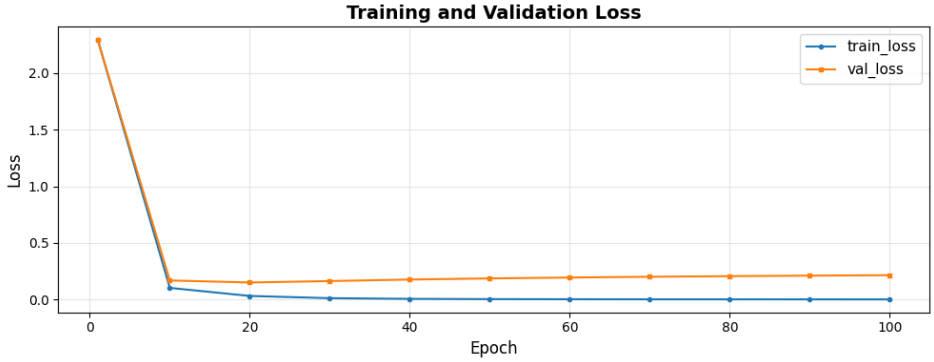

Figure IV.8: Training and validation loss curve for set configuration

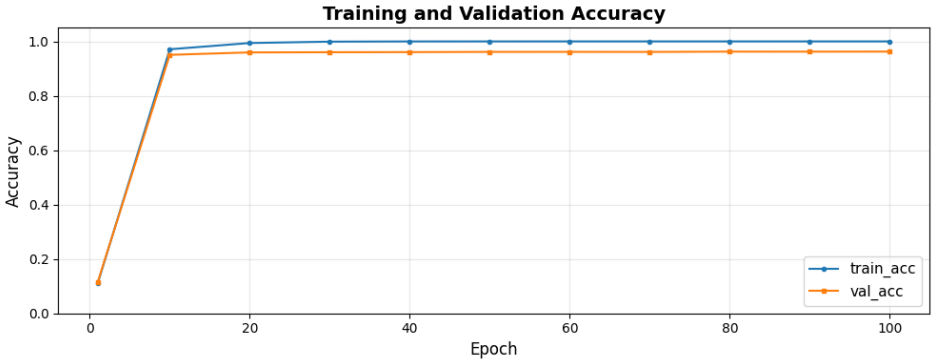

Figure IV.9: Training and validation accuracy curve for set configuration

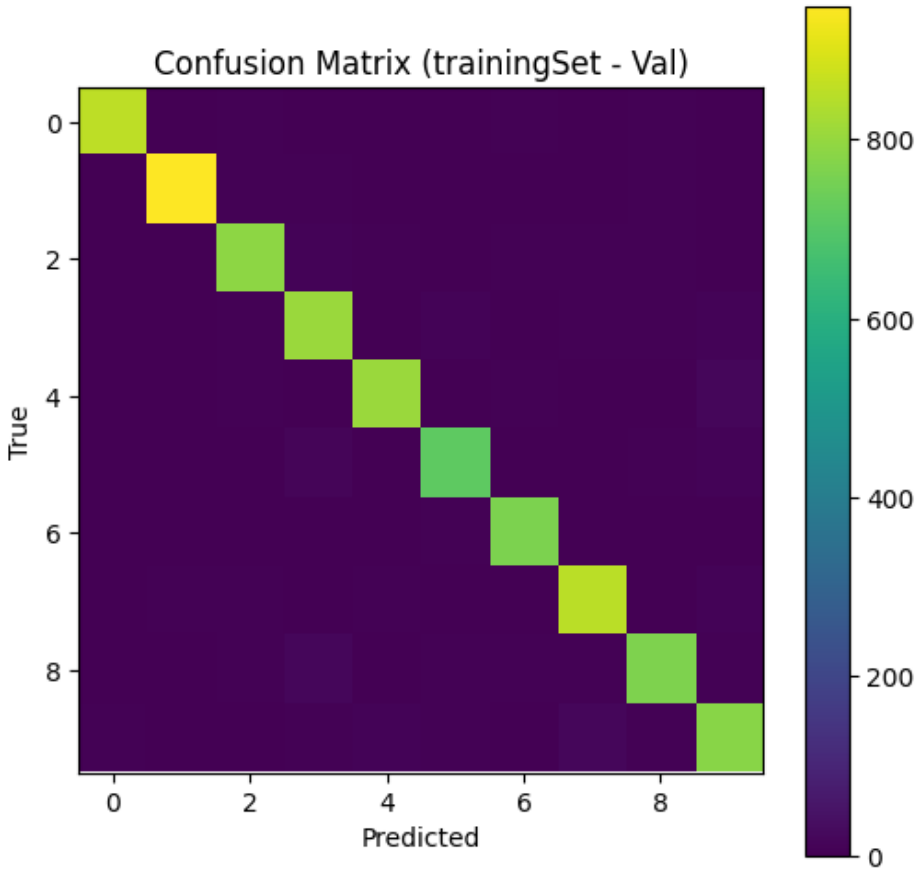

Figure IV.10: Confusion matrix for set configuration

In Figures IV.8 and IV.9, the training set configuration demonstrates strong learning performance with better generalization compared to the sample configuration. The training loss decreases from approximately 2.2 to near zero (around 0.01–0.02) by epoch 20, while the validation loss stabilizes at a lower value between 0.15 and 0.20. The training accuracy reaches approximately 0.985–0.987 and remains stable, whereas the validation accuracy plateaus at around 0.955–0.958 after

epoch 30. The smaller gap between training and validation metrics indicates improved generalization due to the larger dataset size.

3.2. Model Comparison

3.2.1 Compare with one hidden layer

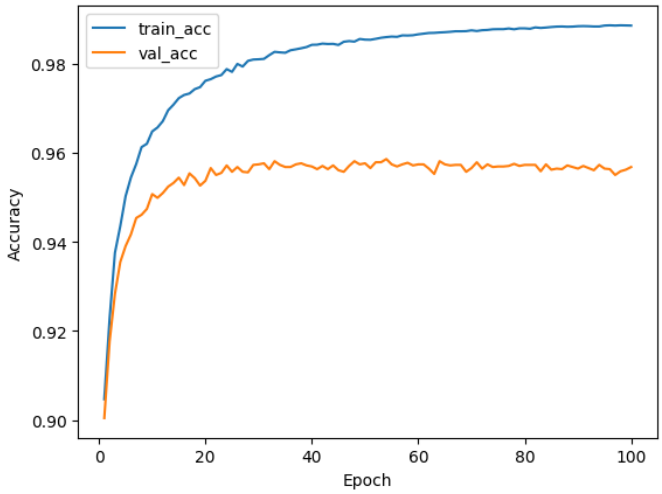

Figure IV.11: Training accuracy curve for set configuration with one hidden layer (Lab 6.1)

Although the network in the previous lab (Lab 6.1) had only one layer and achieved higher accuracy, that model suffered from overfitting and did not perform well on unseen data. In contrast, while the current model's accuracy is slightly lower, we can see from Figure IV.9 that it generalizes better and performs well on new data.

3.2.2 Configuration Comparison

All experimental configurations share the following common parameters:

- **Seed**: 42 (for reproducibility)
- **Dataset**: Training set (full dataset)
- **Input size**: 784
- **Output size**: 10

- **Validation split**: 0.2
- **Shuffle**: True
- **Normalization mode**: Standardize
- **Epsilon**: 1e-6
- **Hidden activation**: ReLU
- **Weight initialization**: Standard deviation 0.01
- **Epochs**: 200
- **Learning rate**: 0.05
- **Print every**: 10 epochs
- **Save every**: 10 epochs

The experimental configurations differ in hidden layer architectures and batch sizes, as shown in Table IV.1.

Table IV.1: Model Configuration Comparison

Config	**Layers**	**Batch Size**
A1_good_2layer_256_128	$256 \rightarrow 128$	64
A2_strong_3layer_512_256_128	$512 \rightarrow 256 \rightarrow 128$	32
B1_batch_32	$256 \rightarrow 128$	32
B2_batch_128	$256 \rightarrow 128$	128

Figure IV.12 presents a comparison of different model configurations, showing the performance metrics across training sample and training set configurations.

```
================================================================================
SUMMARY - EXPERIMENT RESULTS COMPARISON
================================================================================

Config                          Layers                  Batch Size   Train Acc    Val Acc
--------------------------------------------------------------------------------
A1_good_2layer_256_128          256 -> 128              64           1.000000     0.967619
A2_strong_3layer_512_256_128    512 -> 256 -> 128       32           1.000000     0.964881
B1_batch_32                     256 -> 128              32           0.999970     0.968571
B2_batch_128                    256 -> 128              128          1.000000     0.966190

================================================================================
Conclusion:
- Best config (Validation Accuracy): B1_batch_32 with val_acc=0.968571
================================================================================
```

Figure IV.12: Model configuration comparison

As shown in Figure IV.12, all four configurations achieve near-perfect training accuracy (0.999970–1.000000), indicating strong learning capacity. However, validation accuracy varies from 0.964881 to 0.968571, revealing different generalization capabilities. The B1_batch_32 configuration achieves the best validation accuracy of 0.968571, demonstrating that a smaller batch size (32) with a 2-layer architecture (256 → 128) provides optimal generalization. The A2_strong_3layer_512_256_128 configuration, despite having the deepest architecture, achieves the lowest validation accuracy (0.964881), suggesting that increased model complexity does not necessarily improve performance. In conclusion, the batch size has a more significant impact on generalization than the number of hidden layers, with smaller batch sizes (32) yielding better validation performance.

Chapter V: Conclusion

1. Task Completion Summary

No.	Objective	Completion
1	Implement a Feed Forward Neural Network with manual backpropagation in PyTorch using modular design	100%
2	Visualize Image and Network Architecture	100%
3	Load and preprocess image datasets from folder structure for training	100%
4	Train and evaluate the model on both sample and full datasets	100%
5	Visualize training loss, accuracy, and confusion matrices	100%
6	Save and reload trained model parameters for validation	100%
7	Compare the results with different hidden layer sizes, batch sizes, and normalization modes	100%

Table V.1: Objective completion summary

2. Conclusion

This report presented the implementation and evaluation of a Feed Forward Neural Network for handwritten digit classification using an MNIST-like dataset. The network was implemented with manual backpropagation in PyTorch using a modular design approach.

The experimental results demonstrate high classification accuracy with the full training set configuration achieving validation accuracy of 0.955–0.958 and training accuracy of 0.985–0.987. The training sample configuration shows overfitting with training accuracy reaching 1.0 while validation accuracy plateaus at 0.86–0.87. Comparison experiments with four different configurations reveal that batch size has a more significant impact on generalization than the number of hidden layers, with the B1_batch_32 configuration achieving the best validation accuracy of 0.968571.

Best Practices. Using a training sample for early verification helps reduce development time. Training on the full dataset significantly improves generalization, reducing the gap between training and validation metrics. Monitoring validation loss and accuracy provides a reliable indicator

of overfitting. Visualization of training metrics and confusion matrices facilitates understanding of model behavior.

Limitations. The model shows overfitting when trained on limited data, with a consistent gap between training and validation metrics. Increased model complexity does not necessarily improve performance, as demonstrated by the A2_strong_3layer_512_256_128 configuration achieving the lowest validation accuracy despite having the deepest architecture. The fully connected architecture limits the model's ability to exploit spatial information in images compared to convolutional architectures.

Overall, the full training set configuration offers better generalization than the sample configuration, and smaller batch sizes (32) with moderate architectures provide optimal performance for this task.

Reference

[1] GeeksforGeeks. Relu activation function in deep learning. https://www.geeksforgeeks.org/deep-learning/relu-activation-function-in-deep-learning/, 2024. Accessed: 2026-01-02.

[2] E. LeDell and S. Poirier. Exploratory data analysis of mnist. http://varianceexplained.org/r/digit-eda/, 2019. Accessed: 2026-01-02.

[3] Towards Data Science. Creating a multilayer perceptron (mlp) classifier model to identify handwritten digits. https://towardsdatascience.com/creating-a-multilayer-perceptron-mlp-classifier-model-to-identify-handwritten-digits-9bac1b16fe10/, 2020. Accessed: 2026-01-02.

[4] Sefik Ilkin Serengil. Softmax as a neural networks activation function. https://sefiks.com/2017/11/08/softmax-as-a-neural-networks-activation-function/, 2017. Accessed: 2026-01-02.

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.