VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 6.1: PyTorch - Images Input & Visualization - Version #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

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 supports image-based input, data preprocessing from raw image folders, CSV-based dataset construction, 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 optionally normalizing pixel intensities to the range $[0, 1]$. The dataset is stored in a CSV format, where each row consists of one label column followed by 784 pixel values.

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

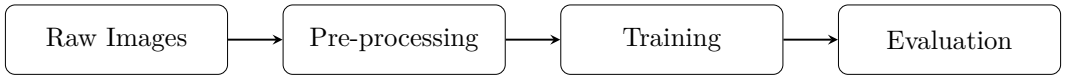

Figure I.2: High-level processing pipeline of the handwritten digit classification system.

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. These images serve as the original data source for the learning process.

Pre-processing stage converts raw images into numerical vectors by flattening and saving as a CSV file, normalizing pixel values, and organizing data into a structured dataset format suitable for learning.

Training stage, model parameters are optimized using labeled data through supervised learning, with the objective of minimizing classification error.

Evaluation stage measures the predictive performance of the trained model on unseen data using quantitative metrics such as accuracy and loss.

3. Objectives

No.	Objective
1	Implement a Feed Forward Neural Network with manual backpropagation in PyTorch.
2	Visualize Image and Network Architecture.
3	Convert image datasets into MNIST-like CSV format 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 sizes and learning rates.

Table I.1: Laboratory Objectives

Chapter II: Network Architecture and 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 [3]. The input space consists of grayscale images of size 28×28 pixels, which are flattened into latent representations of dimension 784 [2]. Each latent vector $\mathbf{x} \in \mathbb{R}^{784}$ encodes the spatial intensity distribution of a handwritten digit, where pixel values are normalized to the range $[0, 1]$.

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: $\hat{y}_i = \arg\max_k f(\mathbf{x}_i)_k$.

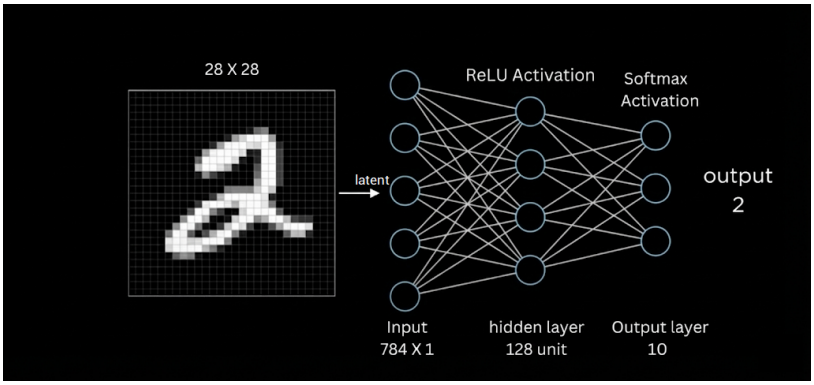

Figure II.1: Feed-forward neural network architecture for digit classification.

2. Network Architecture

The implemented feed-forward neural network consists of three layers: an input layer, a single hidden layer, 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.

2.1. Layer Structure

The network architecture is defined as follows:

- **Input layer**: Receives flattened image vectors of dimension $d_{\text{in}} = 784$, corresponding to 28×28 pixel images.
- **Hidden layer**: Contains d_h neurons (configurable, typically 32 or 128), 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.

2.2. Parameter Initialization

The network parameters consist of weight matrices and bias vectors for each layer. The weight matrix $\mathbf{W}_1 \in \mathbb{R}^{784 \times d_h}$ connects the input layer to the hidden layer, while $\mathbf{W}_2 \in \mathbb{R}^{d_h \times 10}$ connects the hidden layer to the output layer. Corresponding bias vectors are $\mathbf{b}_1 \in \mathbb{R}^{d_h}$ and $\mathbf{b}_2 \in \mathbb{R}^{10}$.

Initialization follows a small-variance normal distribution strategy:

$$\mathbf{W}_1 \sim \mathcal{N}(0, \sigma^2), \quad \sigma = 0.01 \tag{II.1}$$

$$\mathbf{W}_2 \sim \mathcal{N}(0, \sigma^2), \quad \sigma = 0.01 \tag{II.2}$$

$$\mathbf{b}_1 = \mathbf{0} \tag{II.3}$$

$$\mathbf{b}_2 = \mathbf{0} \tag{II.4}$$

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

The network employs two distinct activation functions: Rectified Linear Unit (ReLU) for the hidden layer [1] and Softmax for the output layer [4]. Each function serves a specific purpose in the network's computational pipeline.

3.1. ReLU Activation Function

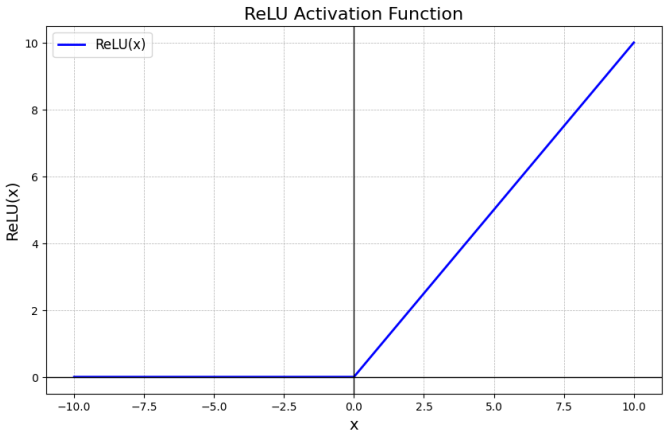

Figure II.2: ReLU activation function and its derivative. [1]

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

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{II.5}$$

The derivative of ReLU with respect to its input is:

$$\frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{II.6}$$

3.1.1 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.

3.2. Softmax Activation Function

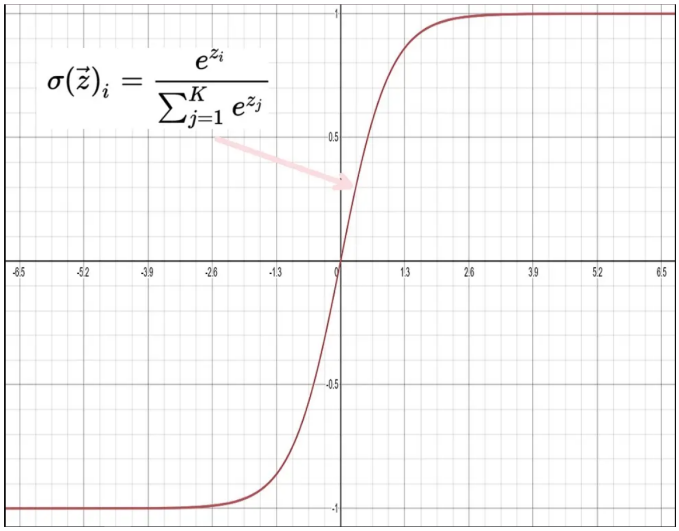

Figure II.3: Softmax activation function mapping logits to probability distribution.

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:

$$\text{Softmax}(\mathbf{z})_k = \frac{\exp(z_k - \max_j z_j)}{\sum_{j=1}^{10} \exp(z_j - \max_j z_j)} \tag{II.7}$$

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:

$$\sum_{k=1}^{10} \text{Softmax}(\mathbf{z})_k = 1 \tag{II.8}$$

$$\text{Softmax}(\mathbf{z})_k \geq 0 \quad \forall k \in \{1, \ldots, 10\} \tag{II.9}$$

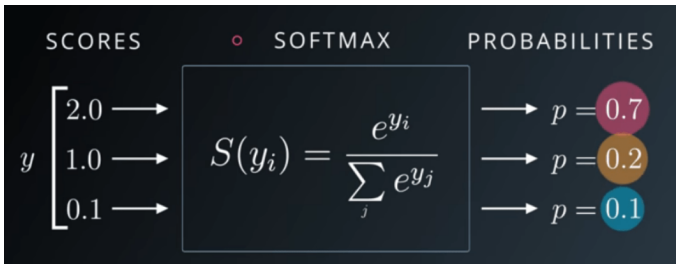

Figure II.4: Softmax example. [4]

3.2.1 Rationale for Softmax

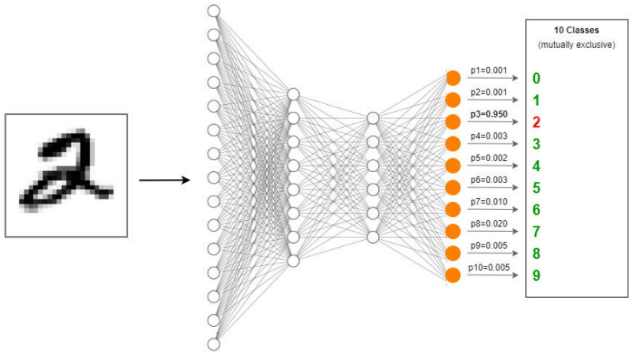

Figure II.5: Softmax architecture. [3]

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.

3.3. Gradient Computation for Softmax

The partial derivative of Softmax with respect to its input logits is required for backpropagation. For a Softmax output $\mathbf{p} = \text{Softmax}(\mathbf{z})$, the Jacobian matrix has elements:

$$\frac{\partial p_k}{\partial z_j} = \begin{cases} p_k(1 - p_k) & \text{if } k = j \\ -p_k p_j & \text{if } k \neq j \end{cases} \tag{II.10}$$

This can be expressed compactly as:

$$\frac{\partial p_k}{\partial z_j} = p_k(\delta_{kj} - p_j) \tag{II.11}$$

where δ_{kj} is the Kronecker delta function.

4. Forward Propagation

Forward propagation computes the network output by sequentially applying linear transformations and activation functions to the input latent representation. The process transforms the input through each layer, storing intermediate values for use in backpropagation.

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:

4.1.1 Input to Hidden Layer

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

$$\mathbf{Z}_1 = \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1 \tag{II.12}$$

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

$$\mathbf{H}_1 = \text{ReLU}(\mathbf{Z}_1) = \max(\mathbf{0}, \mathbf{Z}_1) \tag{II.13}$$

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

4.1.2 Hidden to Output Layer

The hidden activations are transformed through the second linear layer:

$$\mathbf{Z}_2 = \mathbf{H}_1\mathbf{W}_2 + \mathbf{b}_2 \tag{II.14}$$

where $\mathbf{Z}_2 \in \mathbb{R}^{N \times 10}$ contains the output logits. The Softmax function is applied row-wise to produce the final probability distribution:

$$\hat{\mathbf{Y}} = \text{Softmax}(\mathbf{Z}_2) \tag{II.15}$$

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

4.2. Example: Forward Pass with Small Matrices

Consider a simplified example with $N = 2$ samples, $d_{\text{in}} = 4$ (reduced input dimension), and $d_h = 3$ hidden neurons. The forward pass computation proceeds as follows:

4.2.1 Step 1: Input to Hidden

Given input matrix and parameters:

$$\mathbf{X} = \begin{bmatrix} 0.2 & 0.5 & 0.8 & 0.3 \\ 0.1 & 0.9 & 0.4 & 0.6 \end{bmatrix}, \quad \mathbf{W}_1 = \begin{bmatrix} 0.1 & 0.2 & -0.1 \\ 0.3 & -0.2 & 0.4 \\ -0.1 & 0.3 & 0.2 \\ 0.2 & 0.1 & -0.3 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 0.1 & -0.1 & 0.2 \end{bmatrix} \tag{II.16}$$

The pre-activation computation yields:

$$\mathbf{Z}_1 = \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1 = \begin{bmatrix} 0.2 & 0.5 & 0.8 & 0.3 \\ 0.1 & 0.9 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 0.1 & 0.2 & -0.1 \\ 0.3 & -0.2 & 0.4 \\ -0.1 & 0.3 & 0.2 \\ 0.2 & 0.1 & -0.3 \end{bmatrix} + \begin{bmatrix} 0.1 & -0.1 & 0.2 \end{bmatrix} \tag{II.17}$$

$$\mathbf{Z}_1 = \begin{bmatrix} 0.25 & 0.35 & 0.15 \\ 0.42 & 0.18 & 0.28 \end{bmatrix} \tag{II.18}$$

Applying ReLU activation:

$$\mathbf{H}_1 = \text{ReLU}(\mathbf{Z}_1) = \begin{bmatrix} 0.25 & 0.35 & 0.15 \\ 0.42 & 0.18 & 0.28 \end{bmatrix} \tag{II.19}$$

4.2.2 Step 2: Hidden to Output

Given output layer parameters:

$$\mathbf{W}_2 = \begin{bmatrix} 0.2 & -0.1 & 0.3 & 0.1 & -0.2 & 0.15 & 0.05 & -0.1 & 0.25 & 0.0 \\ -0.1 & 0.3 & -0.2 & 0.2 & 0.1 & -0.15 & 0.3 & 0.05 & -0.1 & 0.2 \\ 0.1 & -0.2 & 0.15 & -0.1 & 0.3 & 0.2 & -0.15 & 0.25 & 0.1 & -0.05 \end{bmatrix}, \tag{II.20}$$

$$\mathbf{b}_2 = \begin{bmatrix} 0.05 & -0.05 & 0.1 & 0.0 & 0.05 & -0.1 & 0.15 & -0.05 & 0.1 & 0.0 \end{bmatrix} \tag{II.21}$$

Note: In practice, $\mathbf{W}_2$ has shape $d_h \times 10$, but for brevity, this example uses a reduced output dimension. The computation proceeds:

$$\mathbf{Z}_2 = \mathbf{H}_1\mathbf{W}_2 + \mathbf{b}_2 \tag{II.22}$$

After computing logits, Softmax is applied row-wise. For a sample logit vector $\mathbf{z}_2^{(i)} = [z_1, z_2, \ldots, z_{10}]$, the Softmax output is:

$$p_k = \frac{\exp(z_k - \max_j z_j)}{\sum_{j=1}^{10} \exp(z_j - \max_j z_j)} \tag{II.23}$$

yielding the final probability distribution $\hat{\mathbf{Y}}$ where each row represents class probabilities for one sample.

5. Backward Propagation

Backward propagation computes gradients of the loss function with respect to all network parameters using the chain rule of calculus. The gradients are then used to update parameters via gradient descent, minimizing the classification error.

5.1. Loss Function

The network is trained using the cross-entropy loss function, which measures the discrepancy between predicted probability distributions and true class labels. For a batch of N samples, the loss is defined as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{10} y_{i,k}\log(\hat{y}_{i,k}) \tag{II.24}$$

where $\mathbf{y}_i \in \mathbb{R}^{10}$ is the one-hot encoded ground-truth label vector for sample i, and $\hat{\mathbf{y}}_i = \text{Softmax}(\mathbf{z}_2^{(i)})$ is the predicted probability distribution. The one-hot encoding ensures that $y_{i,k} = 1$ for the true class k and $y_{i,k} = 0$ for all other classes.

When labels are provided as integer class indices $c_i \in \{0, 1, \ldots, 9\}$, the loss simplifies to:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\log(\hat{y}_{i,c_i}) \tag{II.25}$$

5.2. Gradient Computation

The backpropagation algorithm computes gradients layer by layer, starting from the output layer and propagating backward to the input layer.

5.2.1 Output Layer Gradients

The gradient of the cross-entropy loss with respect to the Softmax logits $\mathbf{Z}_2$ is computed first. For a single sample with one-hot target $\mathbf{y}$ and Softmax output $\hat{\mathbf{y}}$, the gradient is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} = \hat{\mathbf{y}} - \mathbf{y} \tag{II.26}$$

For a batch of N samples, this extends to:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_2} = \frac{1}{N}(\hat{\mathbf{Y}} - \mathbf{Y}) \tag{II.27}$$

where $\mathbf{Y} \in \mathbb{R}^{N \times 10}$ is the batch of one-hot encoded labels. This elegant form arises from the combination of cross-entropy loss and Softmax activation, where the gradient reduces to the difference between predictions and targets.

5.2.2 Hidden Layer Parameter Gradients

The gradient with respect to the output layer weights $\mathbf{W}_2$ is computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \mathbf{H}_1^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Z}_2} \tag{II.28}$$

yielding a matrix of shape $d_h \times 10$. The gradient with respect to the output bias $\mathbf{b}_2$ is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_2} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_2^{(i)}} \tag{II.29}$$

which sums the gradient contributions across all samples in the batch.

5.2.3 Propagation to Hidden Layer

The gradient is propagated backward through the hidden layer activation. The gradient with respect to the hidden pre-activations $\mathbf{Z}_1$ is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_1} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_2} \mathbf{W}_2^\top \right) \odot \text{ReLU}'(\mathbf{Z}_1) \tag{II.30}$$

where $\odot$ denotes element-wise multiplication and $\text{ReLU}'(\mathbf{Z}_1)$ is the derivative of ReLU, which equals 1 where $\mathbf{Z}_1 > 0$ and 0 elsewhere.

5.2.4 Input Layer Parameter Gradients

The gradients with respect to the hidden layer weights and biases are:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \mathbf{X}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Z}_1} \tag{II.31}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_1^{(i)}} \tag{II.32}$$

5.3. Parameter Update

After computing all gradients, parameters are updated using gradient descent with learning rate α:

$$\mathbf{W}_1 \leftarrow \mathbf{W}_1 - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} \tag{II.33}$$

$$\mathbf{b}_1 \leftarrow \mathbf{b}_1 - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} \tag{II.34}$$

$$\mathbf{W}_2 \leftarrow \mathbf{W}_2 - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} \tag{II.35}$$

$$\mathbf{b}_2 \leftarrow \mathbf{b}_2 - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}_2} \tag{II.36}$$

5.4. Example: Backward Pass with Small Matrices

Continuing from the forward pass example, assume the predicted probabilities and true labels are:

$$\hat{\mathbf{Y}} = \begin{bmatrix} 0.15 & 0.10 & 0.20 & 0.12 & 0.08 & 0.10 & 0.05 & 0.08 & 0.10 & 0.02 \\ 0.08 & 0.12 & 0.15 & 0.18 & 0.10 & 0.12 & 0.08 & 0.10 & 0.05 & 0.02 \end{bmatrix} \tag{II.37}$$

$$\mathbf{Y} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{II.38}$$

The gradient at the output layer is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}_2} = \frac{1}{2}(\hat{\mathbf{Y}} - \mathbf{Y}) = \begin{bmatrix} 0.075 & 0.05 & -0.4 & 0.06 & 0.04 & 0.05 & 0.025 & 0.04 & 0.05 & 0.01 \\ 0.04 & 0.06 & 0.075 & -0.41 & 0.05 & 0.06 & 0.04 & 0.05 & 0.025 & 0.01 \end{bmatrix} \tag{II.39}$$

The gradient with respect to $\mathbf{W}_2$ is computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \mathbf{H}_1^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Z}_2} \tag{II.40}$$

which involves matrix multiplication between the transposed hidden activations and the output gradient. Similar computations yield gradients for all remaining parameters.

6. Training Methodology

The training process encompasses data preprocessing, normalization, forward propagation, backward propagation, and validation. Each stage transforms the raw image data through mathematical operations to enable effective learning.

6.1. Dataset Preprocessing and Splitting

In this work, pre-processing is defined as the conversion of raw image data into a structured MNIST-like CSV format. The original dataset consists of grayscale digit images stored in label-based directories. Each image is resized to 28×28, flattened into a 784-dimensional vector, and stored together with its digit label in a CSV file.

To support both rapid verification and full-scale training, two pre-processed datasets are generated: a training sample used for debugging and hyperparameter tuning, and a full training set

used for final model evaluation.

For each CSV dataset, training and validation subsets are created during the data loading stage. 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.

6.2. Preprocessing: Image to Latent Representation

Raw grayscale images of size 28×28 pixels are converted into latent vector representations suitable for neural network processing. The transformation pipeline consists of the following steps:

6.2.1 Image Flattening

Each two-dimensional image matrix $\mathbf{I} \in \mathbb{R}^{28\times28}$ is reshaped into a one-dimensional vector $\mathbf{x} \in \mathbb{R}^{784}$ by concatenating rows:

$$\mathbf{x} = \text{flatten}(\mathbf{I}) = [I_{1,1}, I_{1,2}, \ldots, I_{1,28}, I_{2,1}, \ldots, I_{28,28}]^\top \tag{II.41}$$

This operation preserves spatial information while converting the image into a format compatible with fully-connected layers.

6.2.2 Pixel Value Normalization

Pixel intensities are initially in the range $[0, 255]$ (8-bit grayscale). These values are normalized to $[0, 1]$ by division:

$$\mathbf{x}_{\text{norm}} = \frac{\mathbf{x}}{255} \tag{II.42}$$

This normalization ensures that input features are on a consistent scale, preventing numerical instability and facilitating gradient-based optimization.

6.2.3 Batch Formation

Multiple latent vectors are organized into batches for efficient processing. A batch matrix $\mathbf{X} \in \mathbb{R}^{N\times784}$ contains N samples, where each row represents one flattened and normalized image.

6.3. Feature Normalization

After initial pixel normalization, additional feature normalization may be applied to stabilize training. Two common approaches are considered:

6.3.1 Column-wise Normalization

Each feature (pixel position) is normalized by its maximum value across the training set:

$$x_{i,j}^{\text{norm}} = \frac{x_{i,j}}{\max_k x_{k,j}} \tag{II.43}$$

where $x_{i,j}$ denotes the j-th pixel of the i-th sample. This ensures all features have a maximum value of 1, preventing any single pixel from dominating the learning process.

6.3.2 Standardization

Alternatively, features can be standardized to have zero mean and unit variance:

$$x_{i,j}^{\text{std}} = \frac{x_{i,j} - \mu_j}{\sigma_j} \tag{II.44}$$

where $\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$ and $\sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2}$ are the mean and standard deviation of feature j computed over the training set.

6.4. Forward Propagation

During training, forward propagation computes predictions for each batch. The process follows the sequence described in Section 3, transforming input latent vectors $\mathbf{X}$ through linear layers and activations to produce probability distributions $\hat{\mathbf{Y}}$.

6.5. Backward Propagation

Backward propagation computes gradients and updates parameters as described in Section 4. The process is repeated for each batch in the training set, with gradients accumulated or averaged appropriately depending on the batch size.

6.6. Validation and Loss Computation

After each training epoch, model performance is evaluated on a held-out validation set to monitor generalization and detect overfitting.

6.6.1 Validation Accuracy

For each validation sample, the predicted class is obtained via:

$$\hat{c}_i = \arg\max_k \hat{y}_{i,k} \tag{II.45}$$

The validation accuracy is computed as:

$$\text{Accuracy} = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} \mathbb{1}[\hat{c}_i = c_i] \tag{II.46}$$

where $\mathbb{1}[\cdot]$ is the indicator function, c_i is the true class label, and N_{val} is the number of validation samples.

6.6.2 Validation Loss

The cross-entropy loss on the validation set provides a measure of prediction confidence:

$$\mathcal{L}_{\text{val}} = -\frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} \log(\hat{y}_{i,c_i}) \tag{II.47}$$

Lower validation loss indicates better model calibration, where predicted probabilities align more closely with true class assignments.

6.6.3 Training Monitoring

Training and validation metrics are tracked across epochs to assess learning progress. Key observations include:

- Decreasing training loss indicates successful optimization.
- Validation accuracy plateauing suggests convergence.
- Large gap between training and validation accuracy may indicate overfitting.

- Consistent improvement in both metrics indicates effective learning.

6.7. Example: Complete Training Step

Consider a mini-batch of two samples with simplified dimensions. The preprocessing yields:

$$\mathbf{X} = \begin{bmatrix} 0.2 & 0.5 & 0.8 & 0.3 \\ 0.1 & 0.9 & 0.4 & 0.6 \end{bmatrix}, \quad \mathbf{y}_{\text{int}} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \tag{II.48}$$

After forward propagation, predictions are obtained. Backward propagation computes gradients, and parameters are updated. The process repeats for subsequent batches until all training data is processed in one epoch.

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. For the implemented architecture, the state dictionary includes:

- $\mathbf{W}_1$: Input-to-hidden weight matrix
- $\mathbf{b}_1$: Hidden layer bias vector
- $\mathbf{W}_2$: Hidden-to-output weight matrix
- $\mathbf{b}_2$: Output layer bias vector

Each parameter tensor is stored with its associated key, allowing precise reconstruction of the network's learned mapping.

7.2. Saving Model Parameters

When saving, the state dictionary is serialized to disk using a binary format. The saved file contains only the parameter values, not the network architecture definition. This approach ensures:

- **Portability**: Saved models can be loaded into networks with matching architectures regardless of implementation details.
- **Efficiency**: Only essential parameters are stored, minimizing file size.
- **Version independence**: Parameter values are architecture-agnostic, allowing compatibility across different framework versions.

7.3. Loading Model Parameters

To restore a saved model, a new network instance with the same architecture must be created. The saved state dictionary is then loaded and applied to the network parameters, replacing initial random values with trained weights. This process requires:

- Matching architecture dimensions (input size, hidden size, output size).
- Consistent parameter naming conventions.
- Validation to ensure all required parameters are present.

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

Chapter III: Installation and Usage

This chapter describes the environment setup, dataset acquisition, and execution procedures for the handwritten digit classification system.

1. Environment Setup

1.1. System Requirements

The project requires the following components:

- Python 3.6 or higher
- PyTorch library
- NumPy, pandas, matplotlib, opencv-python
- Jupyter Notebook for interactive execution
- Minimum 2GB RAM (4GB recommended)
- Approximately 500MB disk space for dependencies and model files

1.2. Python and Library Installation

a) Create Virtual Environment.

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

Activate the virtual environment:

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

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

b) Install Required Libraries.

```
pip install torch numpy pandas matplotlib opencv-python pillow
   jupyter
```

For dataset download via Kaggle API:

```
pip install kagglehub
```

1.3. Main Libraries

- `torch`: Tensor computation and neural network implementation
- `numpy`: Numerical computing and array manipulation
- `pandas`: Data handling and CSV operations
- `matplotlib`: Visualization of results and images
- `opencv-python`: Image loading and processing
- `PIL`: Image format conversion and resizing

1.4. Installation Check

Verify installation by importing libraries:

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

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`

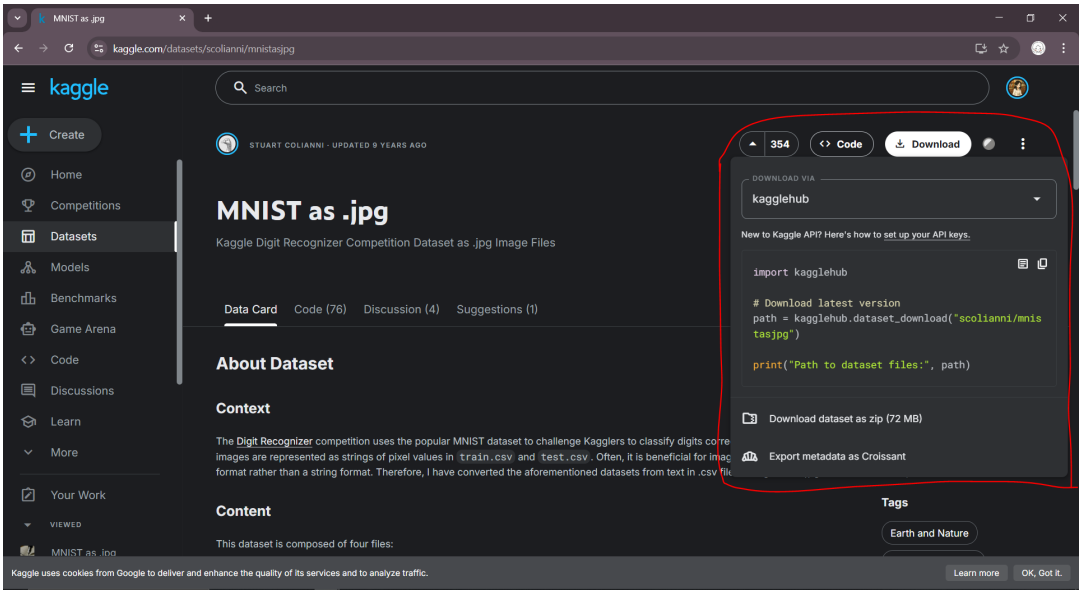

Figure III.1: Kaggle dataset download page for MNIST as JPG.

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. How to Run the Program

3.1. Running the Notebook

The implementation is provided as a Jupyter Notebook (`23127266_p06_01.ipynb`). Execute cells sequentially by clicking "Run" or pressing Shift+Enter.

3.2. Execution Workflow

1. **Environment Setup**: Import required libraries and verify installation
2. **Dataset Download**: Acquire dataset using Method 1 or Method 2
3. **Preprocessing**: Convert images to CSV format using `build_mnist_like_csv()`
4. **Model Definition**: Define `FFNeuralNetwork` class with ReLU and Softmax activations

5. **Data Loading**: Load and normalize data using `load_mnist_csv()`
6. **Training**: Train model using `train_ffnn_from_csv()`
7. **Evaluation**: Visualize results and compute accuracy metrics
8. **Model Persistence**: Save trained models to disk

3.3. Model Files

After training, the following model files are saved:

- `ffnn_mnist_state_dict.pt`: Model trained on training sample
- `ffnn_mnist_state_dict_set.pt`: Model trained on full training set

These files contain the model's state dictionary (weights and biases) serialized using PyTorch's `torch.save()` function.

4. Project Directory Structure

4.1. Project Structure

The implementation is organized as a single Jupyter Notebook containing all components: data preprocessing, neural network definition, training procedures, and evaluation utilities.

4.2. Class Structure

4.2.1 FFNeuralNetwork Class

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

Table III.1: FFNeuralNetwork class methods

Method	**Input**	**Output**
`__init__(config)`	Configuration dict with input_size, hidden_size, output_size	Initialized model with random weights
`relu(z)`	Pre-activation tensor z	ReLU-activated tensor
`relu_derivative(z)`	Pre-activation tensor z	ReLU derivative tensor
`softmax(z)`	Logits tensor z	Probability distribution tensor
`forward(X)`	Input batch $X \in \mathbb{R}^{N\times 784}$	Output probabilities $\hat{Y} \in \mathbb{R}^{N\times 10}$
`backward(X, y_int, lr)`	Input batch, integer labels, learning rate	Updated model parameters
`train_step(X, y_int, lr)`	Input batch, integer labels, learning rate	Output probabilities
`save_weights(model, path)`	Model instance, file path	Saved state dictionary file
`load_weights(path, config)`	File path, configuration dict	Loaded model instance

4.3. Utility Functions

Table III.2: Utility functions for data processing and visualization

Function	Input	Output
`build_mnist_like_csv (dataset_root, split, out_csv, normalize)`	Dataset root path, split name, output CSV path, normalization mode	Tuple: (saved CSV path, statistics dict)
`load_mnist_csv (config)`	Configuration dict with csv_path, input_size, output_size	Dict: X_train, y_train_int, y_train_oh, X_val, y_val_int, y_val_oh, mean, std
`train_ffnn_from_csv(config)`	Configuration dict with training parameters	Dict: model, history, train_acc, val_acc, y_val_int, val_out
`visualize_input_images_from_csv (csv_path, num_images)`	CSV path, number of images	Displayed image grid
`visualize_results(results)`	Results dict from training	Plots: loss curves, accuracy curves, confusion matrix
`evaluate_and_visualize_from_csv (config, model_path, num_show)`	Config dict, model path, number of samples	Dict: val_acc, picked_indices, y_true, y_pred

5. Implementation Structure and Execution Steps

5.1. Implementation Sections

No.	Section Name	Summary (Inputs / Outputs / Properties)
1	Environment Setup	Inputs: none. Outputs: imported libraries, verified installation.
2	Dataset Download	Inputs: dataset URL or Kaggle API credentials. Outputs: `mnistasjpg_data` directory with image folders.

No.	Section Name	Summary (Inputs / Outputs / Properties)
3	Pre-processing Dataset	Inputs: image directory paths. Outputs: CSV files (`mnist_trainingSample.csv`, `mnist_trainingSet.csv`) with label and 784 pixel columns.
4	Define FFNeuralNetwork	Class properties: W1, b1, W2, b2 (weights and biases), z1, h1, z2, out (intermediate values). Inputs: configuration dict. Outputs: initialized model.
5	Build CSV from Images	Function: `build_mnist_like_csv()`. Inputs: dataset root, split name, output CSV path. Outputs: CSV file path and statistics dict.
6	Load and Normalize Data	Function: `load_mnist_csv()`. Inputs: config dict with csv_path. Outputs: train/validation splits with normalized features and one-hot encoded labels.
7	Training Function	Function: `train_ffnn_from_csv()`. Inputs: config dict with hyperparameters. Outputs: trained model, training history, accuracy metrics.
8	Visualization Utilities	Functions: `visualize_input_images_from_csv()`, `visualize_results()`. Inputs: CSV path or results dict. Outputs: displayed plots and images.
9	Model Evaluation	Function: `evaluate_and_visualize_from_csv()`. Inputs: config dict, model path. Outputs: validation accuracy, prediction visualizations.
10	Model Persistence	Methods: `save_weights()`, `load_weights()`. Inputs: model instance, file path. Outputs: saved/loaded model files (`ffnn_mnist_state_dict.pt`, `ffnn_mnist_state_dict_set.pt`).

No.	Section Name	Summary (Inputs / Outputs / Properties)

Table III.3: Implementation sections in the notebook

5.2. Key Implementation Details

Data Preprocessing: Images are loaded, resized to 28×28 pixels, converted to grayscale, flattened to 784-dimensional vectors, and normalized to $[0, 1]$ range. The processed data is stored in CSV format with one label column and 784 pixel columns.

Training Process: Training uses manual forward and backward propagation with ReLU activation in the hidden layer and Softmax in the output layer. Cross-entropy loss is minimized using gradient descent with configurable learning rate and batch size.

Model Architecture: The network consists of an input layer (784 neurons), one hidden layer (configurable size, typically 32 or 128 neurons), and an output layer (10 neurons). Weights are initialized from a normal distribution with small variance.

Model Persistence: Model parameters are saved using PyTorch's `state_dict()` mechanism, storing only weights and biases. Saved models can be loaded by creating a new instance with matching architecture and applying the state dictionary.

Chapter IV: Experimental Evaluation

1. Visualize Input Images and the Model Architecture

1.1. Input Images

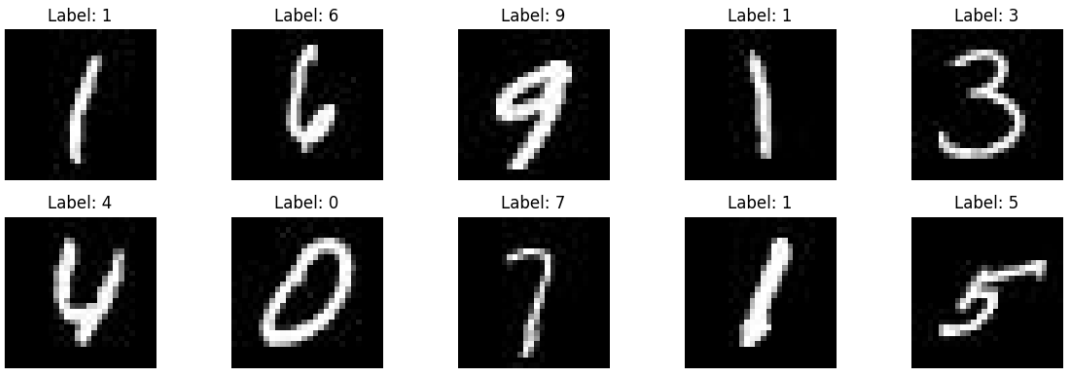

Figure IV.1: Sample input images from the dataset

1.2. Model Architecture

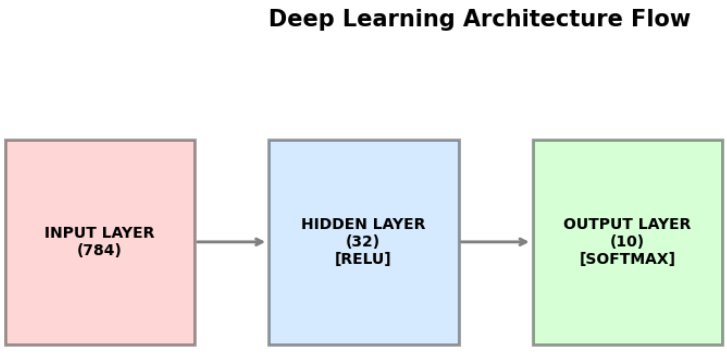

Figure IV.2: Feedforward Neural Network Architecture

2. Execution Results

2.1. Sample

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

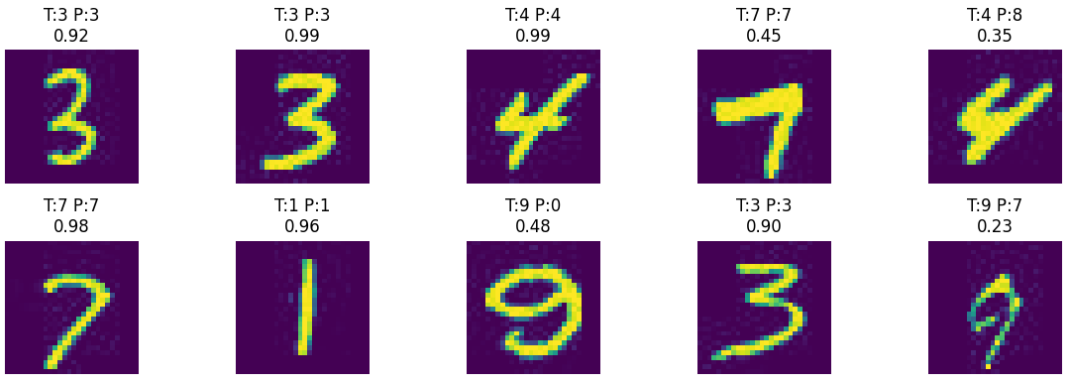

Figure IV.3: Prediction results on training sample

2.2. Set

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

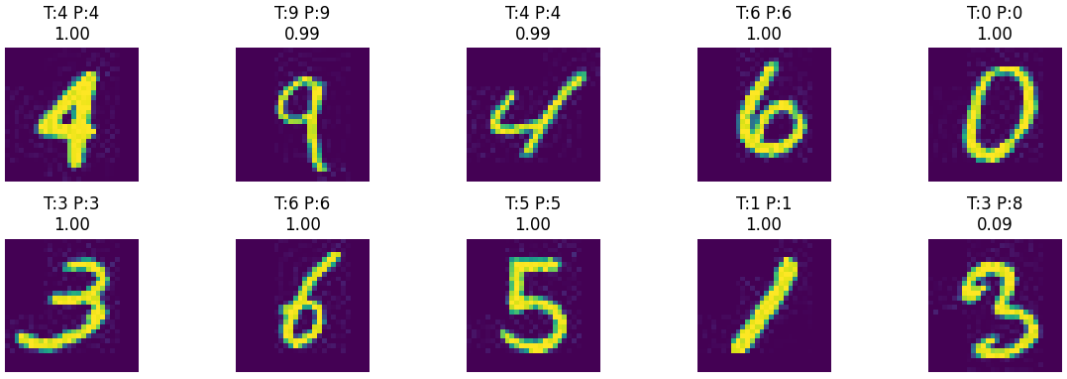

Figure IV.4: Prediction results on training set

2.3. Loading model and Testing in Test Set Results

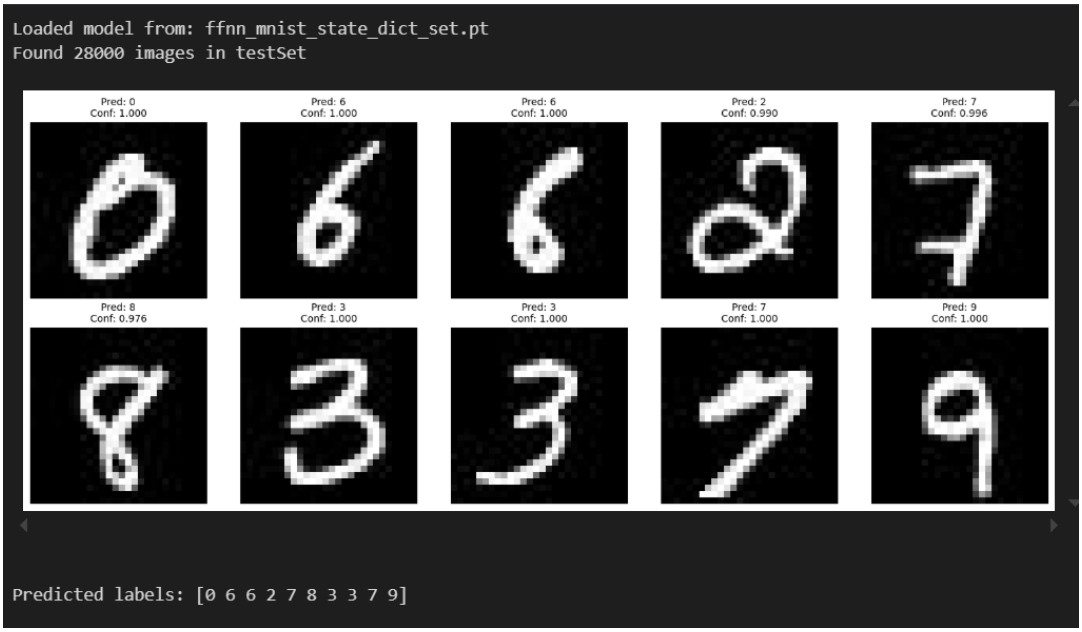

Figure IV.5: Loading results

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.6, IV.7, and IV.8 show the loss curve, accuracy curve, and confusion matrix for the training sample configuration, respectively.

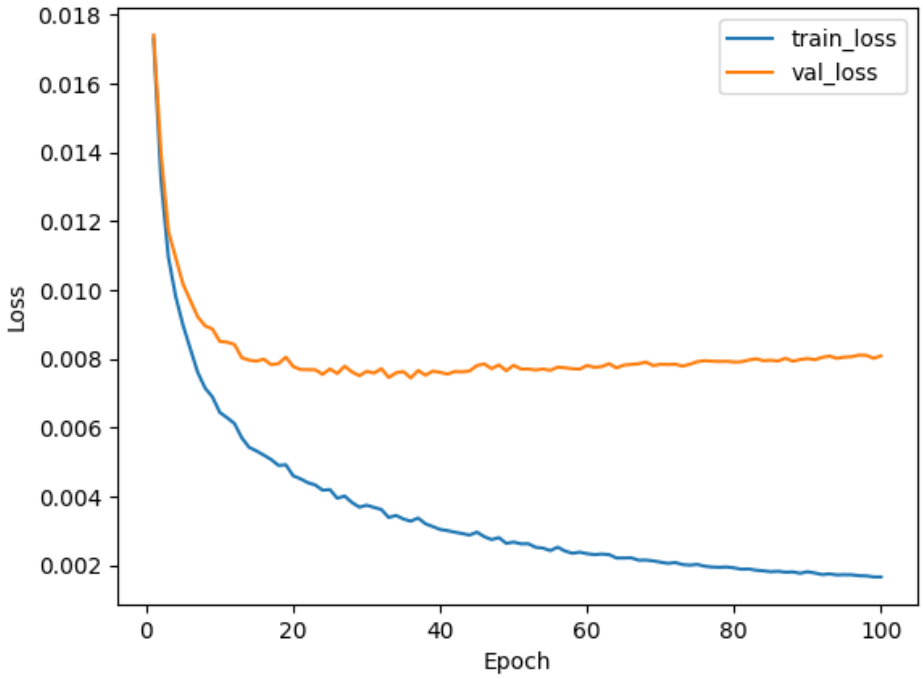

Figure IV.6: Training loss curve for sample configuration

The training loss decreases rapidly toward zero, whereas the validation loss decreases initially and then gradually increases after approximately 20–30 epochs. This divergence further confirms overfitting behavior.

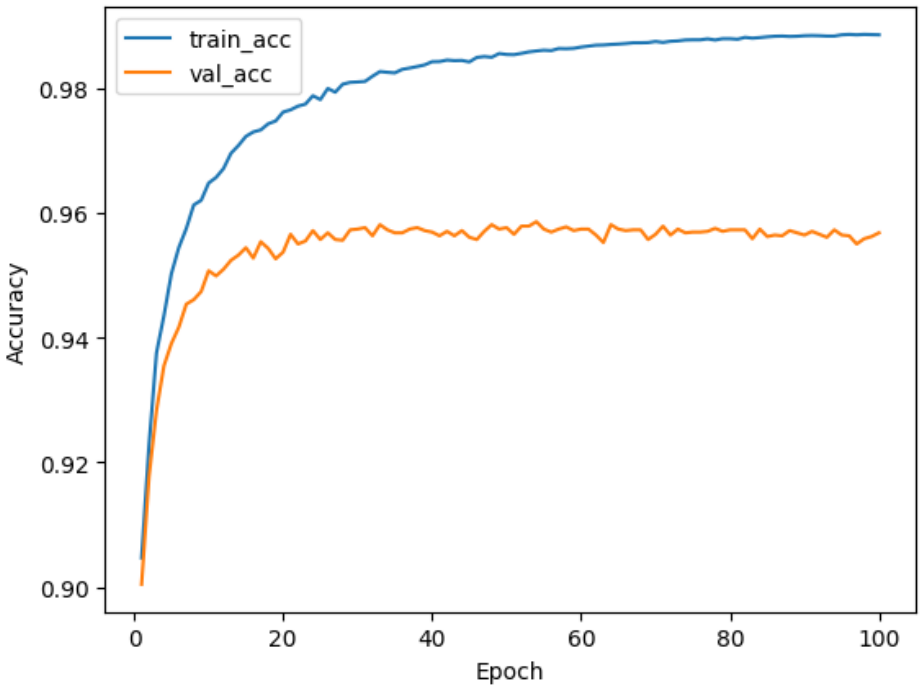

Figure IV.7: Training accuracy curve for sample configuration

For the training sample configuration, the model reaches a training accuracy of approximately 99% after 100 epochs, while the validation accuracy stabilizes around 95–96%. The gap of about 3–4% between training and validation accuracy indicates noticeable overfitting due to the limited data size.

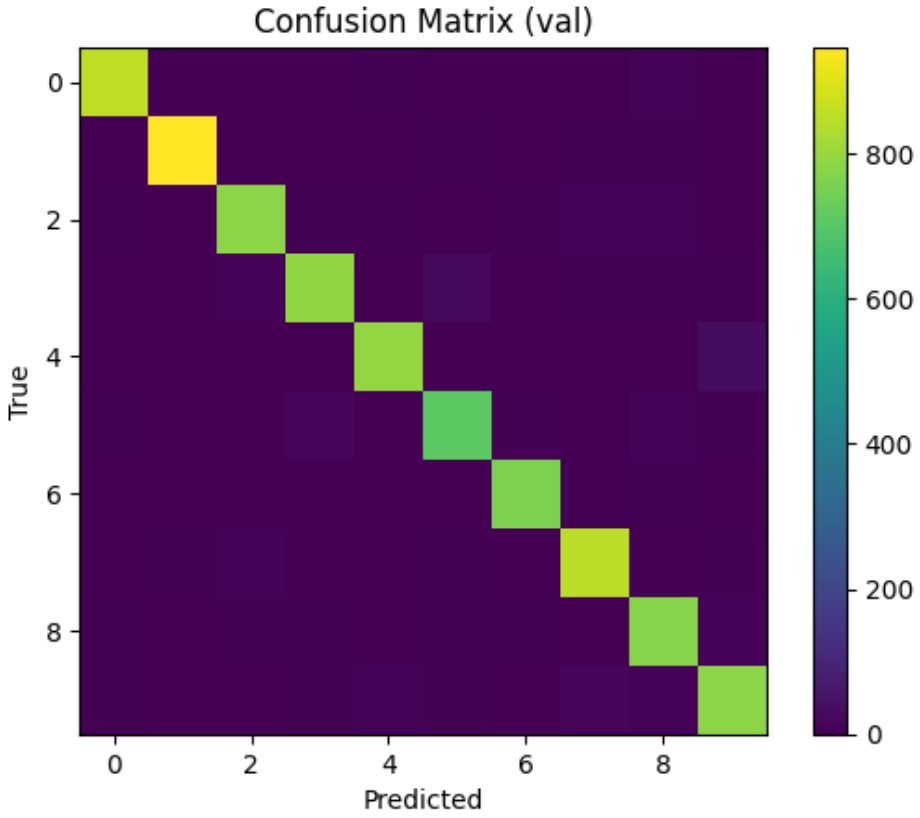

Figure IV.8: Confusion matrix for sample configuration

The confusion matrix shows strong performance on most digit classes; however, several misclassifications are observed, especially for visually similar digits such as 3 and 8. Overall, the training sample configuration is effective for debugging and hyperparameter tuning, but it does not provide optimal generalization.

3.1.2 Training Set Results

Figures IV.9, IV.10, and IV.11 show the loss curve, accuracy curve, and confusion matrix for the training set configuration, respectively.

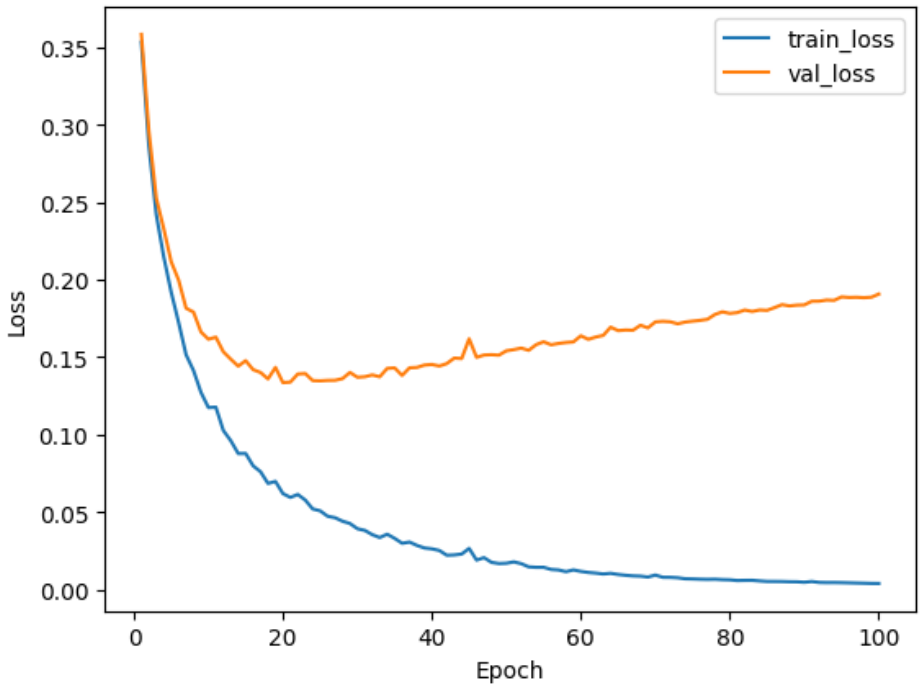

Figure IV.9: Training loss curve for set configuration

The loss curves show stable convergence, with validation loss remaining relatively low and increasing only slightly after convergence. This behavior suggests mild overfitting but significantly better stability than the training sample configuration.

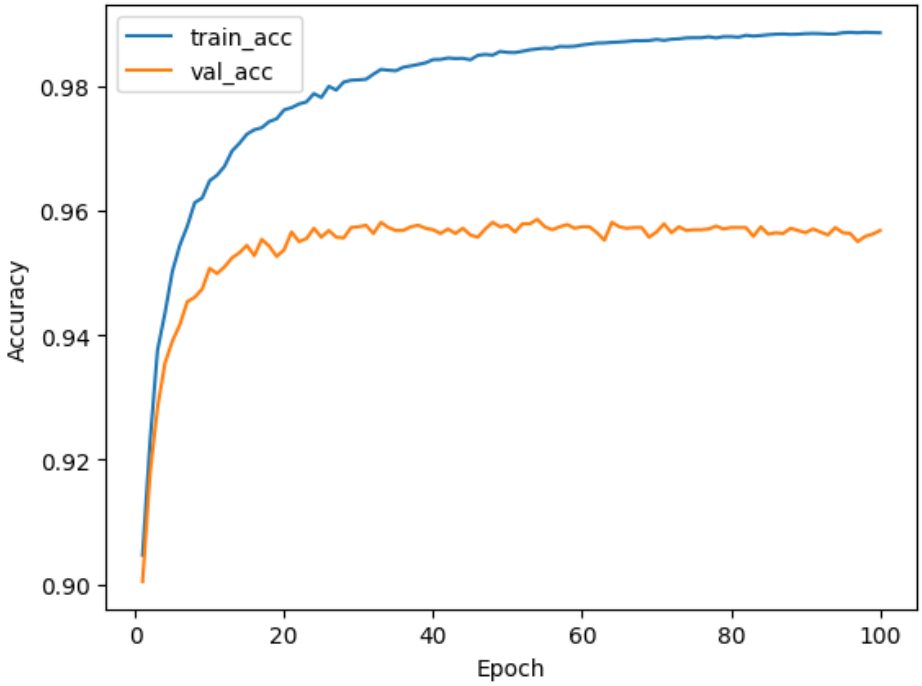

Figure IV.10: Training accuracy curve for set configuration

For the training set configuration, the model achieves a final training accuracy close to 100%, while the validation accuracy converges to approximately 96.1%. Compared to the sample setting, the accuracy gap is reduced to about 3.9%, indicating improved generalization.

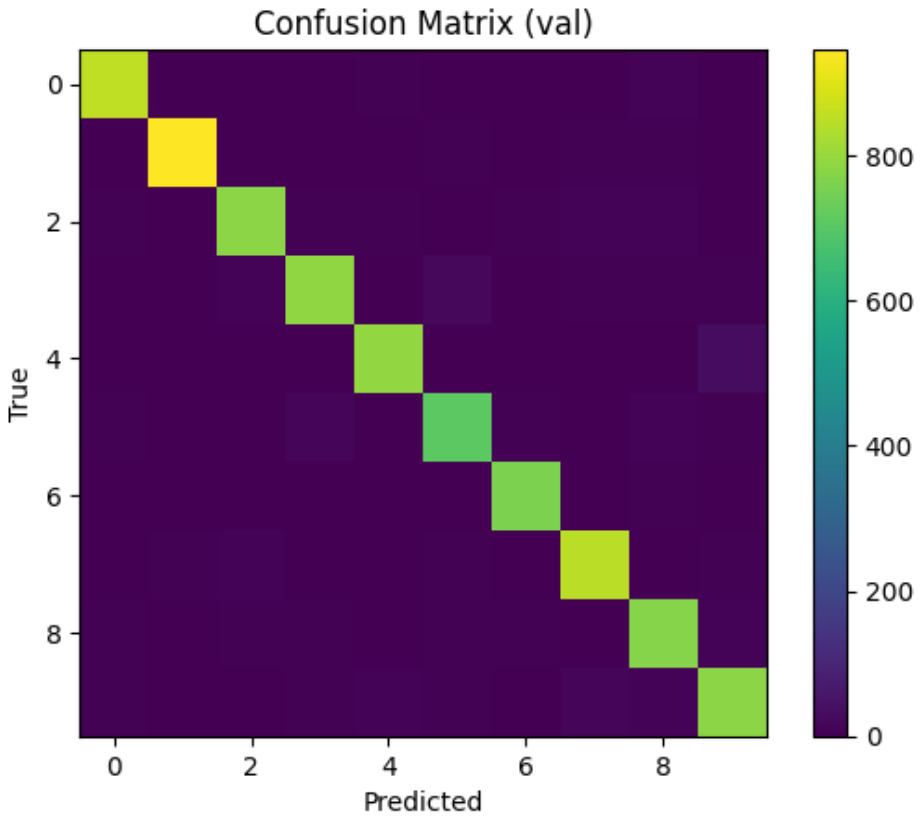

Figure IV.11: Confusion matrix for set configuration

The confusion matrix is strongly diagonal, indicating high classification accuracy across all digit classes. Misclassifications are rare and mostly occur among visually ambiguous digits. Overall, this configuration demonstrates robust performance and reliable generalization.

3.2. Model Comparison

3.2.1 Model Configuration Comparison

Three model configurations are evaluated to analyze the impact of hidden layer size and learning rate on performance. The configurations are designed as follows:

CONFIG (baseline): Hidden size 128 with learning rate 0.2. The larger hidden size provides increased model capacity for complex feature learning, while the higher learning rate enables faster convergence. This configuration achieves the best validation accuracy of 97.49%.

CONFIG_32: Hidden size 32 with learning rate 0.05. The smaller hidden size reduces model complexity and computational cost, suitable for resource-constrained scenarios. The lower learning

rate prevents instability with the reduced capacity. This configuration achieves 96.17% validation accuracy, showing a trade-off between model size and performance.

CONFIG_64: Hidden size 64 with learning rate 0.1. This configuration balances model capacity and training stability. The moderate hidden size offers a compromise between CONFIG_32 and CONFIG, while the intermediate learning rate provides stable convergence. Validation accuracy reaches 96.90%.

The results demonstrate that increasing hidden size generally improves validation accuracy, with CONFIG achieving the best generalization. However, larger models require higher learning rates to effectively utilize their capacity. The learning rate scaling is proportional to hidden size to maintain training stability and convergence speed.

Table IV.1: Model Configuration Comparison

Config	Hidden Size	Learning Rate
CONFIG	128	0.2
CONFIG_32	32	0.05
CONFIG_64	64	0.1

These configurations are trained with batch size 64.

3.2.2 Results Comparison

Figure IV.12 presents a comparison of different model configurations, showing the performance metrics across various hyperparameter settings.

```
============================================================
SUMMARY - ACCURACY COMPARISON
============================================================

Config              Hidden Size     Learning Rate   Train Acc       Val Acc
------------------------------------------------------------------------------
CONFIG              128             0.2             1.000000        0.974881
CONFIG_32           32              0.05            0.998155        0.961667
CONFIG_64           64              0.1             1.000000        0.969048

============================================================
Conclusion:
- Best config (Validation Accuracy): CONFIG (hidden_size=128) with val_acc=0.974881
============================================================
```

Figure IV.12: Model configuration comparison

Figure IV.12 highlights the impact of dataset size and configuration on model performance. Models trained on the full training set consistently outperform those trained on the reduced sample, particularly in validation accuracy and stability.

While both configurations achieve high training accuracy, the training set configuration yields lower validation loss and a more consistent accuracy curve. This confirms that increasing dataset size is more effective for improving generalization than solely adjusting hyperparameters.

Therefore, the training set configuration is considered the better choice for final evaluation, whereas the training sample configuration is suitable for rapid experimentation and model verification.

Chapter V: Conclusion

1. Task Completion Summary

No.	Objective	Completion
1	Implement a Feed Forward Neural Network with manual backpropagation in PyTorch	100%
2	Visualize input images and network-related results	100%
3	Convert image datasets into MNIST-like CSV format 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 sizes and learning rates	100%

Table V.1: Objective completion summary

2. Conclusion

3. Conclusion

This report presented the design, implementation, and evaluation of a Feed Forward Neural Network for handwritten digit classification using an MNIST-like dataset. The experimental results demonstrate that the proposed approach is able to achieve high classification accuracy with stable convergence behavior, especially when trained on the full training set.

Best Practices. Effective dataset preprocessing through CSV-based representation simplifies data loading and ensures reproducibility. Using a training sample for early verification and hyper-parameter tuning helps reduce development time and detect implementation errors. Training on the full dataset significantly improves generalization, while monitoring validation loss and accuracy provides a reliable indicator of overfitting.

Limitations. The model shows overfitting when trained on limited data due to its fully

connected architecture. In addition, the absence of convolutional layers limits the model's ability to exploit spatial information in images, which restricts its performance compared to more advanced architectures.

Overall, the full training set configuration offers the best balance between accuracy and generalization and is recommended for final evaluation.

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.