# Deep Learning Final Report

NGUYEN Thien An
*M1 Student*
*University of Science and Technology of Hanoi (USTH)*
Hanoi, Vietnam
annt2440048@usth.edu.vn

***Abstract*—Convolutional Neural Networks (CNNs) are one of the many Deep Learning models that leverage the spatial structure of data, typically in processing images but also used in various fields such as speech recognition, to perform tasks like classification, detection, and segmentation with high accuracy. This report provides the details implementation of a CNN from scratch without relying on high-level deep learning frameworks. Key components include convolutional layers, pooling mechanisms, and activation functions. Additionally, an automatic differentiation (autograd) mechanism is implemented to support gradient-based optimization and enable efficient backpropagation. This project serves as both a technical exploration and an educational resource for understanding the foundations of CNNs.**

***Index Terms*—Convolutional Neural Networks, MNIST, Deep Learning, Backpropagation, Autograd, Tensor**

## I. Introduction

Convolutional Neural Networks (CNNs) are one of the many Deep Learning models that leverage the spatial structure of data, typically in processing images but also used in various fields such as speech recognition, to perform tasks like classification, detection, and segmentation with high accuracy. Their ability to automatically learn spatial hierarchies of features from raw image data has made them the backbone of many modern AI systems.

This report focuses on the implementation of a CNN from scratch, specifically for handwritten digit recognition using the MNIST dataset-a widely used benchmark in the machine learning community-consisting of 60000 training images and 10000 test images of grayscale digits ranging from 0 to 9. In this version, a reduced version with 12000 samples is used.

Since the primary objective of this work is to develop a comprehensive understanding of the internal mechanisms of CNNs. To achieve this, key components such as convolutional layers, pooling layers, and fully connected layers must be implemented manually. This approach not only reinforces the mathematical foundations of CNNs but also offers insight into the computational processes involved in their training and optimization.

The code for this work can be found here: https://github.com/TGraceAn/dl2025. To use, upload your own MNIST datasets in the CNN folder, edit your configurations, and then run:

```
python main.py
```

Also, make sure that you're in the CNN folder if you want to use the configs directly.

## II. Related Work

The MNIST dataset has been extensively studied and serves as a standard benchmark for evaluating machine learning and deep learning models on digit recognition tasks. Early approaches to this problem utilized traditional machine learning techniques such as Support Vector Machines (SVMs), k-Nearest Neighbors (k-NN), and handcrafted feature extraction methods, including Histogram of Oriented Gradients (HOG) and edge detection.

With the rise of deep learning, Convolutional Neural Networks (CNNs) have become the dominant architecture for image-related tasks due to their ability to automatically learn local representations from raw pixel data. CNNs solve the problem of fully connected (FC) layers by introducing convolutional and pooling operations that reduce the number of parameters and exploit spatial hierarchies in images, enabling more efficient and effective learning. Of course, we all know Yann LeCun, person who is considered the "Godfather" of CNN.

## III. Motivation

Traditional machine learning models, such as Support Vector Machines and k-Nearest Neighbors, rely heavily on manual feature extraction techniques, which often require domain expertise and may not generalize well across different image datasets. These methods typically treat images as flat vectors, ignoring the inherent spatial structure of the data.

Convolutional Neural Networks (CNNs) offer a powerful alternative by automatically learning spatial hierarchies of features directly from pixel data. Unlike fully connected neural networks, CNNs preserve the two-dimensional structure of images and use local connections, weight sharing, and pooling operations to reduce the number of parameters while maintaining important spatial information. This makes CNNs highly efficient and scalable for image-related tasks.

## IV. Methodology

### A. *Simple CNN Architecture*

In this implementation, a simple CNN model is implemented, consisting of three convolutional layers, one max pooling layer, followed by flattening and a fully connected (dense) layer:

**Network Structure:**

- **Input Layer:** 28×28 grayscale image (1 channel)
- **Convolutional Layer 1:** 3 filters, kernel size 3×3, padding=1, output size: 28×28×3
- **Convolutional Layer 2:** 6 filters, kernel size 3×3, padding=1, output size: 28×28×6
- **Convolutional Layer 3:** 9 filters, kernel size 3×3, padding=1, output size: 28×28×9
- **Max Pooling Layer:** 2×2 pool size with stride 2, output size: 14×14×9
- **Flatten Layer:** Flattens the output to a vector of size $9 \times 14 \times 14 = 1764$
- **Fully Connected Layer:** Linear layer mapping from 1764 inputs to 10 outputs (digit classes)

## B. *Convolutional Layers*

The convolutional layers implement 2D convolution operations to extract spatial features from input images. Each convolutional layer applies multiple learnable filters to detect different patterns such as edges, corners, and textures.

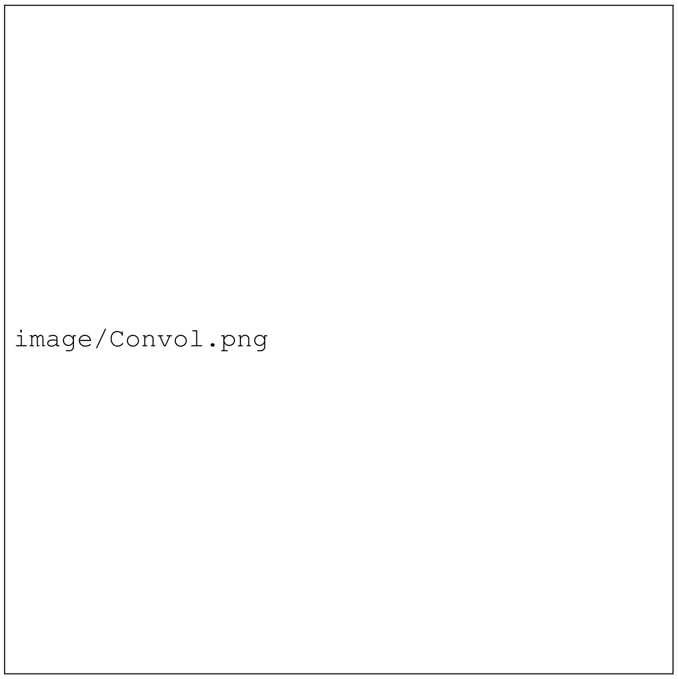

Fig. 1. Convolution operation

In many standard CNN implementations, convolutional kernels slide over input images or feature maps with a stride of one, leveraging the spatial structure where neighboring pixels are highly correlated. The convolution kernel approach in CNN differs somewhat from traditional convolution in probability theory or signal processing, particularly in how the kernel aligns with image boundaries and resolution, and is not flipped. This is similar to cross-correlation in image processing. During convolution, learnable kernels move across the input, performing element-wise multiplications followed by summations at each position to extract meaningful local features.

Padding is also used to add extra pixels around the input image or feature map boundaries, which helps preserve spatial dimensions after convolution and prevents the loss of information at the edges. By applying padding, the network can better capture features near the borders, ensuring consistent output sizes and enabling deeper architectures without excessive downsampling.

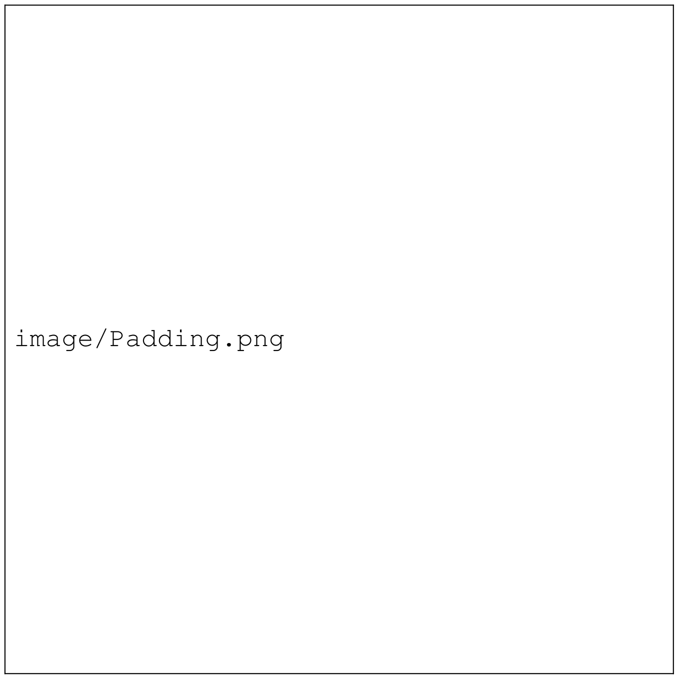

Fig. 2. Padding used in Convolution

## C. *Pooling Layers*

Pooling layer is usually placed between different convolution layers to reduce the size of the output data and still preserve the important features of images. In practice, a pooling layer with size = (2,2), stride = 2, and padding = 0 is often used so that the output width and height of the data are reduced half while depth is unchanged.

## D. *Flatten and Dense Layers (Linear)*

The flatten layer converts the multi-dimensional output of the convolutional and pooling layers into a one-dimensional vector, making it suitable for input into dense (fully connected) layers. Dense layers perform linear transformations on their inputs, followed by nonlinear activation functions to introduce complexity and enable the network to learn intricate patterns.

Mathematically, each neuron in a dense layer computes an output $y$ as:

$$y = f(\mathbf{w}^\top \mathbf{x} + b)$$

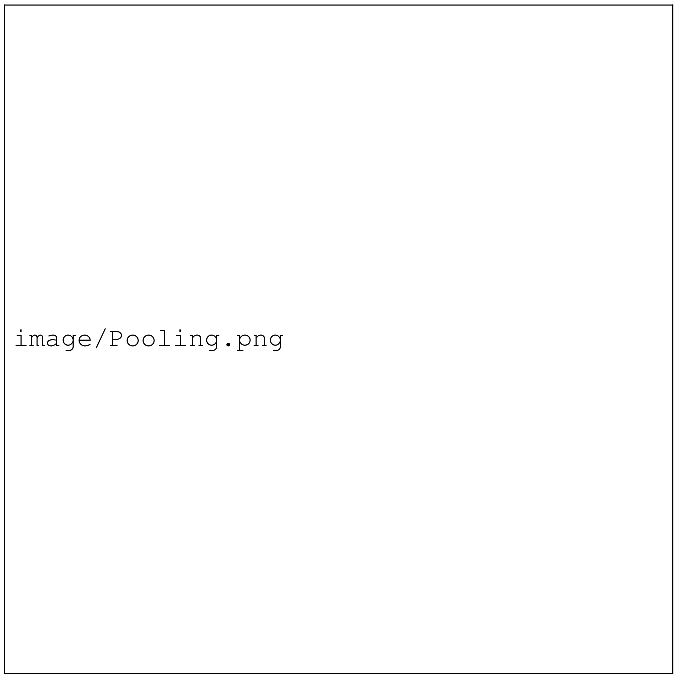

Fig. 3. Pooling operation with 2×2 window and stride 2

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, $b$ is the bias term, and $f$ is the activation function (e.g., ReLU or softmax).

## *E. Activation Functions*

The network uses two main activation functions to introduce non-linearity:

**ReLU (Rectified Linear Unit):** This function is applied after each convolutional layer and dense layer. It outputs the input directly if it is positive; otherwise, it outputs zero:

$$\text{ReLU}(x) = \max(0, x)$$

ReLU helps the network learn complex patterns by adding non-linearity and mitigates the vanishing gradient issue, which in turn allows for faster and more effective training.

**Softmax:** Applied in the final output layer, softmax converts raw scores into probabilities across the multiple classes:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

where $K = 10$ corresponds to the number of digit classes. By normalizing outputs to sum to one, softmax creates a sort of probability distribution for each class.

## *F. Loss Function*

In this implementation, the cross-entropy loss function is implemented. This shows the difference between the predicted probability distribution produced by the network and the true probability.

**Cross-Entropy Loss:** For a single sample with predicted probabilities $\hat{y}$ and true labels $y$, the loss is computed as:

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{1}$$

where $K$ is the number of classes, $y_i$ is the true label indicator (1 for the correct class, 0 otherwise), and $\hat{y}_i$ is the predicted probability for class $i$.

For a batch of $N$ samples, the average cross-entropy loss is calculated as:

$$L_{\text{batch}} = \frac{1}{N}\sum_{n=1}^{N} L_n = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} y_{n,i} \log(\hat{y}_{n,i}) \tag{2}$$

where $y_{n,i}$ and $\hat{y}_{n,i}$ denote the true label and predicted probability for class $i$ of sample $n$, respectively.

Minimizing Cross-entropy loss during training encourages the model to produce probability distributions that closely match the true labels, improving classification accuracy.
Additionally, this loss function is differentiable, which makes it suitable for optimization via gradient-based methods such as stochastic gradient descent (SGD) or its variants.

## *G. Optimization Algorithm*

A Stochastic Gradient Descent (SGD) is used as the optimization algorithm to update model parameters during training.

**SGD Parameter Update:** For each trainable parameter $\theta$, the update rule is defined as:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial L}{\partial \theta_t} \tag{3}$$

where $\eta$ is the learning rate and $\frac{\partial L}{\partial \theta_t}$ denotes the gradient of the loss function with respect to the parameter $\theta$ at time step $t$.

**Learning Rate:** A configurable learning rate controls the size of each parameter update step. Adjusting the learning rate balances the speed of convergence with training stability.

**Batch Processing:** The implementation processes data in mini-batches, computing gradients over multiple samples before updating parameters. This approach offers several advantages:

- More stable gradient estimates compared to single-sample updates
- Improved computational efficiency through vectorized operations
- Enhanced convergence properties in non-convex optimization landscapes

Gradient computation accumulated via AutoGrad in this implementation provides accurate and simple gradient handling. This also helps with coding since after implementing AutoGrad, we don't have to write a backward() pass for each and every module we add.

## V. Implementation

### A. AutoGrad

The automatic differentiation system forms the backbone of this CNN implementation, enabling efficient gradient computation through the computational graph. The AutoGrad implementation follows a reverse-mode differentiation approach, similar to PyTorch's design philosophy. The AutoGrad is based on the AutoGrad implemented by Andrej Kapathy in his series of LLM from scratch https://www.youtube.com/@AndrejKarpathy/videos

*1) Core Tensor Class:* The Tensor class serves as the fundamental building block, encapsulating both data and gradient information: Each tensor maintains:

- **Data structure**: Nested lists representing multi-dimensional arrays
- **Gradient tracking**: requires_grad flag and accumulated gradients
- **Computational graph**: References to parent nodes (_prev) and backward function (_backward)

*2) Computational Graph Construction:* Operations between tensors automatically construct a computational graph by setting up backward functions and parent relationships.

*3) Backward Propagation Algorithm:* The backward() method implements reverse-mode automatic differentiation through topological sorting:

- **Topological Ordering**: Build dependency graph to ensure gradients flow in correct order
- **Gradient Initialization**: Set output gradient to ones
- **Reverse Traversal**: Execute backward functions from output to inputs

*4) Gradient Accumulation:* The system supports gradient accumulation through the _add_gradients helper function, essential for:

- Parameter updates across multiple samples
- Handling tensors used multiple times in computation
- Broadcasting scenarios with dimension reduction

*5) Broadcasting and Shape Handling:* This implementation handles complex broadcasting scenarios, particularly in matrix operations. The __matmul__ method includes sophisticated broadcasting logic:

- **Shape Padding**: Align tensor dimensions for batch operations
- **Gradient Reduction**: Properly reduce gradients for broadcasted dimensions
- **Dimension Compatibility**: Validate matrix multiplication constraints

*6) Parameter Management:* The Parameter class extends Tensor with requires_grad=True by default, designed specifically for learnable parameters:

*7) Activation Functions with AutoGrad:* Activation functions are implemented as standalone functions that maintain gradient tracking:

*8) Memory and Efficiency Considerations:* While this implementation prioritizes clarity over optimization, several design choices support reasonable efficiency:

- **Manual Evaluation:** Gradients computed only when backward() is called
- **Graph Pruning**: Only tensors with requires_grad=True participate in gradient computation
- **In-place Operations**: Limited support for memory-efficient operations

*9) Integration with CNN Components:* The AutoGrad system seamlessly integrates with CNN layers through the Module base class, which automatically tracks parameters and enables recursive gradient computation across the entire network architecture. This design allows complex neural networks to be trained with minimal boilerplate code while maintaining full control over the gradient flow.

### B. Optimizer

The optimizer module provides the parameter update mechanism for training this CNN. Following the strategy pattern, we implement a flexible architecture that supports different optimization algorithms while maintaining a consistent interface.

*1) Optimizer Base Class:* The Optimizer abstract base class defines the common interface for all optimization algorithms. The design provides several key functionalities:

- **Parameter Discovery**: Automatically collects all trainable parameters from the model hierarchy
- **Unified Interface**: Consistent step() and zero_grad() methods across all optimizers
- **Extensibility**: Abstract update_param() method allows easy implementation of new algorithms

*2) Parameter Collection Mechanism:* The optimizer automatically discovers parameters through the model's parameters() method, which recursively traverses the module hierarchy. This approach ensures that complex nested architectures (such as this CNN with multiple convolutional and linear layers) have all their parameters automatically included in the optimization process.

*3) Stochastic Gradient Descent Implementation:* The SGD implementation follows the standard gradient descent update rule as mention in Equation **??**

*4) Recursive Parameter Updates:* The compute_update function handles the nested list structure of the tensor implementation through recursive traversal. This design choice allows the optimizer to work seamlessly with tensors of arbitrary dimensionality

*5) Training Loop Integration:* The optimizer integrates into the training loop through a standard three-step process:

- **Zero Gradients**: Clear accumulated gradients from previous iteration
- **Backward Pass**: Compute gradients via backpropagation
- **Parameter Update**: Apply computed gradients to update parameters

*6) Gradient Management:* The optimizer provides two essential gradient management methods:

- **zero_grad()**: Resets all parameter gradients to zero, preventing accumulation between training steps
- **step()**: Iterates through all parameters and applies updates only to those with requires_grad=True

This design supports gradient accumulation scenarios where multiple forward passes contribute to a single parameter update, commonly used in mini-batch training or when dealing with memory constraints.

*7) Extensibility and Future Optimizers:* The modular design facilitates easy implementation of advanced optimizers such as Adam, RMSprop, or AdaGrad. Each new optimizer would only need to implement the update_param method while inheriting the parameter discovery and gradient management functionality.

Several design choices support reasonable training efficiency.

- **In-place Updates**: Parameter data is modified directly, avoiding unnecessary memory allocation
- **Conditional Updates**: Only parameters with non-None gradients are processed
- **Manual Evaluation**: Gradient computations occur only when explicitly requested

The optimizer's integration with the AutoGrad system ensures that gradient computation and parameter updates work seamlessly together, providing a robust foundation for training the CNN architecture.

## VI. Dataset

### A. Dataset Preparation

The MNIST dataset preprocessing is handled through a custom DataLoader implementation that provides comprehensive data management capabilities. The dataset preparation process involves several key steps:

The dataloader expects a directory structure following the pattern:

```
data_dir/
 train/
    0/  # Images for digit 0
    1/  # Images for digit 1
    ...
 test/
 0/
 1/
 ...
```

Each loaded image undergoes a standardized preprocessing pipeline:

The normalization step is crucial for CNN training stability, as it:

- Prevents gradient explosion/vanishing
- Ensures consistent input ranges across different images
- Improves convergence speed and training stability

### B. Data Splitting Strategy

*1) Train-Validation Split:* The dataloader implements a randomized train-validation split to ensure unbiased model evaluation.

1) **Grayscale Conversion**: Ensures consistent single-channel input
2) **Dimension Validation**: Confirms 28×28 pixel dimensions for MNIST compatibility
3) **Normalization**: Scales pixel values from [0, 255] to [0, 1] range
4) **Tensor Conversion**: Reshapes flat pixel data into 2D arrays

**Data Loading:** Raw MNIST data is loaded using PIL. Each image is converted from the original 28×28 pixel format with integer values ranging from 0-255.

## VII. Training and Evaluation

### A. Training Process

In order to update model weights, the training process iterates through the training dataset in batches using a forward pass and backpropagation. In this implementation it misses early stopping, but the weights are saved.

**Forward Pass:** As mention, each batch will go through a custom CNN model. The output stores the logits for each class.

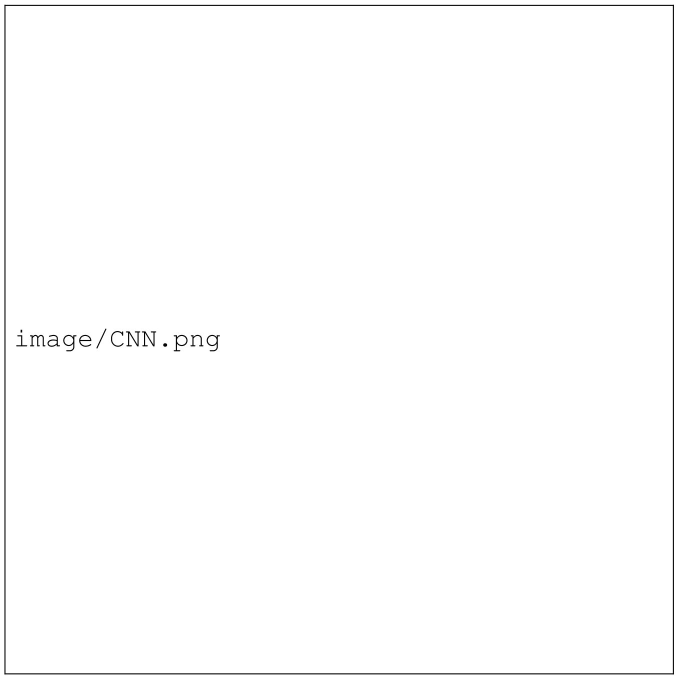

Fig. 4. Simple CNN

The logits will be used by the CrossEntropy loss function to calculate the total loss. Also, the loss is the average loss over each batch.

**Backpropagation:** When call loss.backward(), every node (Tensor) inside the topological gradient tree will be updated using the SGD algorithm.

**Training Configuration:** Simple edits through the confix.txt file will set up the training pipeline.

```
data_dir = data/reduced_mnist_png
batch_size = 32
epochs = 10
learning_rate = 0.01
optimizer_type = SGD
loss_function = cross_entropy
seed = 11
image_type = jpg
train_val_split = 0.8
log_file = training_log.txt
save_dir = saved
```

**Performance Monitoring:** During training, the following metrics are tracked and logged:

- Training loss per iteration (batch) and training/validation loss per epoch
- Validation accuracy

Finally, when done, the weights of the model will be saved in a file as well.

### B. Model Evaluation

The trained model is evaluated on distinct datasets to assess performance and generalization capability

*In the implementation, actually only a few samples of the Validation split (320 samples) is used to evaluate the performance, since each iteration took too long to run. (About 10s per iteration with a batch of 32), similarly for the Test split.*

### C. Evaluation Criteria

Accuracy is used as the default criterion.

For each prediction, the class with the highest probability from the SoftMax output is compared against the true class. The implementation provides detailed evaluation statistics, including:

- Validation accuracies.
- Training convergence statistics (epochs trained, loss improvement)
- Model architecture summary and hyperparameter configuration

## VIII. Results and Analysis

### A. Experimental Setup

The CNN model was trained and evaluated on a subset of the MNIST dataset using the following hardware and software configuration:

- CPU: Intel Core i5-1038NG7
- RAM: 16 GB

**Dataset Split:**

- Total samples: 11,000
- Training set: 9,000 samples
- Validation set: 1,000 samples
- Test set: 2,000 samples (only 320 samples were used)
- Batch size: 32

**Training Configuration:**

- Maximum epochs: 10
- Learning rate: 0.01
- Optimizer: Stochastic Gradient Descent (SGD)

### B. Training Results

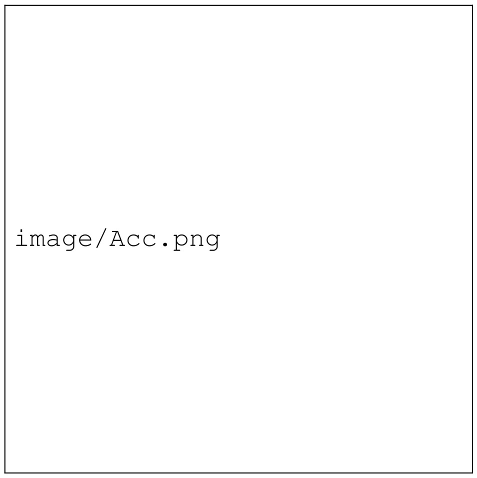

Fig. 5. Training and validation accuracy over epochs

Figure **??** illustrates the progression of training and validation accuracy across epochs. Figure **??** displays the corresponding loss curves. Although there is a slight increase in the validation loss during training, both curves follow a similar trend, suggesting that the model is generalizing reasonably well. This behavior indicates that the model architecture is relatively lightweight, which may limit its ability to fully capture complex patterns in the data. Nevertheless, the model achieved a peak validation accuracy of 84.06%, demonstrating effective learning with minimal overfitting.

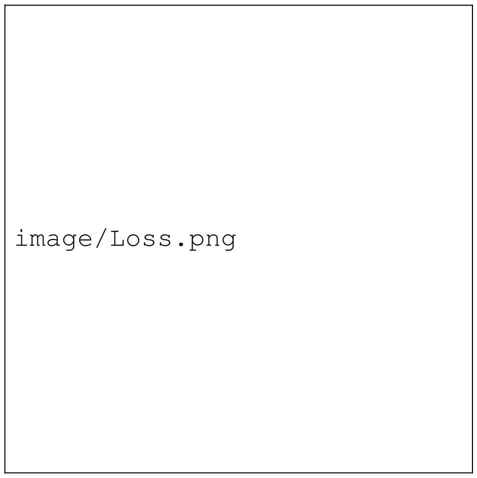

Fig. 6. Training and validation loss over epochs

Look at this, we can see test performance was observed after the fourth epoch, with the following metrics for (epoch 1-5)

- Validation accuracy: 84.06%
- Training loss: 1.17
- Validation loss: 1.13

## IX. Conclusion and Future Work

This study presents a from-scratch implementation of a Convolutional Neural Network (CNN) tailored for handwritten digit classification using the MNIST dataset. The network incorporates core components such as convolutional, pooling, and dense layers, along with non-linear activation functions, modeled after the LeNet-5 architecture. Despite training on a limited dataset of 11,000 samples due to computational constraints, the model achieved a strong validation accuracy of 84.06%, demonstrating its effectiveness.

Future work will focus on scaling the implementation to utilize the full MNIST dataset, optimizing computational performance, and experimenting with more advanced CNN architectures. Additionally, exploring different hyperparameter settings and incorporating regularization techniques may further improve model generalization and robustness.