Contents

1 Problem: 1

LeNet-5, introduced by Yann LeCun et al. in the 1998 paper Gradient-Based Learning Applied to Document Recognition, is a pioneering convolutional neural network (CNN) architecture designed for handwritten digit recognition. It consists of seven layers (excluding input) that include convolutional layers, subsampling (pooling) layers, and fully connected layers. The architecture follows a hierarchical pattern: the input (28×28 grayscale images) is processed through two convolutional layers with 5×5 filters, each followed by average pooling layers, reducing spatial dimensions while maintaining essential features.

1.1 Part a

- **Problem :** Implement LeNet-5 using PyTorch library in Python. Train the network using MNSIT dataset. Plot accuracy vs epochs and loss vs epochs. Consider a number of epochs till the training procedure converges.

- The code for implementing LeNet-5 was in file name "LeNet_5.py" which is attached in zip file. Below is the values and graphs which is showing accuracy vs epochs and loss vs epoch.

```
Epoch [1] Training Loss: 0.2870, Training Accuracy: 91.57% | Test Loss: 0.1124, Test Accuracy: 96.55%
Epoch [2] Training Loss: 0.0942, Training Accuracy: 97.12% | Test Loss: 0.0723, Test Accuracy: 97.90%
Epoch [3] Training Loss: 0.0623, Training Accuracy: 98.04% | Test Loss: 0.0629, Test Accuracy: 98.10%
Epoch [4] Training Loss: 0.0471, Training Accuracy: 98.58% | Test Loss: 0.0561, Test Accuracy: 98.21%
Epoch [5] Training Loss: 0.0382, Training Accuracy: 98.80% | Test Loss: 0.0505, Test Accuracy: 98.34%
Epoch [6] Training Loss: 0.0326, Training Accuracy: 98.97% | Test Loss: 0.0516, Test Accuracy: 98.35%
Epoch [7] Training Loss: 0.0283, Training Accuracy: 99.07% | Test Loss: 0.0545, Test Accuracy: 98.27%
Epoch [8] Training Loss: 0.0233, Training Accuracy: 99.27% | Test Loss: 0.0477, Test Accuracy: 98.55%
Epoch [9] Training Loss: 0.0212, Training Accuracy: 99.33% | Test Loss: 0.0474, Test Accuracy: 98.54%
Epoch [10] Training Loss: 0.0175, Training Accuracy: 99.42% | Test Loss: 0.0483, Test Accuracy: 98.55%
Epoch [11] Training Loss: 0.0164, Training Accuracy: 99.47% | Test Loss: 0.0470, Test Accuracy: 98.57%
Epoch [12] Training Loss: 0.0134, Training Accuracy: 99.58% | Test Loss: 0.0495, Test Accuracy: 98.58%
Epoch [13] Training Loss: 0.0130, Training Accuracy: 99.56% | Test Loss: 0.0512, Test Accuracy: 98.42%
Epoch [14] Training Loss: 0.0125, Training Accuracy: 99.58% | Test Loss: 0.0480, Test Accuracy: 98.76%
Epoch [15] Training Loss: 0.0100, Training Accuracy: 99.65% | Test Loss: 0.0535, Test Accuracy: 98.56%
Epoch [16] Training Loss: 0.0106, Training Accuracy: 99.64% | Test Loss: 0.0571, Test Accuracy: 98.51%
Epoch [17] Training Loss: 0.0094, Training Accuracy: 99.68% | Test Loss: 0.0560, Test Accuracy: 98.59%
Epoch [18] Training Loss: 0.0076, Training Accuracy: 99.75% | Test Loss: 0.0595, Test Accuracy: 98.43%
Epoch [19] Training Loss: 0.0090, Training Accuracy: 99.72% | Test Loss: 0.0570, Test Accuracy: 98.57%
```

Figure 1: Values of training and testing

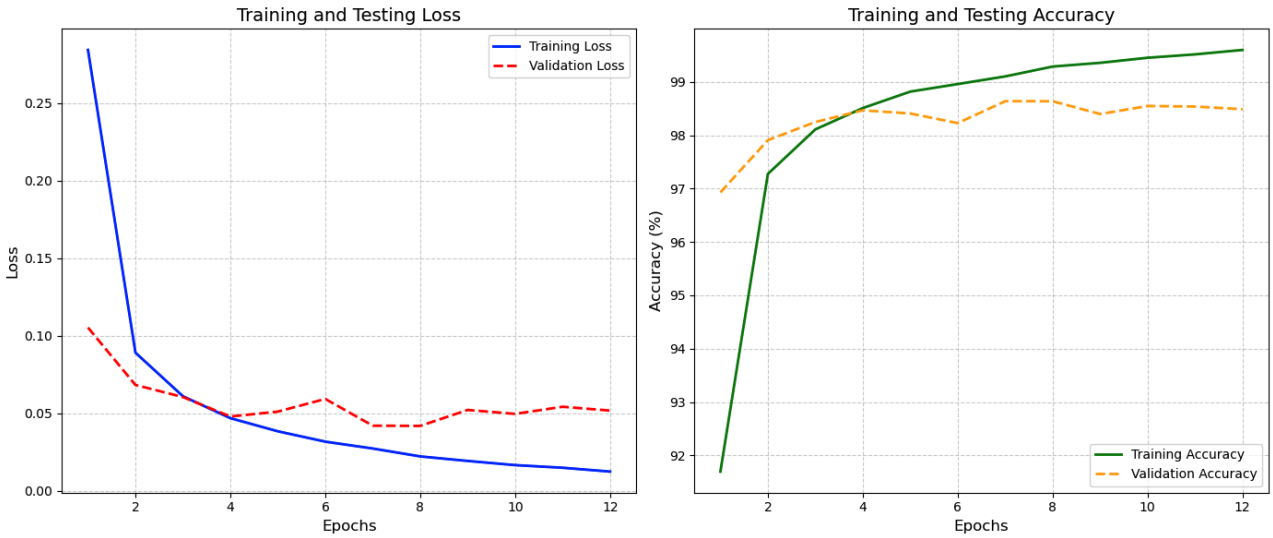

Figure 2: Accuracy vs Epochs and Loss vs Epochs

The graph consists of two subplots:

- **Left Plot: Training and Testing Loss**
  - **Blue Solid Line**: Training Loss
  - **Red Dashed Line**: Tetsing Loss
- **Right Plot: Training and Testing Accuracy**
  - **Green Solid Line**: Training Accuracy
  - **Orange Dashed Line**: Testing Accuracy

Each plot shows the performance of the model over multiple epochs, helping to analyze the training progress and detect overfitting or underfitting.

1.2 Part b

- **Problem :** Replace all convolution layers with depth-wise separable convolution. Plot accuracy vs epochs and loss vs epochs. Consider a number of epochs till the training procedure converges.

- In the adapted LeNet-5 architecture, which replace all convolution layers with depth-wise separable convolutions, are used to enhance efficiency by decomposing standard convolution into two steps:
  - Depth-wise convolution
  - Point-wise convolution

  This approach significantly reduces computational complexity and the number of parameters while preserving essential feature extraction capabilities.

  The implementation of this modified architecture is implemented in Depth_Wise_Separable_Convolution.py file within the provided zip file.

  Below are the graphs illustrating accuracy and loss trends over multiple training epochs.

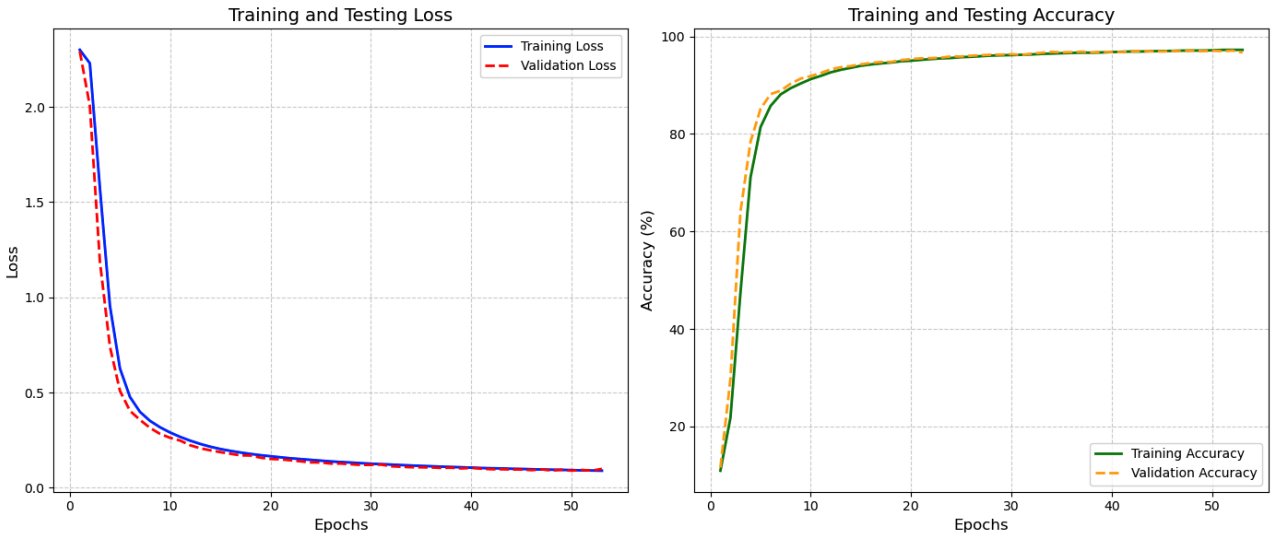

Figure 3: Accuracy vs Epochs and Loss vs Epochs

The graphs visualize the model's performance over multiple epochs.

- **Left Plot: Loss vs. Epochs**
  - **Blue Solid Line**: Training Loss
  - **Red Dashed Line**: Testing Loss
- **Right Plot: Accuracy vs. Epochs**

Figure 4: Values of training and testing

* **Green Solid Line**: Training Accuracy
* **Orange Dashed Line**: Testing Accuracy

These graphs help in analyzing the training progress and identifying potential overfitting or underfitting.

1.3 Part c

- **Problem :** What is the total number of multiply operations and addition operation with regular convolution (network obtain in a) and depth wise separable convolution (network obtain in b)?

  We focus solely on the significant multiply/add operations in the three convolutional layers 1, 2, 3 and the two fully connected layers at the end (FC and Output), while generally disregarding the computational costs of the average pooling due to their minimal impact compared to the convolutions.

- **Regular Convolution :**

- **Layer 1:** $32 \times 32 \times 1(Input) \rightarrow 28 \times 28 \times 6$ (Output), Kernel $= 5 \times 5$

$$\begin{aligned} \text{Number of Output elements} &= 6 \times 28 \times 28 = 4704, \\ \text{Multiplication of each output} &= 1 \times (5 \times 5) = 25, \\ \text{Multiplications} &= 4704 \times 25 = 117,600, \\ \text{Additions} &= 117,600 - 4704 = 112,896. \end{aligned}$$

- **Layer 2:** $14 \times 14 \times 6(Input) \rightarrow 10 \times 10 \times 16$ (Output), Kernel $= 5 \times 5$

$$\begin{aligned} \text{Number of Output elements} &= 16 \times 10 \times 10 = 1600, \\ \text{Multiplication of each output} &= (6 \times 5 \times 5) = 150, \\ \text{Multiplications} &= 1600 \times 150 = 240,000, \\ \text{Additions} &= 240,000 - 1600 = 238,400. \end{aligned}$$

- **Layer 3:** $5 \times 5 \times 16(Input) \rightarrow 1 \times 1 \times 120$ (Output), Kernel $= 5 \times 5$

$$\begin{aligned} \text{Number of Output elements} &= 120, \\ \text{Multiplication of each output} &= (16 \times 5 \times 5) = 400, \\ \text{Multiplications} &= 120 \times 400 = 48,000, \\ \text{Additions} &= 48,000 - 120 = 47,880. \end{aligned}$$

- **FC:** $120 \rightarrow 84$

$$\begin{aligned} \text{Multiplications} &= 120 \times 84 = 10,080, \\ \text{Additions} &= 84 \times (120 - 1) = 9996. \end{aligned}$$

- **Output:** $84 \rightarrow 10$

$$\begin{aligned} \text{Multiplications} &= 84 \times 10 = 840, \\ \text{Additions} &= 10 \times (84 - 1) = 830. \end{aligned}$$

- **Total Regular Convolution:**

$$\begin{aligned} \text{Total Multiplications} &= 416,520, \\ \text{Total Additions} &= 410,002. \end{aligned}$$

- **Depthwise-Separable Convolution Version**
  - **Layer 1:** $32 \times 32 \times 1(Input) \rightarrow 28 \times 28 \times 6$ (Output)

$$\begin{aligned} \text{Depthwise Multiplications} &= 19,600, \\ \text{Additions} &= 18,816, \\ \text{Pointwise Multiplications} &= 4704, \\ \text{Additions} &= 0, \\ \text{Total (Layer\_1, DSC)} &= 24,304 \textit{ and } 18,816. \end{aligned}$$

  - **Layer 2:** $14 \times 14 \times 6(Input) \rightarrow 10 \times 10 \times 16$ (Output)

$$\begin{aligned} \text{Depthwise Multiplications} &= 15,000, \\ \text{Additions} &= 14,400, \\ \text{Pointwise Multiplications} &= 9600, \\ \text{Additions} &= 8000, \\ \text{Total (Layer\_2, DSC)} &= 24,600 \textit{ and } 22,400. \end{aligned}$$

– **Layer 3:** $5 \times 5 \times 16(Input) \rightarrow 1 \times 1 \times 120$ (Output)

$$\begin{aligned} \text{Depthwise Multiplications} &= 400, \\ \text{Additions} &= 384, \\ \text{Pointwise Multiplications} &= 1920, \\ \text{Additions} &= 1800, \\ \text{Total (Layer\_3, DSC)} &= 2320 \textit{ and } 2184. \end{aligned}$$

– **FC:** $120 \rightarrow 84$

$$\begin{aligned} \text{Multiplications} &= 10,080, \\ \text{Additions} &= 9996. \end{aligned}$$

– **Output:** $84 \rightarrow 10$

$$\begin{aligned} \text{Multiplications} &= 840, \\ \text{Additions} &= 830. \end{aligned}$$

– **Total Depthwise-Separable Convolution:**

$$\begin{aligned} \text{Total Multiplications} &= 62,144, \\ \text{Total Additions} &= 54,226. \end{aligned}$$

1.4 Part d

- **Problem :** Prepare a mathematical function (f) which maps number of operations in a regular convolutional layer to depth-wise separable convolutional layer. Specifically, derive f(x, other parameters) = y, where x is the number of operations in a regular convolution and y is the number operation in depth-wise operation resulting in same output size.

- **Standard Convolution: Computational Complexity**

  Consider a convolutional layer with the following parameters:

  – C_{in}: Number of input channels
  – C_{out}: Number of output channels
  – $K_h \times K_w$: Kernel size
  – $I_h \times I_w$: Spatial dimensions of the output feature map

  The total number of multiplication operations required for a conventional 2D convolution is computed as:

$$O_m = C_{in} \cdot C_{out} \cdot K_h \cdot K_w \cdot I_h \cdot I_w. \tag{1}$$

- **Depth-wise Separable Convolution: Computational Complexity**

  It is composed of two sequential operations:

  1. **Depth-wise Convolution**: Each input channel undergoes convolution independently using a $K_h \times K_w$ filter.
  2. **Point-wise Convolution**: A 1×1 convolution integrates information across channels, generating C_{out} output channels.

1.5 Depth-wise Convolution

The total multiplication operations for depth-wise convolution are given by:

$$O_{dc} = C_{in} \cdot K_h \cdot K_w \cdot I_h \cdot I_w. \tag{2}$$

- **Point-wise Convolution**

  Since each output channel is formed by applying a 1×1 convolution across all input channels, the total number of multiply operations is:

$$O_{pc} = C_{in} \cdot C_{out} \cdot I_h \cdot I_w. \tag{3}$$

- **Total Operations for Depth-wise Separable Convolution**

  Thus, the overall computational cost of depth wise separable convolution is:

$$O_{dsc} = C_{in} \cdot K_h \cdot K_w \cdot I_h \cdot I_w + C_{in} \cdot C_{out} \cdot I_h \cdot I_w. \tag{4}$$

- We derive value that transforms the computational cost of standard convolution (O_m) into that of depth-wise separable convolution (O_{dsc}) as follows.

  From the formula for standard convolution:

$$O_m = C_{in} \cdot C_{out} \cdot K_h \cdot K_w \cdot I_h \cdot I_w. \tag{5}$$

  Rearranging for $C_{in} \cdot I_h \cdot I_w$:

$$C_{in} \cdot I_h \cdot I_w = \frac{O_m}{C_{out} \cdot K_h \cdot K_w}. \tag{6}$$

  Substituting into the depthwise-separable convolution equation:

$$O_{dsc} = \left(\frac{O_m}{C_{out} \cdot K_h \cdot K_w} \right) K_h \cdot K_w + \left(\frac{O_s}{C_{out} \cdot K_h \cdot K_w} \right) C_{out}. \tag{7}$$

  Simplifying:

$$O_{dsc} = O_s \cdot \frac{K_h \cdot K_w + C_{out}}{K_h \cdot K_w \cdot C_{out}}. \tag{8}$$

  Thus, the transformation will be as follows:

$$=> O_s \cdot \frac{K_h \cdot K_w + C_{out}}{K_h \cdot K_w \cdot C_{out}}. \tag{9}$$

2 Problem: 2

Consider a Multi-head Transformer Algorithm as shown in the figure below. The algorithm consists of one encoder and one decoder (similar to the one discussed in the class). Assume 8 attention heads, embedding size of 256. Assume 16-bit floating point numbers for inputs, outputs and all intermediate operations. Also notice that some of the blocks have been skipped.

- Logical implementation of transformer is attached in zip file with name **"transformer.py"** .

2.1 Part a

- **Problem :** Implement the transformer algorithm.
- **Introduction**

  The Transformer model is a neural network architecture that employs self-attention mechanisms rather than traditional recurrence or convolution operations. This design allows for efficient parallelization and improved handling of long-range dependencies in sequences. Our single-layer implementation consists of an encoder and a decoder, which together transform an input sequence into an output sequence.

- **Encoder Block**

  The encoder processes the input sequence and generates a contextualized representation. The key components of the encoder include:

  Embedding and Positional Encoding : Each token in the sequence is represented as a d_{model}-dimensional vector. To preserve positional information, sinusoidal functions are added to the embeddings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), \tag{10}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), \tag{11}$$

  where *pos* represents the token's position and i indexes the embedding dimensions.

- **Multi-Head Self-Attention**

  Multi-head attention consists of multiple self-attention computations, each with its own learned weight matrices. The final output is obtained by concatenating the results of all attention heads and applying a linear transformation.

  The process can be described as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

  where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

  where:

  - W_i^Q, W_i^K, W_i^V are learnable weight matrices for each attention head.
  - Q, K, V are the query, key, and value matrices from the input.
  - W^O is a weight matrix applied after concatenation.

  Self-attention is computed using the scaled dot-product formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{12}$$

  where $d_k = \frac{d_{model}}{h}$.

- **Residual Connection and Normalization**

  To stabilize training, a residual connection is applied followed by layer normalization:

$$Y = \text{LayerNorm}(X + \text{Attention}(Q, K, V)). \tag{13}$$

- **Feed-Forward Network (FFN)**

  A two-layer fully connected network with a ReLU activation is applied:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2. \tag{14}$$

  Again, a residual connection and normalization are applied:

$$Z = \text{LayerNorm}(Y + \text{FFN}(Y)). \tag{15}$$

- **Decoder Block**

  The decoder generates the output sequence using three main sub-layers:

- **Masked Multi-Head Self-Attention**

  This layer prevents positions from attending to future positions, ensuring autoregressive generation:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V, \tag{16}$$

  where M is a mask that enforces causality.

- **Cross-Attention (Encoder-Decoder Attention)**

  The decoder uses multi-head attention over the encoder's output. Queries come from the decoder, while keys and values originate from the encoder:

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{17}$$

- **Feed-Forward Network**

  Similar to the encoder, the FFN is applied, followed by residual connections and normalization.