**LeNet-5 Model on the MNIST Dataset**

**Venkata Harshitha Bathala**

**1002204412**

AGENDA:

* INTRODUCTION
* LeNet-5 Model Architecture
* IMPLEMENTATION
* RESULTS
* ANALYSIS
* REFERENCE

INTRODUCTION:

LeNet-5 is a pioneering convolutional neural network (CNN) model designed for image classification tasks, particularly handwritten character recognition. Developed by Yann LeCun in 1998, LeNet-5 was a breakthrough in deep learning, demonstrating how CNNs could automatically extract spatial features from images, a key advancement over traditional feature engineering. This model consists of convolutional and pooling layers that progressively capture complex patterns, followed by fully connected layers for classification. The architecture’s effectiveness lies in its ability to handle variations in handwritten digits, making it ideal for datasets like MNIST, which contains grayscale images of handwritten numbers. By implementing LeNet-5 on this dataset, we can observe CNNs' efficiency and accuracy in image recognition tasks.

LeNet-5 Model Architecture:

1. **First Convolutional Layer (conv1):**
   * The layer takes in a single-channel image (grayscale) and applies six 5×55 \times 55×5 filters, producing six feature maps.
   * The feature maps are activated with the hyperbolic tangent (Tanh) activation function (act1) to introduce non-linearity.
   * Code: self.conv1 = nn.Conv2d(1, 6, kernel\_size=5) followed by self.act1 = nn.Tanh()
2. **First Pooling Layer (pool1):**
   * After the first convolution, an average pooling layer with a 2×22 \times 22×2 kernel and a stride of 2 downsamples each feature map, reducing their spatial dimensions by half.
   * This pooling operation helps to reduce computational complexity and improve translational invariance.
   * Code: self.pool1 = nn.AvgPool2d(kernel\_size=2, stride=2)
3. **Second Convolutional Layer (conv2):**
   * This layer applies sixteen 5×55 \times 55×5 filters to the pooled output, capturing more complex features across the input.
   * Like the previous layer, it is followed by a Tanh activation function (act2).
   * Code: self.conv2 = nn.Conv2d(6, 16, kernel\_size=5) followed by self.act2 = nn.Tanh()
4. **Second Pooling Layer (pool2):**
   * Another average pooling layer with a 2×22 \times 22×2 kernel and a stride of 2 further reduces the spatial dimensions of each feature map.
   * Code: self.pool2 = nn.AvgPool2d(kernel\_size=2, stride=2)
5. **Flattening Layer:**
   * The output from the pooling layer is flattened to a one-dimensional vector (x.view(x.size(0), -1)) to feed into the fully connected layers.
6. **First Fully Connected Layer (fc1):**
   * The first fully connected layer receives the flattened vector, which has 16 \* 4 \* 4 (256) input units, and connects to 120 neurons.
   * A Tanh activation function (act3) is applied here as well.
   * Code: self.fc1 = nn.Linear(16 \* 4 \* 4, 120) followed by self.act3 = nn.Tanh()
7. **Second Fully Connected Layer (fc2):**
   * This layer has 120 input units connected to 84 neurons and also uses Tanh activation (act4).
   * Code: self.fc2 = nn.Linear(120, 84) followed by self.act4 = nn.Tanh()
8. **Output Layer (fc3):**
   * The final layer has 84 input units connected to 10 output neurons, each representing a class in a classification task (e.g., digits 0–9).
   * Code: self.fc3 = nn.Linear(84, 10)

The forward method defines how data flows through each layer in sequence, applying convolutions, activations, pooling, and fully connected transformations to predict the class of each input image. This implementation follows the classic structure of LeNet-5, using PyTorch to create a modular and flexible neural network.

Implementation

The **LeNet-5** model implementation was developed using the **PyTorch** framework with support from the **torchvision** library for data preprocessing and model training. This section outlines the key stages of data preparation, the model architecture, and the training and evaluation processes, along with the hyperparameters selected to optimize performance.

**1. Framework and Libraries**

* **PyTorch**: Used for building and training the neural network.
* **torchvision**: Supports data loading, transformations, and dataset handling, which simplifies the data preprocessing pipeline.

**2. Data Preprocessing Steps**

To ensure consistent input formatting, the following steps were performed on the dataset:

* **Image Resizing**: Each input image is resized to 32×3232 \times 3232×32 pixels to match the model’s expected input dimensions.
* **Normalization**: All pixel values are scaled to the range [−1,1][-1, 1][−1,1] by applying a mean and standard deviation of 0.5. This step standardizes the data distribution, which helps stabilize training.
* **Tensor Conversion**: The images are converted to PyTorch tensors to make them compatible with the model and the PyTorch framework.

**3. Training Process and Hyperparameters**

The training phase involves optimizing the LeNet-5 model on the dataset through a combination of forward and backward passes.

**Hyperparameters**:

* **Batch Size**: 64
* **Number of Epochs**: 10
* **Learning Rate**: 0.001
* **Optimizer**: Adam, which adapts the learning rate during training.
* **Loss Function**: Cross-Entropy Loss, which combines softmax and negative log likelihood, is used to compute the loss for multi-class classification.

**Training Workflow**:

* **Forward Pass**: For each batch of images, the model predicts outputs by processing the data through each layer of the network.
* **Loss Calculation**: Cross-Entropy Loss is calculated by comparing the model’s predictions to the true labels.
* **Backward Pass and Optimization**: The gradients are computed through backpropagation, and the model’s weights are updated using the Adam optimizer to minimize the loss.
* **Accuracy Tracking**: After processing each batch, predictions are compared to actual labels to calculate training accuracy. Loss and accuracy values for each epoch are saved for later analysis.

**Epoch-based Training**:

* Training runs for the specified number of epochs, with each epoch consisting of multiple forward and backward passes over the training dataset. At the end of each epoch, the training loss and accuracy are recorded to evaluate performance improvement.

**4. Evaluation and Testing**

After each epoch, the model’s performance is evaluated on a separate test dataset to monitor its generalization capability.

**Testing Workflow**:

* **Forward Pass**: Each image in the test set is passed through the trained model without computing gradients (using torch.no\_grad()), reducing memory usage and computation time.
* **Loss and Accuracy Calculation**: The loss and accuracy are calculated in the same way as during training to assess the model’s generalization on unseen data.
* **Performance Metrics**: At the end of each epoch, the average test loss and accuracy are printed, providing a snapshot of the model’s ability to generalize and classify new data correctly.

Results

The **LeNet-5** model was trained for **10 epochs** on the dataset using a batch size of **64**. Here’s a summary of the results, including the **training and test losses** and **accuracy metrics** recorded during each epoch.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Epoch** | **Train Loss** | **Train Accuracy** | **Test Loss** | **Test Accuracy** |
| 1 | 0.2835 | 91.68% | 0.0944 | 97.03% |
| 2 | 0.0863 | 97.36% | 0.0703 | 97.88% |
| 3 | 0.0585 | 98.18% | 0.0610 | 97.99% |
| 4 | 0.0452 | 98.57% | 0.0456 | 98.43% |
| 5 | 0.0382 | 98.73% | 0.0556 | 98.21% |
| 6 | 0.0321 | 98.95% | 0.0510 | 98.36% |
| 7 | 0.0267 | 99.14% | 0.0428 | 98.64% |
| 8 | 0.0239 | 99.22% | 0.0488 | 98.39% |
| 9 | 0.0215 | 99.28% | 0.0478 | 98.55% |
| 10 | 0.0185 | 99.39% | 0.0437 | 98.64% |

A graph with a line and a line graph

Description automatically generated

A graph with a line and a line

Description automatically generated with medium confidence

**Confusion Matrix:**

* Evaluation of Classification Performance: It helps assess how well the model is performing in classifying the different classes in the dataset.
* Understanding Errors: It allows you to analyze the types of errors made by the model (e.g., false positives and false negatives).
* Performance Metrics Calculation: It serves as a basis for calculating various performance metrics, such as accuracy, precision, recall, and F1-score.

**precision recall f1-score support**

**0 0.99 0.99 0.99 980**

**1 0.99 0.99 0.99 1135**

**2 0.99 0.99 0.99 1032**

**3 0.98 0.99 0.99 1010**

**4 0.98 0.99 0.99 982**

**5 0.98 0.98 0.98 892**

**6 0.99 0.99 0.99 958**

**7 0.99 0.99 0.99 1028**

**8 0.97 0.99 0.98 974**

**9 0.99 0.97 0.98 1009**

**accuracy 0.99 10000**

**macro avg 0.99 0.99 0.99 10000**

**weighted avg 0.99 0.99 0.99 10000**

A graph of a number

Description automatically generated with medium confidence

he LeNet-5 model demonstrated high effectiveness in recognizing handwritten digits, achieving a training accuracy of **99.39%** and a test accuracy of **98.64%.** The model's convergence behavior showed consistent improvements in both training and testing metrics, with minimal signs of overfitting.

Prediction

The predict\_minst function takes a trained LeNet-5 model and an image path as input, preprocesses the image, and predicts the digit represented in the image.

* Input Image: /content/9.png

A screen shot of a computer

Description automatically generated

* Predicted Digit: The function outputs the predicted digit 9.

Analysis:

**Training Accuracy and Loss Trend:**

* The model demonstrates a strong learning capacity, as evidenced by the steady increase in training accuracy, which starts at **91.68%** in the first epoch and gradually reaches **99.39%** by the final epoch. This upward trend suggests the model's effectiveness in capturing complex patterns in the data.
* Training loss also shows a corresponding decrease, beginning at **0.2835** and declining to **0.0185** by the tenth epoch. This consistent reduction highlights the model's ability to minimize prediction errors on the training set, indicating successful weight updates by the optimizer (Adam) with each epoch.

**Generalization Performance on the Test Set:**

* The test accuracy also follows a positive trajectory, starting from an initial high accuracy of **97.03%** and reaching **98.64%** by the last epoch. This high initial test accuracy suggests that the model's architecture aligns well with the dataset, allowing it to generalize effectively from the beginning.
* Test loss remains low throughout the training, with values in the range of approximately **0.0944 to 0.0437**. Notably, there are slight increases in test loss around **Epochs 5 and 8**, which could indicate minor instances of overfitting or noise in the data. However, the loss stabilizes quickly, pointing to a well-calibrated model that maintains a consistent performance on unseen data.

**Model Convergence:**

* After **Epoch 5**, improvements in both training and test accuracy begin to taper off, showing marginal increases only. This plateau in accuracy and minimal change in test loss suggest that the model has nearly converged by this point, and further epochs yield diminishing returns in terms of performance gains.
* The convergence indicates that the chosen number of **epochs (10)** is adequate, as the model has reached its optimal performance without unnecessary computational cost from additional training.

**Effective Regularization and Overfitting Prevention:**

* The LeNet-5 model shows minimal overfitting throughout the training, evidenced by the stability of the test accuracy and loss. The lack of a substantial gap between training and test accuracy suggests that the model generalizes well to new data, a desirable characteristic indicating it has learned relevant features without excessive memorization.
* The stable test loss around **0.04**, despite minor fluctuations, confirms that the model remains resilient to overfitting, likely due to the regularizing effect of batch normalization and dropout, inherent in the LeNet-5 structure.

**Overall Model Effectiveness:**

* The final accuracy levels (**99.39%** on the training set and **98.64%** on the test set) indicate that LeNet-5 is highly effective for this classification task, demonstrating its ability to capture essential patterns in the input images. These results affirm that LeNet-5, though simple compared to modern architectures, remains powerful and efficient for image classification tasks, especially with smaller and more straightforward datasets.

**Potential for Further Optimization:**

* Although the model performed well, minor performance improvements could be explored through hyperparameter tuning or additional training epochs. Since convergence was nearly achieved around Epoch 5, experimenting with early stopping might optimize training time.
* Alternative regularization techniques or optimizers, such as adding weight decay or using SGD with momentum, could also be tested for potential performance gains or improved convergence speed.

Reference:

* [Implementing Yann LeCun’s LeNet-5 in PyTorch | by Eryk Lewinson | Towards Data Science](https://towardsdatascience.com/implementing-yann-lecuns-lenet-5-in-pytorch-5e05a0911320)
* [Integrated Segmentation and Recognition of Hand-Printed Numerals](https://proceedings.neurips.cc/paper_files/paper/1990/file/e46de7e1bcaaced9a54f1e9d0d2f800d-Paper.pdf)
* [SCALABLE STACKING AND LEARNING FOR BUILDING DEEP ARCHITECTURES](https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/DSN-ICASSP2012.pdf?msockid=1308dce6b4dd638537bdcf1ab5a86230)