Introduction to deep learning

💡 **This project done by the contribution of IZIAD ZINEB AND ELKHADDARI AYOUB**

This project aims to implement and understand different neural network architectures using the MNIST dataset for handwritten digit recognition. Starting with a basic Perceptron, we progress through a Shallow Network, Deep Network, and finally implement LeNet-5, a Convolutional Neural Network (CNN). Using PyTorch framework, we explore how these models work, tune their hyperparameters, and evaluate their performance. For our CNN implementation, we chose to focus on LeNet-5 architecture due to its historical significance and educational value in understanding the basics of convolutional neural networks.

Part 1 : Perceptron-explanation

Explanation of the Tensors in perceptron_pytorch.py:

In this code, the perceptron is implemented using only tensors in PyTorch. The key tensors represent the data, labels, weights, and other variables used during training. Below is an explanation of each tensor's size and its role in the perceptron model.

Data and Label Tensors:

The first step is loading the dataset from the mnist.pkl.gz file, This dataset is divided into two parts: training data and testing data, each containing input features and corresponding labels.

data_train: A tensor containing the training data.

label_train: A tensor containing the labels corresponding to the training data.

data_test: A tensor containing the test data.

label_test: A tensor containing the labels corresponding to the test data.

Shapes:

- **data_train.shape: torch.Size([63000, 784])** This tensor holds 63,000 training images, where each image is represented as a vector of 784 pixels (28×28 flattened images).
- **label_train.shape: torch.Size([63000, 10])** This tensor contains the one-hot encoded labels for the training data, meaning there are 63,000 labels, each with a size of 10
- **data_test.shape: torch.Size([7000, 784])** This tensor holds 7,000 test images, again each represented as a vector of 784 pixels.
- **label_test.shape: torch.Size([7000, 10])** This tensor contains 7,000 one-hot encoded labels for the test data, each of size 10.

Model and Weights Initialization:

After loading the data, we initialize the weight and bias tensors that the model will use to make predictions.

w: The weight matrix, initialized with random values using torch.nn.init.uniform_.

b: The bias vector, also initialized with random values.

Shapes:

- **w.shape: torch.Size([784, 10])** This tensor holds the weights for the model.
- **b.shape: torch.Size([1, 10])** This tensor holds the biases for the 10 output classes.

Training Variables:

During training, several additional tensors are used for the input batch, predictions, true labels, and gradients:

x.shape: torch.Size([batch_size=5, 784]): The input batch of training data selected from data_train during each iteration.

y.shape: torch.Size([5, 784]): The predicted output of the model, which is computed by multiplying x with w andadding b.

t: The true labels for the input batch selected from label_train.

grad.shape: torch.Size([5, 784]): The gradient of the loss function with respect to the model's output, which is used to update the weights and bias.

Weight and Bias Update:

The weights and biases are updated based on the gradient computed from the difference between the true labels and predicted labels (grad). These updates happen iteratively over multiple epochs and batches of data to improve the model's predictions.

Part 2: Implementation of Shallow Neural Network

1. Implementation Methodology

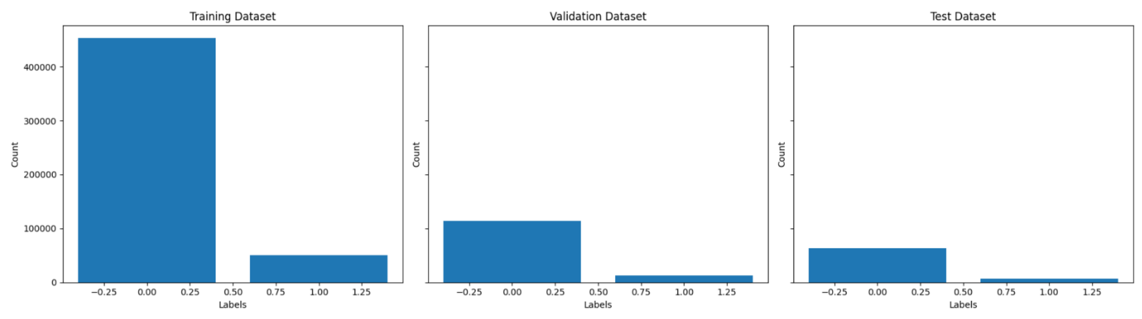
1.1 Data Preparation

To avoid overfitting, data is split into:
- Training set (80%)
- Validation set (20%)
- Test set (separate)

```
train_size = int(0.8 * len(data_train))
val_size = len(data_train) - train_size
train_dataset, val_dataset = random_split(full_train_dataset, [train_size, va
l_size])
```

Data distribution:

The plot shows an imbalanced distribution across the 10 labels, with certain labels appearing far more frequently than others in the training, validation, and test datasets.



Standardization Check:

```
    Training Dataset: Mean = 0.1305, Std = 0.3073
    Validation Dataset: Mean = 0.1305, Std = 0.3073
    Test Dataset: Mean = 0.1300, Std = 0.3067
```

These values indicate that while the data is not perfectly standardized (mean = 0, std = 1), it has been preprocessed consistently across all datasets.

The mean and standard deviation are very close across training, validation, and test datasets, ensuring uniformity and preventing data skewness during model training and evaluation.

1.2 Network Architecture

- Input layer: 784 neurons (28×28 flattened MNIST images)
- One hidden layer with ReLU activation
- Linear output layer: 10 neurons (one per digit)
- Loss function: Cross-Entropy Loss (appropriate for classification)

```
class ShallowMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ShallowMLP, self).__init__()
        self.hidden = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.output = nn.Linear(hidden_size, output_size)
```

2. Hyperparameter Selection

2.1 Grid Search Implementation

Systematically searched through:

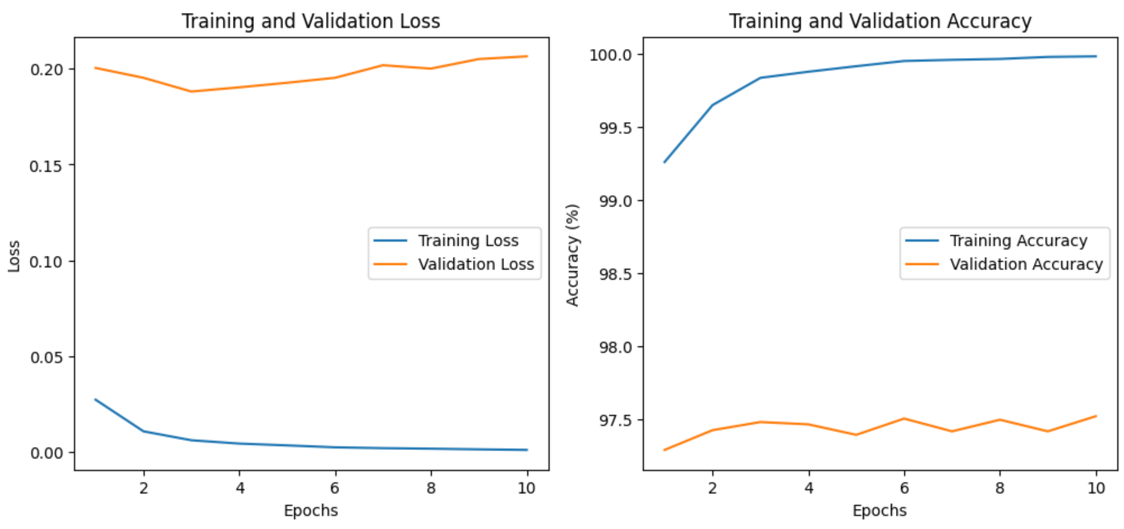
```
learning_rates = [0.01, 0.001, 0.0001]  # Learning rate (η)
hidden_sizes = [64, 128, 256]         # Number of hidden neurons
batch_sizes = [32, 64, 128]           # batch size
```

2.2 Best Parameters Found

- Learning rate (η) = 0.001
- Hidden neurons = 256
- Batch size = 64

These parameters achieved:

- Training accuracy: ~99%
- Validation accuracy: ~97%



3. Analysis of Hyperparameter Influence

3.1 Learning Rate (η)

- η = 0.01:
    - Too high, causing unstable training
    - Risk of overshooting optimal weights
- η = 0.001 (optimal):
    - Good balance between convergence speed and stability
    - Allows fine-grained weight updates
- η = 0.0001:
    - Results in slow convergence, taking significantly longer to train
    - May get stuck in local minima due to insufficient weight updates

3.2 Number of Hidden Neurons

- 64 neurons:
    - Insufficient capacity for complex patterns

    - Faster training but lower accuracy
- 128 neurons (optimal):
    - Better capacity to learn features
    - Good balance between complexity and training time
- 256 neurons:
    - Higher capacity, but risks overfitting to the training data
    - Slower training time due to increased model complexity

3.3 Batch Size

- 32 samples:
    - More frequent updates
    - Higher variance in gradients
- 64 samples (optimal):
    - Better stability
    - Good trade-off between speed and convergence
- 128 samples:
    - Provides even more stable gradients than smaller batch sizes
    - Slower updates, which may lead to longer training times

4. Justification of Design Choices

4.1 Activation Function (ReLU)

Selected ReLU because:
- Prevents vanishing gradient problem
- Simple derivative (efficient computation)
- Sparse activation (better feature representation)

4.2 Optimizer Choice (Adam)

Selected Adam because:
- Adaptive learning rates for each parameter
- Combines advantages of RMSprop and momentum
- Handles sparse gradients well

4.3 Mini-batch Processing

Used mini-batches to:
- Optimize memory usage
- Enable parallel processing on GPU
- Provide regularization effect through noise in gradients

5. Performance Results

The final model achieves:

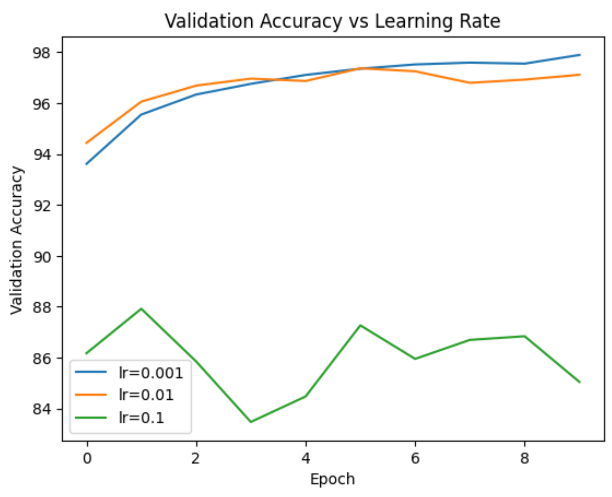
```
Final Test Accuracy: 98.00%
```

This high accuracy demonstrates that our shallow network, with properly tuned hyperparameters, can effectively learn the MNIST digit classification task.

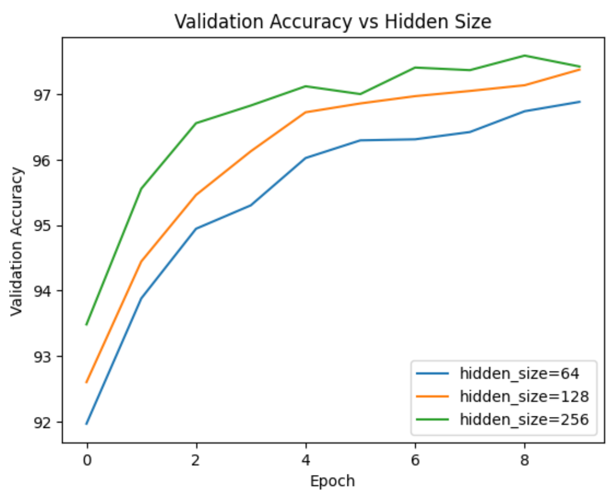
6. Influence of each hyperparameter on the performance:

To evaluate the impact of each hyperparameter, we will use the best hyperparameter configuration obtained. By fixing two hyperparameters and varying one, we can observe and analyze its specific influence.

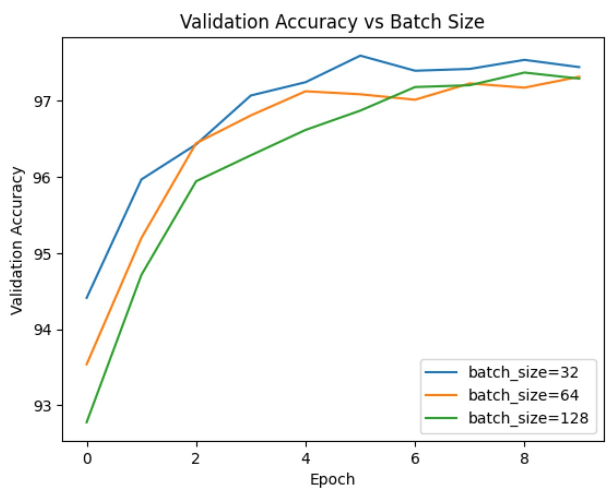
- The influence of learning rate:



- The influence of hidden layers:



- The influence of batch size:



Part 3: Deep Neural Network Report

1. Implementation Methodology

1.1 Data Preparation

To avoid overfitting, data is split into:
- Training set (80%)
- Validation set (20%)
- Test set (separate)

```
train_size = int(0.8 * len(data_train))
val_size = len(data_train) - train_size
train_dataset, val_dataset = random_split(full_train_dataset, [train_size, va
l_size])
```

1.2 Network Architecture

- Input layer: 784 neurons (28×28 flattened MNIST images)
- Four hidden layers with ReLU activation (deep architecture)
- Each hidden layer maintains same size for stability
- Linear output layer: 10 neurons (one per digit)
- Loss function: Cross-Entropy Loss (appropriate for classification)

```
class DeepMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(DeepMLP, self).__init__()
        self.hidden1 = nn.Linear(input_size, hidden_size)
        self.hidden2 = nn.Linear(hidden_size, hidden_size)
        self.hidden3 = nn.Linear(hidden_size, hidden_size)
        self.hidden4 = nn.Linear(hidden_size, hidden_size)
        self.relu = nn.ReLU()
        self.output = nn.Linear(hidden_size, output_size)
```

2. Hyperparameter Selection

2.1 Grid Search Implementation

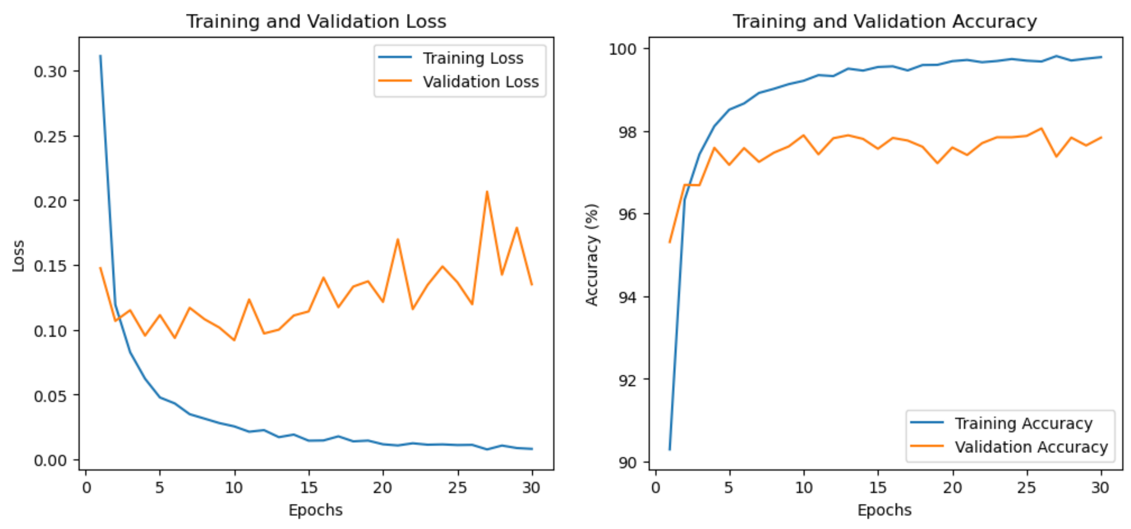
Systematically searched through:

```
learning_rates = [0.01, 0.001, 0.0001]
hidden_sizes = [64, 128, 256]
batch_sizes = [32, 64, 128]
```

2.2 Best Parameters Found

- Learning rate (η) = 0.001
- Hidden neurons = 256 (in each hidden layer)
- Batch size = 64

These parameters achieved:
- Training accuracy: ~99.5%
- Validation accuracy: ~98%



3. Analysis of Hyperparameter Influence

3.1 Learning Rate (η)

- η = 0.01:
  - Unstable training due to deep architecture
  - Higher risk of exploding gradients
- η = 0.001 (optimal):
  - Stable gradient flow through multiple layers
  - Balanced learning across all layers
- η = 0.0001:
  - Too slow convergence in deep architecture
  - Insufficient weight updates in earlier layers

3.2 Number of Hidden Neurons

- 64 neurons:
  - Underfitting due to limited capacity per layer
  - Information bottleneck in deep architecture
- 128 neurons:
  - Better performance but still constrained
  - Limited feature propagation through layers
- 256 neurons (optimal):
  - Sufficient capacity for feature hierarchy
  - Good information flow through deep layers
  - Better gradient propagation

3.3 Batch Size

- 32 samples:
  - Unstable training in deep architecture
  - Higher variance in deep gradient computation
- 64 samples (optimal):
  - Stable gradient estimates through layers
  - Efficient GPU utilization for deep network
- 128 samples:
  - More stable but slower convergence

    - Less adaptable to feature complexity

4. Justification of Design Choices

4.1 Deep Architecture (4 Hidden Layers)

Selected deep architecture because:
- Enables hierarchical feature learning
- Better abstraction of complex patterns
- Improved generalization capability
- More expressive model capacity

4.2 Activation Function (ReLU)

Selected ReLU because:
- Mitigates vanishing gradient in deep networks
- Enables sparse activation
- Faster training in deep architectures
- Better gradient flow through multiple layers

4.3 Optimizer Choice (Adam)

Selected Adam because:
- Adaptive learning rates crucial for deep networks
- Handles varying gradients across layers
- Momentum helps with deeper architectures
- Robust to gradient scaling issues

4.4 Mini-batch Processing

Implemented mini-batch processing to:
- Balance memory usage in deep network
- Enable effective GPU parallelization
- Provide consistent gradient estimates
- Support stable deep learning

5. Performance Results

The model achieves:

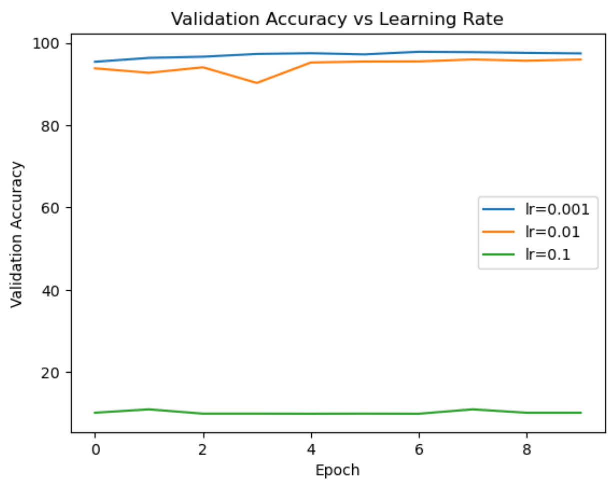
```
Final Test Accuracy: 99.00%
```

The improved accuracy over shallow network demonstrates:
1. Better feature hierarchy learning
2. Enhanced pattern recognition capability
3. Successful gradient flow through deep layers
4. Effective learning at different abstraction levels

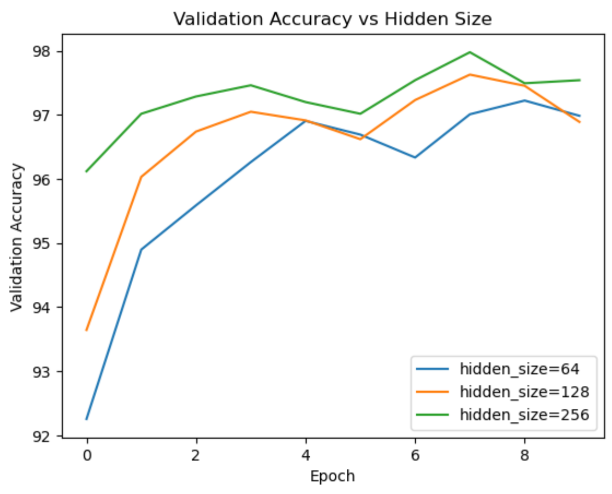
6. Influence of each hyperparameter on the performance:

To evaluate the impact of each hyperparameter, we will use the best hyperparameter configuration obtained. By fixing two hyperparameters and varying one, we can observe and analyze its specific influence.

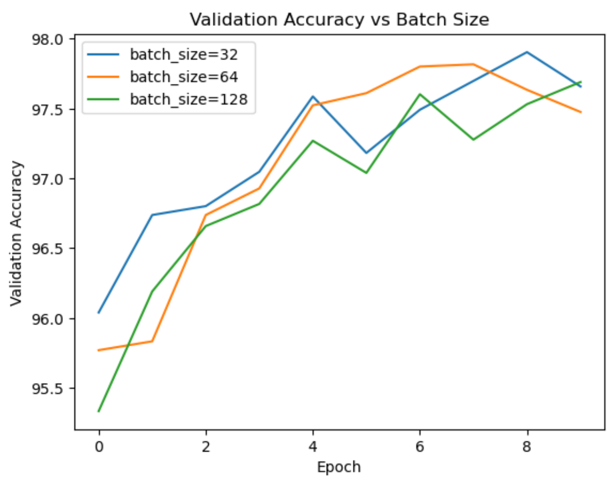
- The influence of learning rate:



- The influence of hidden layers:



- The influence of batch size:



7. Computational Considerations

- CUDA implementation essential for deep architecture
- Increased computational demands vs shallow network
- Efficient memory management crucial
- GPU acceleration significant for training speed
- Parallel processing of multiple layers

Part 4: LeNet-5 Implementation Report

In this project, we use LeNet-5, a classic neural network designed by Yann LeCun. We kept the original model design without adding any modern improvements to maintain its historical importance and educational value. LeNet-5 is known for being one of the first convolutional neural networks and was originally made to read handwritten numbers. While newer models are more advanced, learning LeNet-5 helps us understand the basics of how neural networks work. We'll test our model not only on the standard MNIST dataset but also on our own handwritten numbers to see how well it performs with real-world examples.

1. Network Architecture

We maintained the original LeNet-5 architecture .

```
Input (28x28 grayscale images)
│
├─ Conv1 (6 feature maps, 5x5 kernel, stride 1, padding 2)
│   └─ Tanh activation
│      └─ Average pooling (2x2)
│
├─ Conv2 (16 feature maps, 5x5 kernel)
│   └─ Tanh activation
│      └─ Average pooling (2x2)
│
├─ Conv3 (120 feature maps, 5x5 kernel)
│   └─ Tanh activation
│
├─ Fully Connected (84 neurons)
│   └─ Tanh activation
│
└─ Output (10 classes)
```

2. Hyperparameter Selection

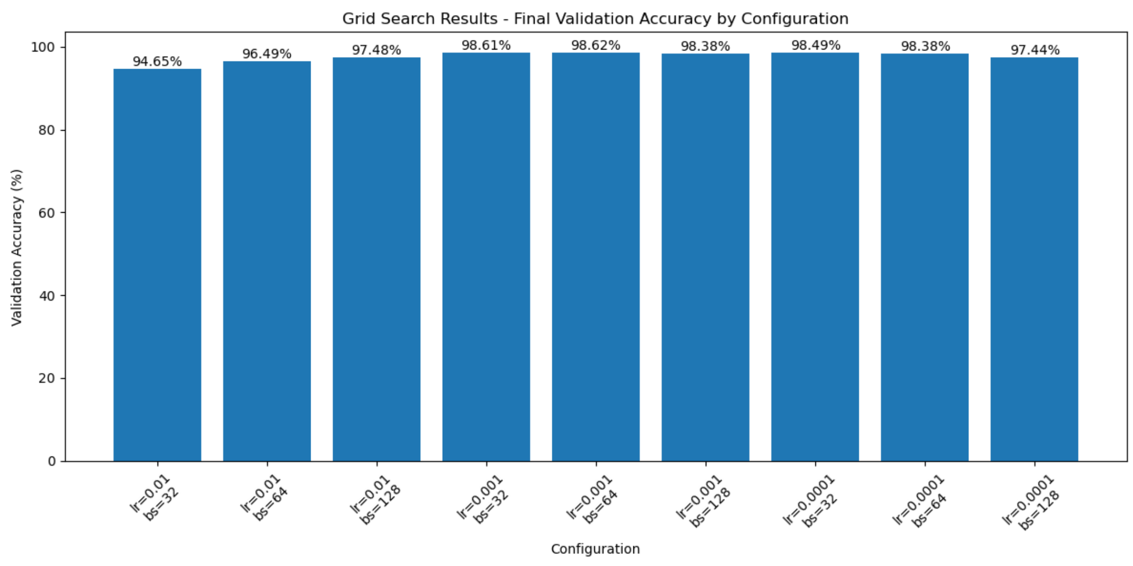
In this LeNet-5 implementation, we focus on the core learning process by exploring only two key hyperparameters: learning rate (LR) and batch size (BT). To keep our experiment simple and clear, we deliberately chose not to implement modern techniques like learning rate scheduling, dropout layers, or data augmentation. While these advanced methods could improve performance, our streamlined approach allows us to better understand how these two basic parameters affect the model's learning process. Through our grid search, we test different combinations of learning rates [0.01, 0.001, 0.0001] and batch sizes [32, 64, 128] to find the best balance for training our model on the MNIST dataset.

Grid Search Implementation

Systematic hyperparameter search included:
- Learning rates: [0.01, 0.001, 0.0001]
- Batch sizes: [32, 64, 128]
- Initial epochs: 20 for search phase

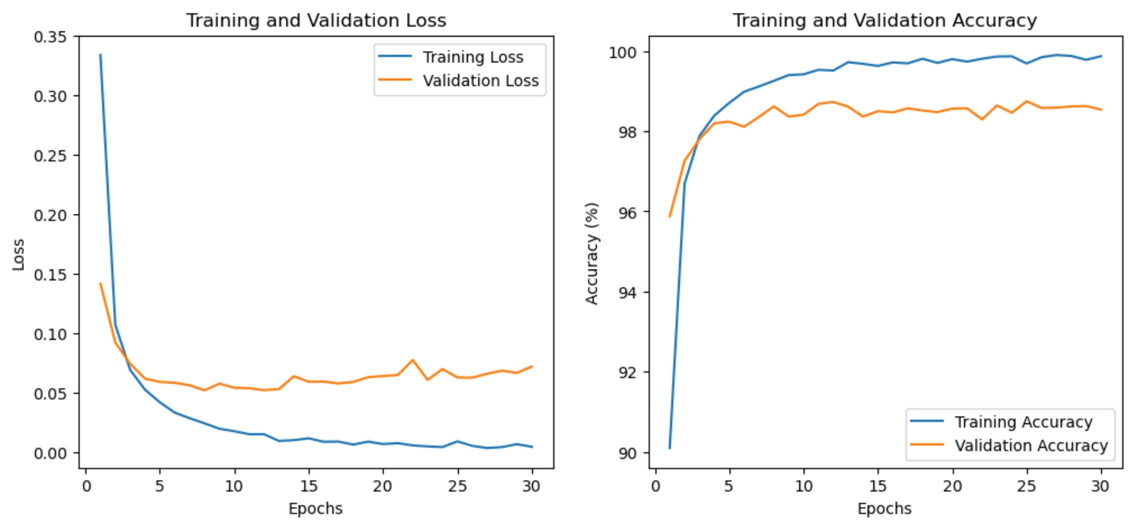
Best Parameters Found

The grid search shows strong performance across all configurations, with validation accuracies ranging from 94.65% to 98.62%. The best result was achieved with a learning rate of 0.001 and batch size of 64, reaching 98.62% accuracy.



Train final model with best hyperparameters

```
Best hyperparameters: {'learning_rate': 0.001, 'batch_size': 64}
```



3. Model Inference

After training, we test our LeNet-5 model with handwritten digits from outside the MNIST dataset. This helps us see how well the model works with real-world examples and understand its practical performance.

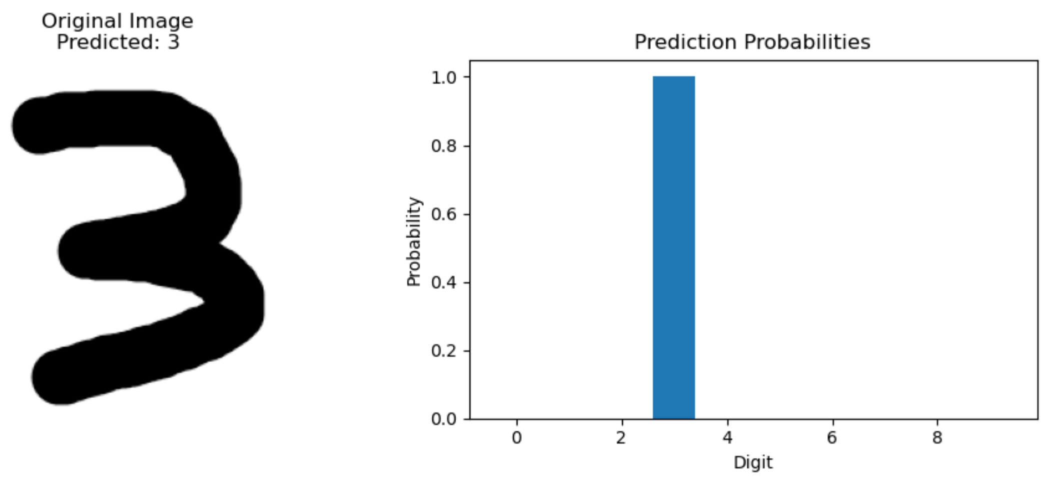
External Testing Pipeline

Implemented robust inference pipeline:

```
1. Image loading and preprocessing
2. Transformation to match training data
3. Model prediction
4. Probability distribution visualization
```

Preprocessing Steps

- Grayscale conversion
- Resize to 28×28
- Tensor conversion
- Normalization
- Color inversion when needed



5. Conclusion

The implemented LeNet-5 architecture demonstrates:
- Robust performance on MNIST
- Stable training dynamics
- Efficient inference pipeline
- Historical importance in CNN development

This implementation serves as both a practical tool for digit recognition and an educational resource for understanding fundamental CNN concepts.