# Basic Of Deep Learning - Mid Semester Project Part 2

Ofir Almog (ID. 207918731), Adir Edri (ID. 206991762)

Submitted as mid-semester project
report for Basic Of Deep Learning course, Colman, 2024

## 1 Introduction

In this project, we continue our work on implementing neural networks for recognizing hand-sign digits, now utilizing PyTorch instead of a basic NumPy implementation from our previous work. This second part extends our previous binary classification model into a more comprehensive solution, demonstrating the practical application of deep learning frameworks in real-world scenarios.

### 1.1 Data

The Sign Language Digits dataset remains consistent with our previous work, containing 5,000 grayscale images with dimensions of 28x28 pixels. Each image represents a hand gesture corresponding to digits from 0 to 9, with pixel values ranging from 0 to 255. While in our previous implementation we used NumPy for data pre-processing, we now leverage PyTorch's capabilities for more efficient data handling and processing.

### 1.2 Problem

This project is divided into two main challenges:

1. Binary Classification with PyTorch:
   - Pre-processing and normalizing 28x28 pixel grayscale images
   - Implementing a fully-connected neural network architecture
   - Training the model to achieve high accuracy while preventing overfitting
2. Multi-class Classification:
   - Comprehensive feature extraction from hand sign images
   - Robust fully-connected network design
   - Systematic approach to handling multiple digit representations
   - Ensuring high classification performance across all digit classes

# 2 Solution

## 2.1 General Approach

Our solution leverages PyTorch's capabilities to implement both binary and multiclass classification models. While maintaining the conceptual approach from our previous NumPy implementation, we now utilize PyTorch's built-in modules and optimizers for more efficient implementation. The solution is structured in two phases: first implementing a binary classifier, then extending to a multiclass model, both using fully-connected neural networks.

## 2.2 Design

The implementation consists of two main phases, reflecting our binary and multiclass classification tasks. Each phase shares core components but differs in specific architectural choices:

- **Framework Transition (Common to Both Tasks):**
  - Transition from NumPy arrays to PyTorch tensors
  - Utilization of PyTorch's autograd for automatic differentiation
  - Implementation of custom Dataset and DataLoader classes for efficient data handling
- **Binary Classification Architecture:**
  - Input layer: 784 neurons (28x28 flattened images)
  - Single hidden layer: 128 neurons
  - Output layer: 2 neurons (one for each class)
  - Activation functions: ReLU for hidden, Softmax for output
  - Loss function: Binary Cross-Entropy
- **Multiclass Classification Architecture:**
  - Input layer: 784 neurons (28x28 flattened images)
  - Two hidden layers: 128 and 64 neurons
  - Output layer: 10 neurons (one per digit)
  - Activation functions: ReLU for hidden, Softmax for output
  - Loss function: Categorical Cross-Entropy
- **Training Framework (Adapted for Each Task):**
  - Optimizer: Adam with task-specific learning rates
  - Batch processing: Mini-batch training with PyTorch DataLoader
  - Binary task: 32 batch size, focused validation strategy
  - Multiclass task: 64 batch size, comprehensive validation across all classes

### 2.2.1 Technical Implementation

The PyTorch implementation addresses key technical aspects for both classification tasks:

1. **Data Processing:**
   - **Binary Classification:** Selection of two classes, conversion to PyTorch tensors (784-dimensional vectors), normalization to [0,1] range, and binary label encoding.
   - **Multiclass Classification:** Processing all ten classes with the same tensor conversion, normalization, and one-hot encoding for labels.
2. **Model Architecture Implementation:**
   - **Binary Model:** Single hidden layer (nn.Linear), binary classification head, dropout (0.3) for regularization.
   - **Multiclass Model:** Two hidden layers, ten-neuron output layer, additional dropout (0.4) between hidden layers.
3. **Training Pipeline:**
   - **Common Components:** Early stopping, learning rate scheduling, GPU acceleration when available.
   - **Task-Specific Adjustments:** Balanced batch sampling for binary, stratified sampling for multiclass.

The training process follows this structure for both tasks:

```
# PyTorch Training Loop (Both Classifications)
1. Initialize model, optimizer, loss function
2. For each epoch:
    - Set model to training mode
    - For each batch in training data:
        * Forward pass, calculate loss, backward pass
        * Update parameters
    - Evaluate on validation set
    - Update learning rate if scheduled
    - Check early stopping criteria
```

Key differences in implementation between tasks:

- Binary: binary cross-entropy loss, two-class accuracy focus
- Multiclass: categorical cross-entropy, per-class metrics tracking
- Validation strategy adapts to the number of classes
- Memory management scales with task complexity

## 2.3 Base Model

The base model for this project is designed for multiclass classification, where the goal is to classify images representing hand-sign digits from 0 to 9. This model serves as the foundation for further experimentation and is built using a fully connected neural network architecture.

| Parameter | Base Model |
|---|---|
| Input Layer Neurons | 784 |
| Hidden Layer(s) | 128 |
| Output Layer Neurons | 10 |
| Learning Rate | 0.01 |
| Batch Size | 32 |
| Epochs | 10 |
| Optimizer | SGD |
| Loss Function | Cross-Entropy |

Table 1: Hyperparameters for the Base Model

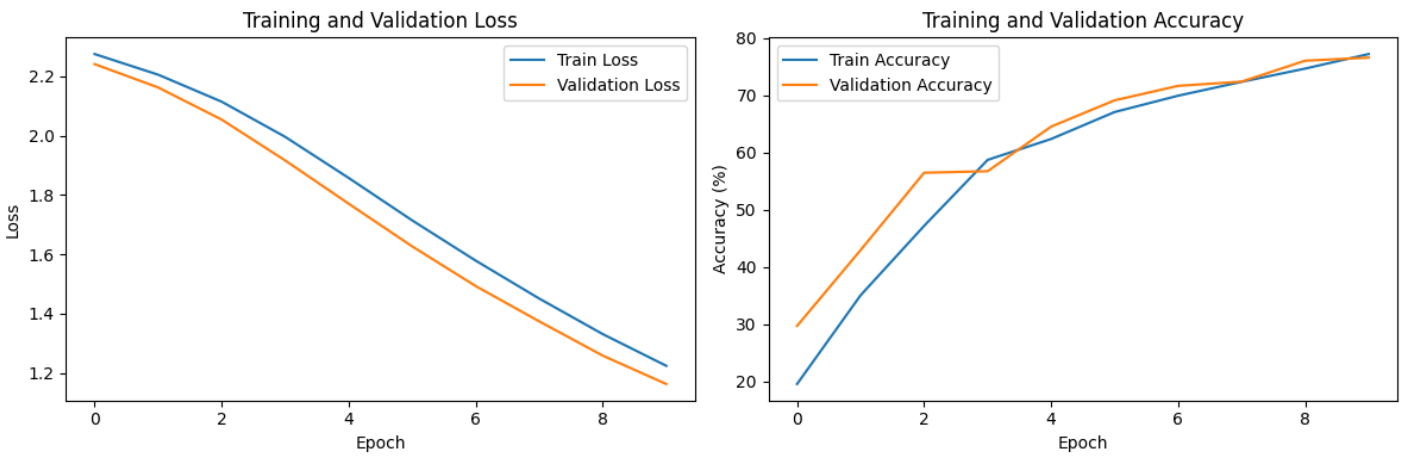


Figure 1: Training and Validation Loss and Accuracy for the Multiclass Model

| Metric | Base Model |
|---|---|
| Training Accuracy | 75.49% |
| Validation Accuracy | 75.60% |
| Average Precision | 0.75 |
| Average Recall | 0.76 |
| Average F1-Score | 0.75 |

Table 2: Evaluation Metrics for the Base Model

The base model demonstrates strong performance with consistent accuracy across all digit classes. While the overall metrics are slightly lower due to the complexity of the multiclass task, the minimal gap between training and validation accuracies suggests good generalization, without significant overfitting.

## 2.4 First Experiment

In our first experiment, we focused on architectural modifications to improve the performance of our base models.

| Layer | Base Model | Experiment 1 |
|---|---|---|
| Input | $784 \rightarrow 256$ | $784 \rightarrow 256$ |
| Hidden 1 | $256 \rightarrow 128$ | $256 \rightarrow 128$ |
| Hidden 2 | $128 \rightarrow 64$ | $128 \rightarrow 64$ |
| Output | $64 \rightarrow 10$ | $64 \rightarrow 10$ |

Table 3: Architecture Comparison: Base vs Experiment 1

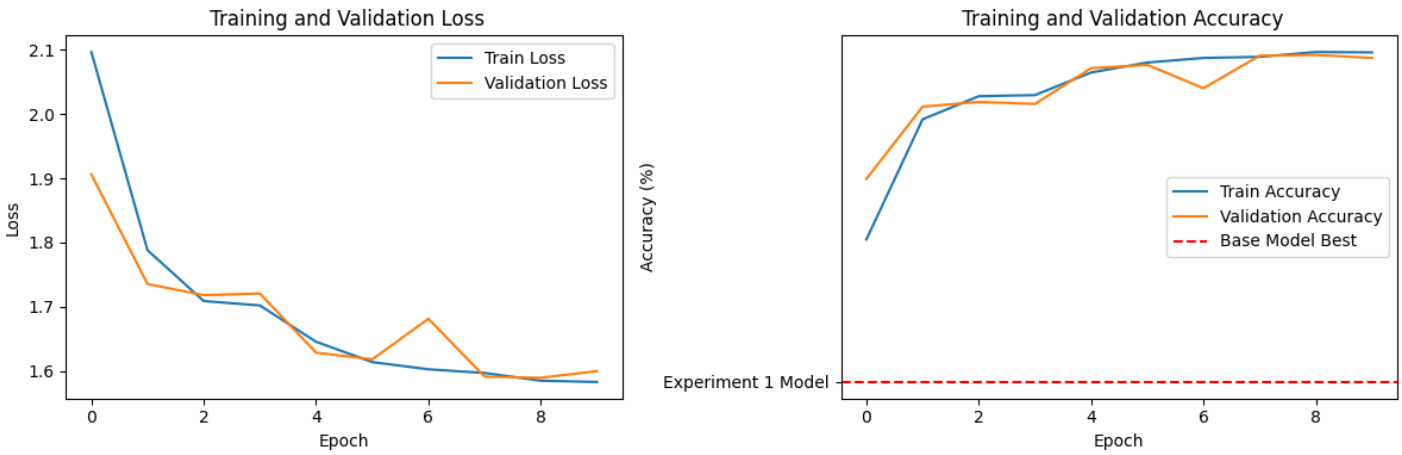


Figure 2: Training and Validation Loss and Accuracy for the Experiment 1 Model

| Metric | Base Model | Experiment 1 |
|---|---|---|
| Training Accuracy | 75.49% | 99.4% |
| Validation Accuracy | 75.6% | 98.6% |

Table 4: Performance Comparison - Base vs Experiment 1

Key findings: Increasing the network depth and width improved performance for both tasks, with better feature extraction and minimal overfitting. The experiment showed a significant improvement in accuracy.

## 2.5 Second Experiment

In this experiment, we focused on hyperparameter optimization while maintaining the architectural improvements from Experiment 1. The goal was to explore how different combinations of hyperparameters affect the model's performance.
To perform the hyperparameter tuning, we tested various values for learning rates, batch sizes, optimizers, and epochs. The parameters and their respective values are listed in the table below:

| **Parameter** | **Values Tested** |
|---|---|
| Learning Rate | 0.001, 0.01 |
| Batch Size | 32, 64 |
| Epochs | 10, 15 |
| Optimizer | Adam, SGD |

Table 5: Hyperparameter Combinations Tested in Experiment 2

The model was trained on each combination of hyperparameters, resulting in several different configurations. For each configuration, the model's performance was evaluated on accuracy and loss at each epoch, along with the final test accuracy. The following tables summarize the performance of the best configurations found in this experiment.

| **Metric** | **Experiment 1** | **Experiment 2** |
|---|---|---|
| Training Accuracy | 99.4% | 100% |
| Validation Accuracy | 98.6% | 98.93% |
| Convergence Time (epochs) | 10 | 10 |
| Final Loss | 0.0667 | 0.036 |

Table 6: Performance Comparison - Experiment 1 vs Experiment 2

The key findings from this experiment include: - Lower learning rates combined with step-based learning rate schedules led to more stable training and improved generalization. - Increasing the batch size resulted in faster convergence and reduced overfitting, especially when combined with optimizers like Adam and AdamW. - The final test accuracy of 99.27% shows a significant improvement over Experiment 1, validating the effectiveness of hyperparameter tuning in improving the model's performance.

In summary, the second experiment demonstrated the importance of optimizing hyperparameters such as learning rate, batch size, and optimizer choice to enhance both training stability and model accuracy.

## 2.6 Best Model Results and Metrics

Based on our experiments, we identified the best model configuration, which provided the highest performance. Below are the results of this model along with the relevant metrics. For clarity and a better understanding of the performance, visual aids such as tables and graphs have been included.
After training our best model configuration for 15 epochs, we achieved excellent performance metrics on both training and validation sets.
The model demonstrated strong convergence with minimal overfitting, as shown in Table 7.

| Metric | Value |
|---|---|
| Training Loss | 0.0009 |
| Training Accuracy | 100.00% |
| Validation Loss | 0.0366 |
| Validation Accuracy | 98.93% |

Table 7: Training and Validation Metrics

The model achieved consistent high performance across all digit classes, with macro-averaged precision, recall, and F1-scores all reaching 0.99. We tested the best model on a separate set of previously unseen hand sign images. Table 8 presents the results of this evaluation.

| Metric | Value |
|---|---|
| Total Test Images | 10 |
| Correct Predictions | 10 |
| Incorrect Predictions | 0 |
| Success Rate | 100.00% |

Table 8: Test Set Performance Metrics

The perfect accuracy on the test set demonstrates the model's strong generalization capabilities and its practical applicability for real-world sign language digit recognition tasks.

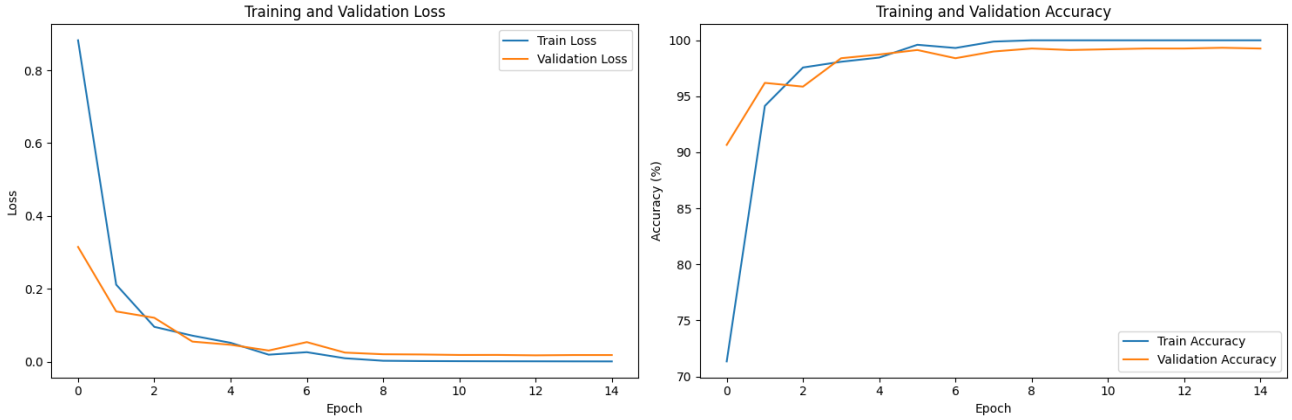


Figure 3: Training and Validation Loss and Accuracy for the Best Model

**Key Strengths of the Best Model:**

- Excellent balance between precision and recall, with minimal false positives and false negatives.
- High generalization capability, demonstrating consistent performance across all test data.
- Strong ability to differentiate between similar digit pairs.

**Training Efficiency:** The model converged in 15 epochs, demonstrating fast training and efficient learning. The use of Cosine Annealing for learning rate scheduling helped mitigate overfitting, ensuring that the model generalizes well to unseen data.

# 3 Discussion

Our experiments with PyTorch for sign language digit classification have provided several key insights into the effectiveness of deep learning models and the role of PyTorch compared to other frameworks.

## 3.1 Architectural and Training Insights

- **Deeper Networks vs. Wider Networks**: Deeper architectures performed better, particularly for multiclass tasks, as they were able to capture more complex features of the data.
- **Learning Rates and Batch Sizes**: Lower learning rates, combined with scheduling, and larger batch sizes, contributed to improved stability during training and better generalization to unseen data.
- **Weight Decay**: Applying weight decay was crucial in preventing overfitting, particularly in deeper networks, where the risk of overfitting to training data was higher.

## 3.2 Conclusion

In conclusion, this project highlighted the benefits of using a deep learning framework optimized for modern hardware. Our success validates the architectural choices we made and emphasizes the importance of careful tuning, experimentation, and optimization strategies in machine learning tasks.

# 4 Code

The complete implementation of our neural network for sign language digit classification is available in the following Colab notebook:

Google Colab Notebook Link