Convolutional Neural Network for Sign Language MNIST Classification

Do Thanh Dat[1]
Deep Learning Project, University of Science and Technology of Hanoi

Executive Summary

This report presents a detailed analysis of a custom-built Convolutional Neural Network (CNN) implementation for American Sign Language (ASL) alphabet classification using the Sign Language MNIST dataset. The implementation demonstrates a from-scratch approach to deep learning, featuring manual implementation of core neural network components including convolutional layers, pooling operations, dense layers, and backpropagation algorithms.

The model achieves reasonable performance on a limited dataset subset, showcasing the fundamental principles of deep learning without relying on established frameworks like TensorFlow or PyTorch. This implementation serves as an educational tool for understanding the underlying mechanics of CNN architectures and gradient-based optimization.

1 Dataset Overview and Preprocessing

The Sign Language MNIST dataset contains grayscale images of hand gestures representing letters A-Y of the American Sign Language alphabet (excluding J and Z due to their motion-based nature). Each image is 28×28 pixels, similar to the traditional MNIST digit dataset.

Dataset Characteristics:

- Image dimensions: 28×28 pixels
- Color space: Grayscale (single channel)
- Number of classes: 25 (letters A-Y, excluding J and Z)
- Pixel value range: Normalized to [0, 1] from original [0, 255]

Data Loading and Preprocessing: The following features: `read_sign_mnist_csv` function handles data ingestion with the following features:

- CSV parsing with header skipping

- Pixel value normalization (division by 255.0)
- Reshape operation to convert flat pixel arrays to 28×28 matrices
- Optional row limiting for memory management
- Label extraction and integer conversion

The preprocessing pipeline ensures data compatibility with the CNN architecture while maintaining computational efficiency. The normalization step is crucial for gradient stability during training, preventing vanishing or exploding gradients that could occur with raw pixel values.

2 Network Architecture Analysis

The implemented CNN follows a classic deep learning architecture pattern with alternating convolutional and pooling layers, followed by fully connected layers for classification.

2.1 Convolutional Layers

The network employs three convolutional layers with increasing filter counts:

Dataset Characteristics:

- Image dimensions: 28×28 pixels
- Color space: Grayscale (single channel)
- Number of classes: 25 (letters A-Y, excluding J and Z)
- Pixel value range: Normalized to [0, 1] from original [0, 255]

Layer 1 (Conv2D):

- Input channels: 1 (grayscale)
- Filters: 32
- Kernel size: 3×3
- Output size: 26×26×32

Layer 2 (Conv2D):

- Input channels: 32
- Filters: 128
- Kernel size: 3×3
- Output size: 11×11×128

Layer 3 (Conv2D):

- Input channels: 128
- Filters: 512
- Kernel size: 3×3
- Output size: 3×3×512

Each convolutional layer implements:

- Random weight initialization using uniform distribution [-0.05, 0.05]
- Bias terms for each filter
- Standard convolution operation without padding
- Linear activation (ReLU applied separately)

2.2 Pooling Operations

Three MaxPool2D layers with 2×2 pooling windows provide:

- Spatial dimension reduction
- Translation invariance
- Computational efficiency
- Feature hierarchy extraction

Layer	Input Features	Output Features	Purpose
Dense1	4608	4096	Feature extraction from flattened conv output
Dense2	4096	1024	Intermediate representation learning
Dense3	1024	256	High-level feature combination
Output	256	25	Classification logits for 25 classes

Table 1: Layer Specifications and Purposes

The progressive reduction in layer sizes creates a funnel architecture that gradually compresses the learned features into class-specific representations.

3 Pooling Operations

3.1 Forward Pass Implementation

The forward pass follows standard CNN principles:

- **Convolutional Operations:** Each Conv2D layer performs discrete convolution between input feature maps and learned kernels, computing dot products across spatial windows.
- **Activation Functions:** ReLU activation introduces non-linearity, enabling the network to learn complex patterns. The implementation handles both vector and tensor inputs recursively.
- **Pooling Operations:** Max pooling extracts the maximum value from each 2×2 spatial window, reducing spatial dimensions while preserving important features.
- **Flattening:** The flatten operation converts 3D feature maps to 1D vectors for dense layer processing.
- **Dense Computations:** Matrix multiplication between inputs and weight matrices, followed by bias addition.
- **Output Processing:** Softmax normalization converts final logits to probability distributions over the 25 classes.

3.2 Backpropagation and Training

The implementation includes manual backpropagation for dense layers: **Gradient Computation:**

- **Cross-entropy loss gradient:**

$$\boxed{\text{Gradient: pred}[i] - (1 \text{ if } i == \text{label else } 0)}$$

- **Weight updates:**

$$\boxed{w[i][j] -= \text{learning\_rate} \times \text{gradient} \times \text{input}[j]}$$

- **Bias updates:**

$$\boxed{b[i] \mathrel{-}= \text{learning\_rate} \times \text{gradient}}$$

Gradient Computation:

- Learning rate: 0.00005 (conservative to ensure stability)
- Batch size: 8 (small due to memory constraints)
- Epochs: 3 (limited training duration)
- Optimizer: Stochastic Gradient Descent (SGD)

3.3 Limitations and Constraints

The current implementation has several notable limitations:

1. Incomplete Backpropagation: Gradients are only computed for dense layers; convolutional layers use fixed, randomly initialized weights throughout training.

2. Limited Training Data: Only 30 training samples and 10 test samples are used, severely limiting the model's ability to learn generalizable features.

3. No Regularization: Absence of dropout, batch normalization, or weight decay may lead to overfitting.

4. Memory Constraints: The implementation prioritizes educational clarity over computational efficiency.

4 Performance Analysis and Results

4.1 Training Performance

With the limited dataset and incomplete backpropagation, the model's performance is constrained. The training process focuses primarily on optimizing the dense layers while convolutional features remain static.

Training Metrics:

- Loss function: Cross-entropy loss
- Accuracy measurement: Top-1 classification accuracy
- Convergence behavior: Limited by frozen convolutional weights

4.2 Evaluation Strategy

The evaluation process includes:

- **Forward pass through test samples**
- **Prediction generation using argmax on softmax outputs**
- **Accuracy calculation and per-sample analysis**
- **Visual inspection of correct and incorrect predictions**

4.3 Visualization and Analysis

The implementation provides comprehensive visualization capabilities:

- **Training sample visualization:** Displays original images with labels
- **Training metrics plotting:**Loss and accuracy curves over epochs
- **Test prediction analysis:**Separate visualization of correct and incorrect predictions
- **Per-sample prediction reporting:**Detailed output for each test case

5 Technical Strengths and Educational Value

5.1 Implementation Strengths

1. Educational Clarity: The code provides clear insight into CNN mechanics without framework abstractions
2. Modular Design: Each layer is implemented as a separate class with clear interfaces
3. Complete Pipeline: Includes data loading, preprocessing, training, and evaluation
4. Visualization Integration: Comprehensive plotting for analysis and debugging

5.2 Learning Outcomes

This implementation effectively demonstrates:

- Convolution operation mechanics
- Forward pass computation in deep networks
- Gradient computation and backpropagation principles
- Loss function implementation and optimization
- Data preprocessing and normalization importance

6 Recommendations for Enhancement

6.1 Immediate Improvements

- **Complete Backpropagation:** Implement gradient computation for convolutional layers to enable full network training
- **Increased Dataset Size:** Use the full Sign Language MNIST dataset for better generalization
- **Advanced Optimizers:** Implement Adam or RMSprop for improved convergence
- **Regularization Techniques:** Add dropout layers and weight decay to prevent overfitting

6.2 Advanced Enhancements

- **Batch Processing:** Implement true batch processing for improved efficiency
- **Data Augmentation:** Add rotation, scaling, and translation to increase dataset diversity generalization

- **Architecture Optimization:** Experiment with different layer configurations and activation functions
- **Performance Metrics:** Implement additional evaluation metrics like precision, recall, and F1-score

7 Conclusion

This CNN implementation is a valuable educational resource for grasping the fundamentals of deep learning. It highlights core concepts like convolutional operations, activation functions, and dense layer computations while offering hands-on experience with these principles. Despite constraints, such as incomplete backpropagation and a limited dataset, the implementation successfully bridges theoretical understanding and practical application, providing a strong foundation for further development in deep learning projects. Its modular architecture and clear design make it particularly useful for students exploring the mathematical operations behind modern AI systems.

With recommended enhancements, including a complete backpropagation implementation and scaling to larger datasets, this project could evolve into a fully functional CNN capable of delivering competitive performance on tasks like Sign Language MNIST classification. These improvements would not only maximize the architecture's potential but also deepen its educational value, demonstrating the complexities and power of deep learning in real-world applications.