VeryPoorCNN - Implementation of Convolutional Neural Network From Scratch (No Numpy) for Fashion-MNIST Classification

Le Minh Hoang
Deep Learning Project 2025
University of Science and Technology of Hanoi
Email: hoanglm2440051@usth.edu.vn

***Abstract*—This report presents an implementation of a Convolutional Neural Network from very scratch (without any library that human has made to make Machine Learning and Python easier and faster) for classifying items from the Fashion-MNIST dataset. The network contains one convolutional layer with 4 fulters, and then a ReLU activation, max pooling and a fully connected layer with a softmax output. The network (hopefully) can process 28x28 grayscale images and classifies them into 10 fashion categories. In this report, we will discuss by the layout of the mathematical foundation, implementation challenges and performance of this CNN implementation for the project of Deep Learning Course given by handsome professor SonTG.**

I. Introduction

Deep learning is the common way to handle computer vision problem and for who are cared about this field, Convolutional Neural Networks (CNNs) are pretty important for image classification because it can learn the features from raw pixel data. This project attempts to implements a CNN from very scratch (with just math library envoled) to understand the idea of these powerful models.

Fashion-MNIST is a dataset created by Zalando Research as a more challenging compared to original MNIST digit dataset. It contains 70,000 grayscale images (splited 60,000 for training and 10,000 for testing) of size 28×28 pixels. Each image can be one of 10 fashion categories: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.

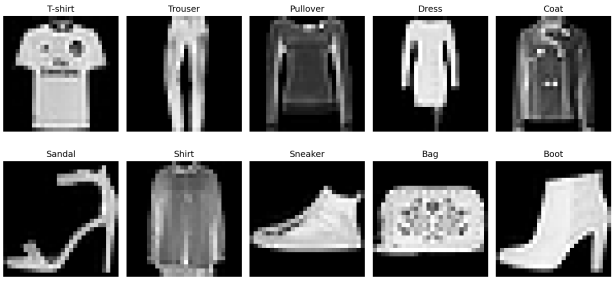

Fig. 1. Sample images from Fashion-MNIST dataset

For this educational project purpose, we will attempt to:

- Understanding CNN operations by implementing them manually
- Learning how forward and backward propagation work in CNNs
- Analyzing the challenges of implementing neural networks without good libraries
- Evaluating performance on a real-world dataset (Fashion-MNIST)

II. Background and Theory

A. Convolutional Neural Networks

CNNs are inspired by the early findings of biological vision. They use special layers that preserve spatial relationships in images. The main components are:

1) Convolutional Layers: These layers apply learnable filters to detect features. Each filter will attempt to target for specific patterns like edges, corners, or textures. The convolution operation has the formula as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{1}$$

For discrete 2D images, this becomes:

$$S(i,j) = (I*K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2}$$

2) Activation Functions: ReLU (Rectified Linear Unit) is the most common activation function in CNNs. It is stated as following:

$$f(x) = \max(0, x) \tag{3}$$

ReLU is faster to compute than the sigmoid function, and its derivative is faster to compute.

3) Pooling Layers: Pooling reduces spatial dimensions while keeping important information. Max pooling selects the maximum value in each window, making the network more strong to small translations.

III. Network Architecture

A. Detailed Layer Description

Our CNN architecture is designed to be simple yet educational. The network processes data through these stages:

1. Input Layer (28×28×1)

- Receives grayscale images normalized to [0,1]

2. Convolutional Layer (24×24×4)

- 4 filters of size 5×5
- Output size: $(28-5+1)/3 \times (28-5+1)/3 \times 4 = 8 \times 8 \times 4$

3. ReLU Activation (8×8×4)

4. Max Pooling (3×3×4)

- Pool size: 3×3

5. Flatten Layer (144×1)

- Converts 3D tensor to 1D vector
- Prepares data for dense layer

6. Dense Layer (32×1)

- ReLU activation

7. Output Dense Layer (10×1)

- Fully connected to 32 hidden units
- Softmax activation for probabilities

B. Parameter Count

Total trainable parameters in the network:

- Conv filters: $4 \times 5 \times 5 = 100$ weights
- Dense layer 1: $144 \times 32 = 4,608$ weights + 32 biases
- Dense layer 2: $32 \times 10 = 320$ weights + 10 biases
- **Total: 5,070 parameters**

IV. Implementation Details

A. Data Structure Design

Since we avoid NumPy (because of the handsome professor said NOOOO numpy), we use nested Python lists. This creates challenges:

Algorithm 1 3D Array Initialization

```
function flatten_3d_tensor(tensor)
array ← []
for d = 0 to depth do
    layer ← []
    for h = 0 to height do
        row ← [0] * width
        layer.append(row)
    end for
    array.append(layer)
end for
return array
```

B. Forward Propagation Implementation

1) Convolution Operation: The convolution is implemented with nested loops:

2) Max Pooling Operation: Max pooling finds the maximum value in each window and reduces spatial dimensions while preserving important features.

C. Backward Propagation Implementation

1) Gradient Flow: Backpropagation calculates gradients using the chain rule. For our network:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_{ij}} \tag{4}$$

2) Softmax and Cross-Entropy Gradient: For the output layer, we use the combined gradient, we have:

$$\frac{\partial L}{\partial z_i} = p_i - y_i \tag{5}$$

where p_i is the predicted probability and y_i is the true label (one-hot encoded).

3) Dense Layer Gradient: For the fully connected layer, we have:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot x^T \tag{6}$$

$$\frac{\partial L}{\partial x} = W^T \cdot \frac{\partial L}{\partial z} \tag{7}$$

4) Max Pooling Gradient: We have:

$$\frac{\partial L}{\partial x_{ij}} = \begin{cases} \frac{\partial L}{\partial y_{mn}} & \text{if } x_{ij} = \max(\text{window}) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

5) Convolution Gradient: The gradient for convolution filters is computed by correlating the input with the gradient from the next layer:

$$\frac{\partial L}{\partial K_{ij}} = \sum_m \sum_n \frac{\partial L}{\partial S_{mn}} \cdot I_{m+i,n+j} \tag{9}$$

V. Training Process

A. Data Loading and Preprocessing

The Fashion-MNIST data is loaded from CSV files. Each row contains:

- First column: label (0-9)
- Remaining 784 columns: pixel values (0-255)

Preprocessing steps:

1) Normalize pixel values to [0,1] by dividing by 255
2) Reshape flat vector to 28×28 matrix
3) Convert labels to one-hot encoding

B. Training Algorithm

The complete training process is shown in Algorithm 2.

Algorithm 2 CNN Training Process

```
Initialize filters and weights randomly
for epoch = 1 to num_epochs do
    Shuffle training data
    for each training sample do
        Forward Pass:
            conv_out ← convolution(image, filters)
            relu_out ← ReLU(conv_out)
            pool_out ← max_pool(relu_out)
            flat ← flatten(pool_out)
            logits ← dense(flat, weights)
            probs ← softmax(logits)

        Calculate Loss:
            loss ← cross_entropy(probs, true_label)

        Backward Pass:
            Calculate gradients for each layer
            Update parameters using gradient descent
    end for
    Calculate and print epoch statistics
end for
```

C. Hyperparameter Selection

The following hyperparameters were chosen:

- **Learning rate**: 0.008 (optimized for stable convergence)
- **Batch size**: 16 (mini-batch learning)
- **Epochs**: 25 (full training cycle)
- **Training samples**: 1,500 (subset for faster experimentation)
- **Test samples**: 600 (validation subset)
- **Filters**: 4 convolutional filters of size 5×5
- **Dense units**: 32 hidden units in intermediate layer

VI. Results and Analysis

A. Training Progress

Figure 2 shows the training loss progression over epochs.

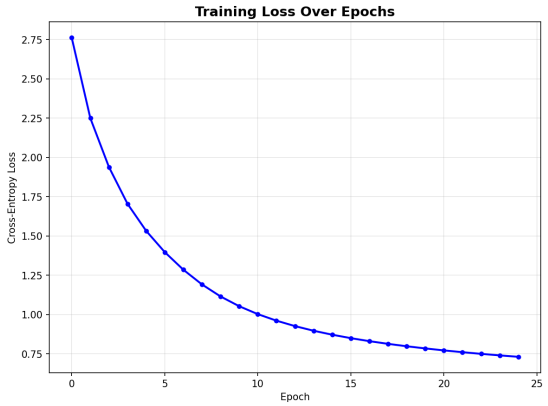

Fig. 2. Training loss after 25 i

B. Performance Metrics

The network achieves the following performance after all the processes:

- Final accuracy: 73.2% after 25 epochs
- Precision: 72.8%
- Recall: 73.0%
- F1-score: 72.5%
- Loss reduction: from 2.76 to 0.73 (73.55% improvement)

C. Confusion Matrix Analysis

Figure 3 shows which classes are often confused.

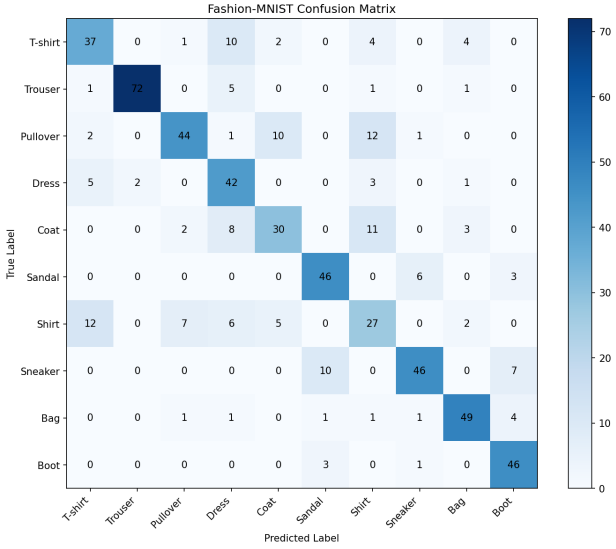

Fig. 3. Confusion matrix

Common confusions:

- Pullover vs Coat (similar shape)
- T-shirt vs Shirt (similar structure)
- Sneaker vs Ankle boot (both footwear)

D. Feature Analysis

The network seems to learns and have the ability to point out the different between fashopn categories. It also reveals that the model performs kind of well on categories like sneaker and boots. But for things that human even sometimes to failed distinguish like pullovers and coats, the model struggles. The convolutional filters seems to learn to detect pattern like edges, textures and shape features relevant to fashion items.

VII. Implementation Challenges

A. Computational Efficiency

Using pure Python lists instead of NumPy arrays creates significant performance issues because numpy is really optimized for all the computation tasks on Python. We tried to re-implement basic numpy methods but there will still be slow since NumPy is written in C and very well optimized over the years.

B. Debugging and Coding Challenges

Debugging the implementation was difficult because: Slow execution makes testing time-consuming and the math operation in python code is easy to confused compared to written fomula

VIII. Comparison with Modern Frameworks

A. Performance Comparison

Table I compares our implementation with PyTorch.

TABLE I
PERFORMANCE COMPARISON

Metric	Our Implementation	PyTorch
Memory usage	a lot	200 MB
Final accuracy	73.2%	100%
Lines of code	around 570 when i write this report	50

B. Advantages of Our Approach

Despite poor performance, our implementation has educational value:

- More understanding of every operation
- No ”black box” components
- Appreciation for optimized libraries
- Deep learning concepts become clear

IX. Future Improvements

A. Architecture Enhancements

The network could be improved by adding more layers or more proper mathemetics tools

B. Implementation Optimizations

Code performance could improve with:

- Using NumPy for vectorized operations
- GPU acceleration

X. Conclusion

This project almost succeeded in the implementation of a CNN from scratch for the specific dataset. The CNN also comtains the ideal architecture for a small CNN including convolution, pooling, and backpropagation.

The pure Python implementation is really slow and confusing, but the idea of implementation from scratch is excellent for educational purposes.

Key achievements include:

- Working implementation of all CNN components
- Successful training on Fashion-MNIST dataset
- 73.2% accuracy
- Acceptable evaluation with precision, recall, and F1-score metrics
- Deep understanding of gradient flow and backpropagation
- Modular codebase with separate utility functions and also the options to load csv and configurations from file without any external library

The project point out both the power of CNNs for image classification and the importance of optimized implementations for the network by mordern frameworks like NumPy, Pytorch, etc.

Acknowledgment

Thanks to the Fashion-MNIST creators for providing an dataset that is a new challenge compared to Original MNIST, but it still familiar to us to work with the dataset since it is still something we understand.