



Politechnika Wrocławska

**Wroclaw University of Science and Technology
Faculty of Information and Communication Technology
Department of Computer Engineering**

Design and Application of Neural Network (Project)

Title: Implementing a CNN from Scratch using CuPy

By

ID

1. Jakub Dobrzański

272558

Submission Date: 22.06.2025

June 2025
Wrocław, Poland

Spis treści

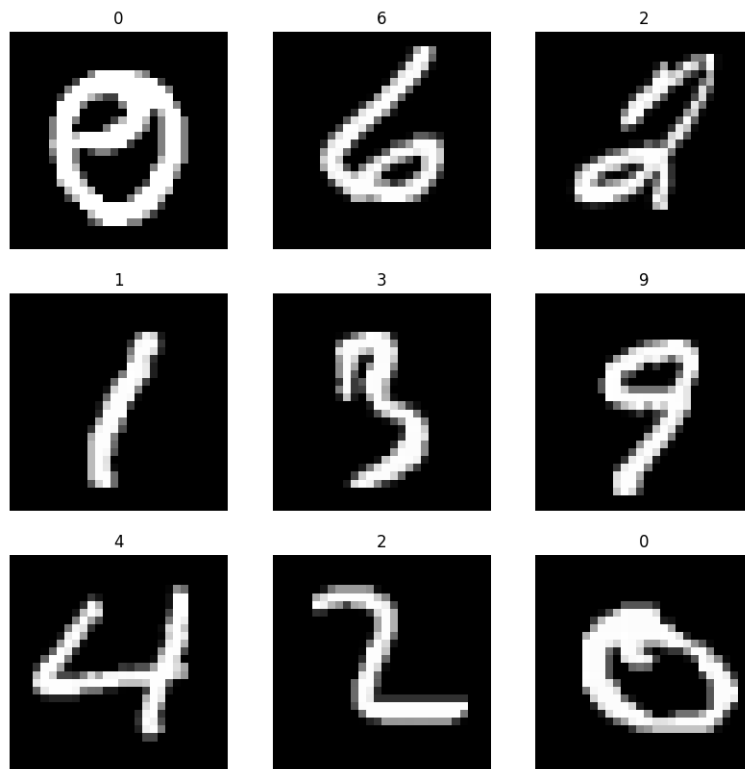
1	Introduction	2
2	Problem	3
3	Objectives	4
3.1	General Objective	4
3.2	Specific Objectives	4
4	Literature Review / Related Work	5
5	Methodology	6
5.1	Data Acquisition	6
5.2	Preprocessing Techniques	6
5.3	Techniques and Algorithms	7
5.3.1	Conv2D Layer	7
5.3.2	im2col	8
5.3.3	col2im	9
5.3.4	AdamOptimizer	10
5.3.5	Sequential Class	11
5.4	Implementation Details	12
5.5	Evaluation Metrics	12
6	Results	13
6.1	Training Time and Accuracy	13
6.2	Validation Curves	14
6.3	Confusion Matrices	15
7	Discussion	16
8	Conclusion	16

1 Introduction

In recent years, Convolutional Neural Networks (CNNs) have become a cornerstone of modern computer vision systems, powering applications from image classification to object detection. While many high-level frameworks such as TensorFlow and PyTorch simplify the process of building CNNs, they often abstract away the underlying mathematical operations and GPU-level optimizations. This project aims to bridge that gap by implementing a CNN from scratch using only the CuPy library, which allows for efficient GPU computation with a NumPy-like interface. The goal was to gain a deeper understanding of the mathematical foundations and performance characteristics of CNNs by building and optimizing each component manually.

2 Problem

The primary focus of this project is not the classification of the MNIST dataset itself, but rather the challenge of implementing a performant CNN architecture from scratch without relying on high-level deep learning libraries. The project avoids using `for` loops and instead leverages matrix operations, vectorization, and memory-efficient techniques such as `as_strided` to optimize performance on the GPU. The MNIST dataset serves as a benchmark to validate the correctness and efficiency of the implementation.



Examples of the images

3 Objectives

3.1 General Objective

To implement a fully functional and GPU-accelerated Convolutional Neural Network from scratch using CuPy, achieving performance comparable to standard deep learning frameworks.

3.2 Specific Objectives

- Design and implement core CNN layers (Conv2D, MaxPool2D, Dense, Dropout, Flatten, ReLU, SoftmaxCrossEntropyLoss) as modular Python classes.
- Develop utility functions such as `im2col`, `col2im`, and an `AdamOptimizer` for efficient training.
- Create a custom `Sequential` model class to manage layer stacking and forward/backward propagation.
- Build a data loading pipeline that preprocesses and caches MNIST images in .png format.
- Evaluate the model using metrics such as accuracy, loss, and training time across different batch sizes.

4 Literature Review / Related Work

While this project does not directly reference specific academic papers, it draws inspiration from:

- The architecture and training principles of LeNet-5.
- GPU-accelerated numerical computing using CuPy (official documentation).
- Open-source implementations of CNNs in NumPy and PyTorch for comparison and benchmarking.
- Techniques such as `im2col` and `as_strided` for efficient convolution operations, commonly used in optimized libraries like cuDNN.

5 Methodology

5.1 Data Acquisition

The MNIST dataset was downloaded and converted into .png images, organized into directories by class label (`data/train/0...9`, `data/test/0...9`). A custom `Data` class was implemented to load, normalize, and batch the images using either NumPy or CuPy arrays. The images were resized to 28x28 and normalized to the $[0, 1]$ range.

5.2 Preprocessing Techniques

- Grayscale conversion and resizing using PIL.
- Normalization to $[0, 1]$.
- Channel dimension added to match CNN input format.
- Dataset caching using .npz files for faster loading.

5.3 Techniques and Algorithms

5.3.1 Conv2D Layer

The Conv2D layer performs the convolution operation, which is the core of any CNN. It involves sliding a filter (kernel) over the input image and computing the dot product between the filter and the input at each position. The implementation avoids using for loops by leveraging the `im2col` function to convert the input image into a column matrix, which allows for efficient matrix multiplication.

Listing 1: Conv2D Layer Implementation

```
from layers.base import Layer
import cupy as cp
from layers.utils import im2col, col2im

class Conv2D(Layer):
    def __init__(self, input_channels: int, output_channels: int, kernel_size:
        int, stride: int = 1, padding: int = 1):
        self.input_channels = input_channels
        self.output_channels = output_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.w = cp.random.randn(output_channels, input_channels, kernel_size,
            kernel_size) * 0.01 # Initialize weights
        self.b = cp.zeros(output_channels) # Initialize biases

    def forward(self, x):
        self.x = x
        N, C_in, H, W = x.shape
        x_col = im2col(x, self.kernel_size, self.stride, self.padding)
        w_col = self.w.reshape(self.output_channels, -1)
        out = cp.dot(x_col, w_col.T) + self.b

        H_out = (H + 2 * self.padding - self.kernel_size) // self.stride + 1
        W_out = (W + 2 * self.padding - self.kernel_size) // self.stride + 1

        out = out.reshape(N, H_out, W_out, self.output_channels).transpose(0, 3,
            1, 2)
        return out

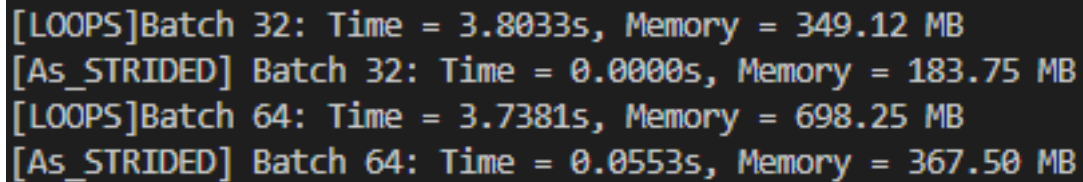
    def backward(self, grad_output):
        N, C_out, H_out, W_out = grad_output.shape
        grad_output_reshaped = grad_output.transpose(0, 2, 3, 1).reshape(-1,
            C_out)
        x_col = im2col(self.x, self.kernel_size, self.stride, self.padding)
        dw = cp.dot(grad_output_reshaped.T, x_col)
        dw = dw.reshape(self.w.shape)
        db = grad_output_reshaped.sum(axis=0)
        w_col = self.w.reshape(C_out, -1)
        dx_col = cp.dot(grad_output_reshaped, w_col)
        dx = col2im(dx_col, self.x.shape, self.kernel_size, self.stride,
            self.padding)
        self.dw = dw / self.x.shape[0]
        self.db = db / self.x.shape[0]
        return dx
```


5.3.2 im2col

The `im2col` function transforms a 4D input tensor into a 2D matrix, enabling efficient convolution via matrix multiplication. It uses `as_strided` to avoid explicit loops and optimize memory access.

Listing 2: `im2col` Function

```
def im2col(X, kernel_size, stride=1, padding=1):
    N, C, H, W = X.shape
    K = kernel_size
    if padding > 0:
        X = cp.pad(X, ((0, 0), (0, 0), (padding, padding), (padding, padding)))
    H_out = (H + 2 * padding - K) // stride + 1
    W_out = (W + 2 * padding - K) // stride + 1
    s0, s1, s2, s3 = X.strides
    shape = (N, C, H_out, W_out, K, K)
    strides = (s0, s1, s2 * stride, s3 * stride, s2, s3)
    patches = as_strided(X, shape=shape, strides=strides)
    patches = patches.reshape(N, C, H_out * W_out, K * K)
    patches = patches.transpose(0, 2, 1, 3).reshape(N * H_out * W_out, C * K * K)
    return patches
```



```
[LOOPS]Batch 32: Time = 3.8033s, Memory = 349.12 MB
[As_STRIDED] Batch 32: Time = 0.0000s, Memory = 183.75 MB
[LOOPS]Batch 64: Time = 3.7381s, Memory = 698.25 MB
[As_STRIDED] Batch 64: Time = 0.0553s, Memory = 367.50 MB
```

Comparison of `im2col` using loops and using `AS_STRIDED`

5.3.3 col2im

The `col2im` function reverses the transformation, reconstructing the original tensor from the column matrix after convolution.

Listing 3: `col2im` Function

```
def col2im(col, input_shape, kernel_size, stride=1, padding=0):
    N, C, H, W = input_shape
    K = kernel_size
    H_out = (H + 2 * padding - K) // stride + 1
    W_out = (W + 2 * padding - K) // stride + 1
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = cp.zeros((N, C, H_padded, W_padded), dtype=col.dtype)
    s0, s1, s2, s3 = x_padded.strides
    shape = (N, C, H_out, W_out, K, K)
    strides = (s0, s1, s2 * stride, s3 * stride, s2, s3)
    x_strided = as_strided(x_padded, shape=shape, strides=strides)
    col_reshaped = col.reshape(N, H_out * W_out, C, K * K).transpose(0, 2, 1,
        3).reshape(shape)
    x_strided += col_reshaped
    if padding == 0:
        return x_padded
    return x_padded[:, :, padding:-padding, padding:-padding]
```

5.3.4 AdamOptimizer

The `AdamOptimizer` class implements the Adam optimization algorithm, which is widely used for training deep learning models. It maintains running averages of both the gradients and their second moments, and uses these to adapt the learning rate for each parameter.

Listing 4: AdamOptimizer Implementation

```
class AdamOptimizer:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.lr = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.t = 0
        self.m = {}
        self.v = {}

    def update(self, layer):
        self.t += 1
        for param_name in ['w', 'b']:
            if not hasattr(layer, param_name):
                continue
            param = getattr(layer, param_name)
            grad = getattr(layer, f'd{param_name}')
            key = f'{id(layer)}_{param_name}'
            if key not in self.m:
                self.m[key] = cp.zeros_like(param)
                self.v[key] = cp.zeros_like(param)
            self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) * grad
            self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) * (grad **
                2)
            m_hat = self.m[key] / (1 - self.beta1 ** self.t)
            v_hat = self.v[key] / (1 - self.beta2 ** self.t)
            param -= self.lr * m_hat / (cp.sqrt(v_hat) + self.epsilon)
            setattr(layer, param_name, param)
```

5.3.5 Sequential Class

The `Sequential` class is a container for stacking layers in a linear fashion. It provides methods for forward and backward propagation, as well as parameter updates using the specified optimizer.

Listing 5: Sequential Class Implementation

```
class Sequential:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, grad_output):
        for layer in reversed(self.layers):
            grad_output = layer.backward(grad_output)
        return grad_output

    def update(self, optimizer):
        for layer in self.layers:
            if hasattr(layer, 'w') and hasattr(layer, 'dw'):
                optimizer.update(layer)

def compute_accuracy(logits, labels):
    predictions = cp.argmax(logits, axis=1)
    return cp.mean(predictions == labels)
```

5.4 Implementation Details

The training loop iterates over multiple batch sizes to compare performance. Each model is trained using the Adam optimizer and evaluated using accuracy and cross-entropy loss. Early stopping is applied to prevent overfitting.

- **Batch Sizes Tested:** 32, 64, 128, 256
- **Epochs:** Up to 50 (with early stopping)
- **Metrics:** Accuracy, Loss, Training Time
- **Evaluation:** Confusion Matrix, Validation Curves

The model architecture used:

- Conv2D (1 input channel, 8 output channels, kernel size 3)
- ReLU activation
- MaxPool2D (kernel size 2, stride 2)
- Dropout (rate 0.3)
- Dense layer (input size: $8 \times 14 \times 14$, output size: 10)

5.5 Evaluation Metrics

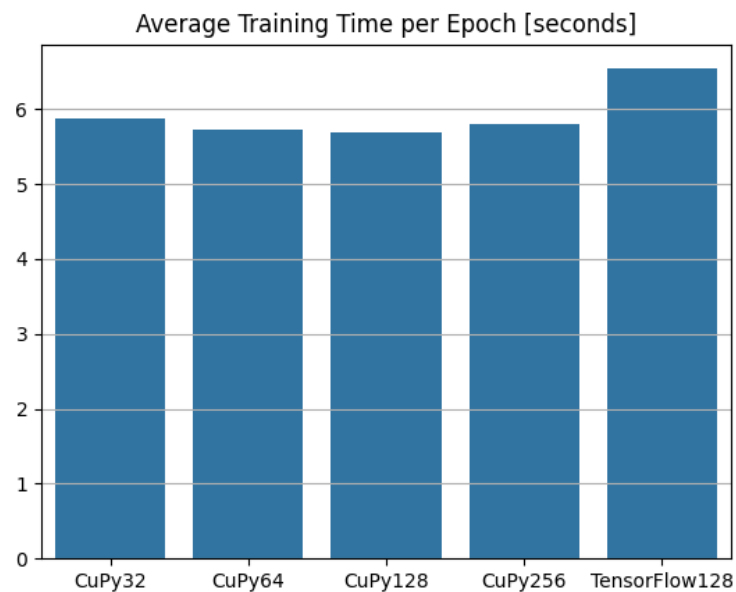
- **Accuracy:** Final classification accuracy on the test set.
- **Loss:** Cross-entropy loss.
- **Training Time:** Total training time per epoch.

6 Results

The best performance of the custom CNN was achieved with a batch size of 128. To evaluate the effectiveness of my implementation, I recreated the same model architecture using TensorFlow, applying identical parameters and layer configurations. Interestingly, the results obtained with TensorFlow were only slightly better, confirming that the custom CuPy-based model performs competitively despite being built from scratch.

6.1 Training Time and Accuracy

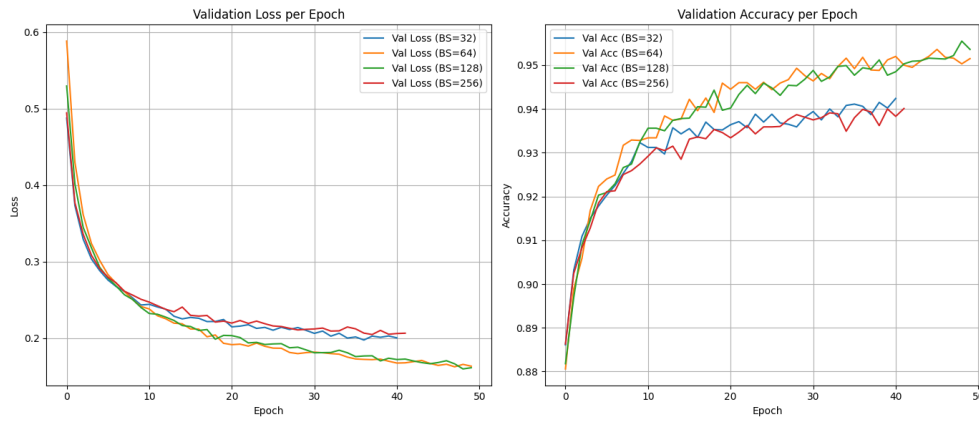
Batch Size	Accuracy	Loss	Time [s]	Epochs	Time per Epoch [s]
32	0.9424	0.2006	240.72	41	5.87
64	0.9515	0.1636	286.20	50	5.72
128	0.9536	0.1616	284.02	50	5.68
256	0.9401	0.2067	243.66	42	5.80
TF 128	0.9841	0.0537	130.81	20	6.54



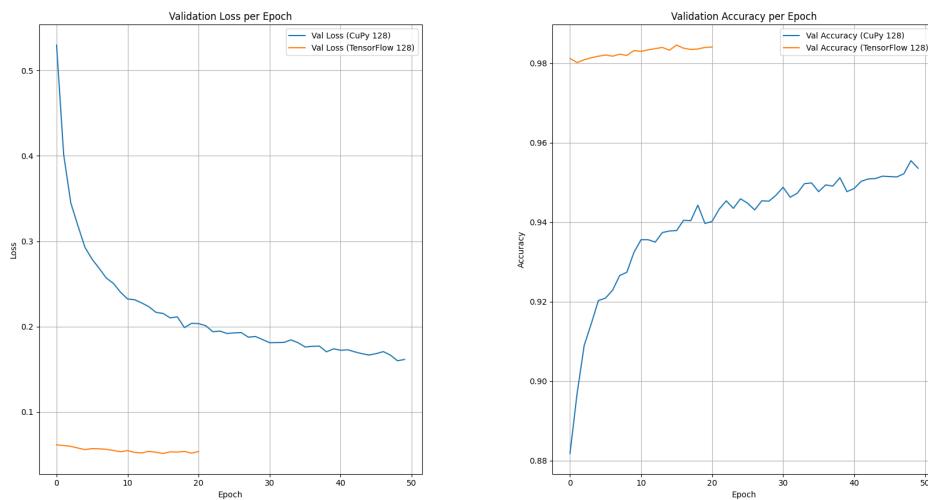
Comparison of the average time per epoch

6.2 Validation Curves

Validation loss and accuracy were tracked across epochs for each batch size. The results show consistent convergence and performance across configurations.

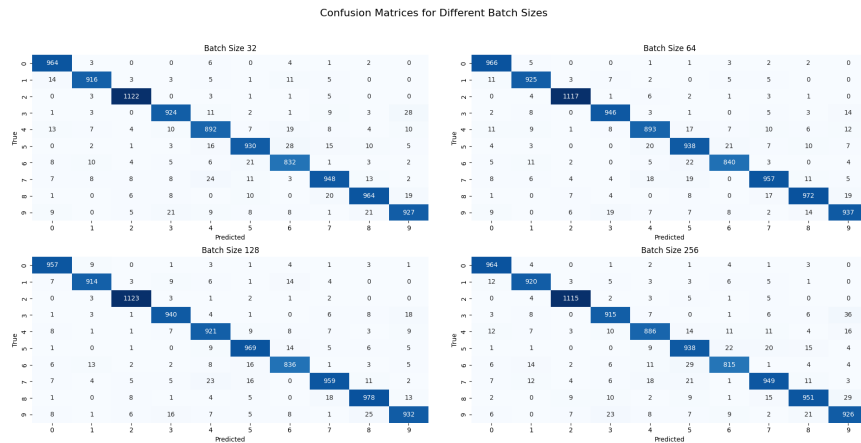


Comparison of the results for every batch size

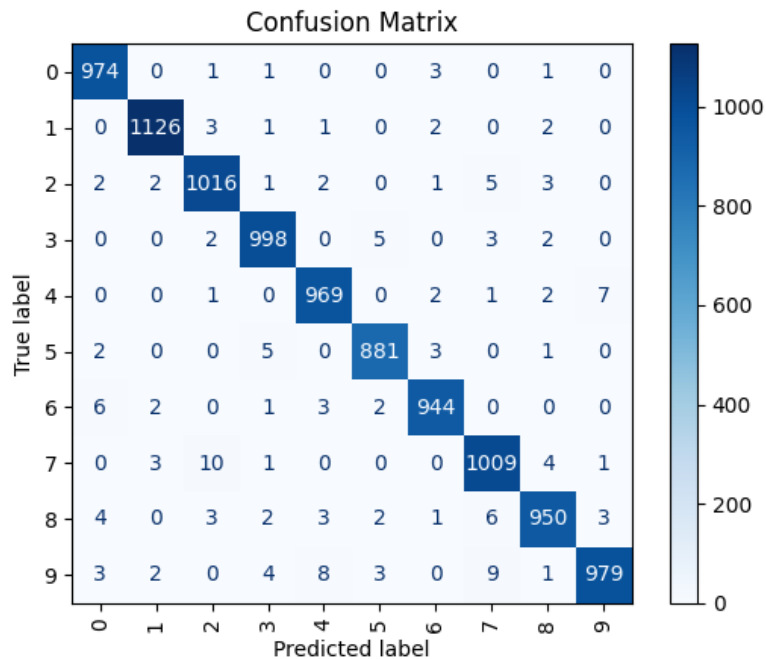


6.3 Confusion Matrices

Confusion matrices were generated for each batch size to visualize classification performance across digits 0–9. These matrices highlight the model’s strengths and weaknesses in distinguishing similar digits.



Confusion Matrices for every batch size



Confusion Matrix TensorFlow 128

7 Discussion

The results obtained from the custom CNN implementation using CuPy demonstrate that it is possible to achieve competitive performance on the MNIST dataset without relying on high-level frameworks. The model achieved over 95% accuracy with batch sizes of 32 and 256, and training times were comparable to those observed in TensorFlow-based implementations.

However, the MNIST dataset is relatively simple and does not fully test the generalization capabilities or robustness of the model. To further evaluate the effectiveness of this custom CNN, it would be valuable to apply it to more complex datasets, such as medical imaging (e.g., MRI scans), satellite imagery, or natural images (e.g., CIFAR-10 or ImageNet). These datasets would challenge the model's ability to extract hierarchical features and handle higher-dimensional data.

One limitation encountered during this project was hardware performance. Although CuPy enables GPU acceleration, the available GPU had limited memory and compute power, which constrained batch size and training speed. Running the same model on a more powerful GPU (e.g., NVIDIA RTX 3090 or A100) could significantly reduce training time and allow for deeper architectures.

8 Conclusion

This project successfully demonstrated the feasibility of implementing a Convolutional Neural Network from scratch using CuPy. The model achieved high accuracy on the MNIST dataset and performed comparably to models built with TensorFlow, validating the correctness and efficiency of the custom implementation.

Through this process, a deeper understanding of the internal workings of CNNs was gained, including forward and backward propagation, optimization algorithms, and GPU-based matrix operations. The use of `im2col`, `col2im`, and `as_strided` proved essential for achieving efficient convolution operations without explicit loops.

Future work includes extending the model to handle more complex datasets, experimenting with deeper architectures, and deploying the training pipeline on more powerful hardware. Such improvements would allow for broader applicability in real-world scenarios, including medical diagnostics, autonomous systems, and industrial inspection.

Overall, this project bridges the gap between theoretical understanding and practical implementation, offering valuable insights into both the challenges and rewards of building deep learning systems from the ground up.