Design and Application of Neural Network (Project)

Title: Implementing a CNN from Scratch using CuPy

By
Jakub Dobrzański

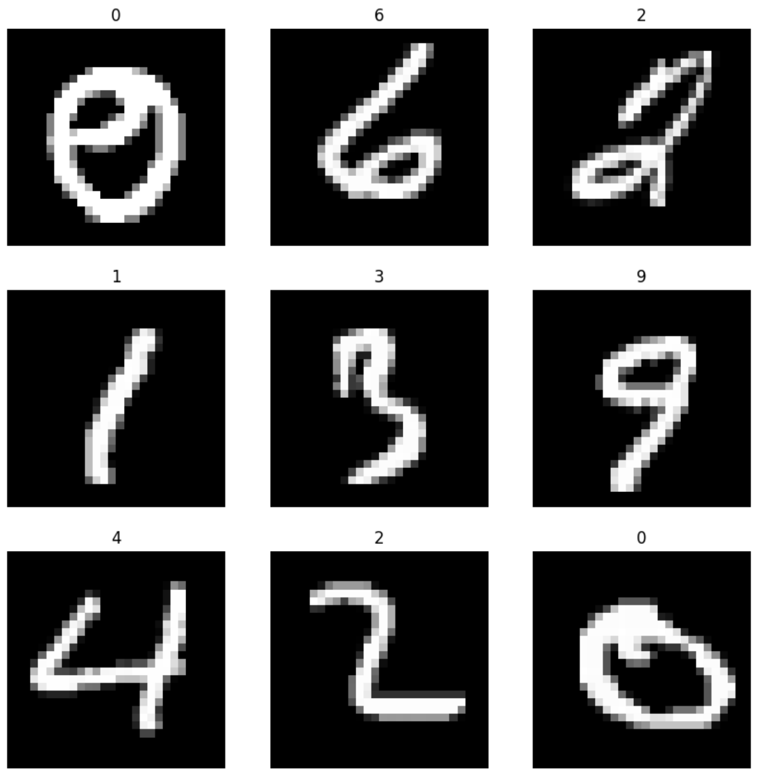
Contents

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 Methodology

4.1 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.

4.2 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.

4.3 Techniques and Algorithms

4.3.1 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.

4.3.2 im2col

The `im2col` function transforms a 4D input tensor into a 2D matrix, enabling efficient convolution via matrix multiplication.

4.3.3 col2im

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

4.3.4 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.

4.3.5 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.

4.4 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)

4.5 Evaluation Metrics

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

5 Results

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.

5.1 Validation Curves

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

5.2 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.

6 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.

7 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.