Performance Study of a Convolutional Layer in MNIST CNN

Manisha Goyal
Email: manisha.goyal@nyu.edu

***Abstract*—This report presents a performance study of a convolutional layer in a Convolutional Neural Network (CNN) trained on the MNIST dataset, specifically focusing on the Conv2d-2 layer. By estimating the layer's time and space complexity using theoretical calculations, and then comparing these estimates with empirical measurements in PyTorch on an NVIDIA V100 GPU, this study explores computational demands across various batch sizes. Key performance characteristics, such as floating point operations (FLOPs) and memory usage, are examined to understand the linear growth pattern that emerges with increasing batch sizes. This pattern underscores the balance between computational load and memory capacity in CNNs, providing practical insights for batch size selection and resource allocation in large-scale and memory-constrained environments. The findings demonstrate the impact of batch size on GPU performance and offer strategies to optimize network architecture for efficient deployment.**

I. Introduction

Convolutional Neural Networks (CNNs) have become fundamental to a wide range of computer vision tasks due to their ability to automatically learn hierarchical features from data. One popular application of CNNs is digit recognition using the MNIST dataset, a benchmark dataset of handwritten digits. Understanding the computational complexity of CNN layers is essential for optimizing model performance, especially when dealing with large-scale datasets or deploying models in resource-constrained environments. The space and time complexity of a convolutional layer is determined by factors like kernel size, input and output dimensions, and batch size.

This report focuses on analyzing the performance of the second convolutional layer (Conv2d-2) in a CNN trained on the MNIST dataset. The Conv2d-2 layer is of particular interest because, as the second layer in the network, it processes intermediate feature maps produced by the first convolutional layer (Conv2d-1). This intermediate processing often requires significantly more computation due to the increased number of channels and feature map dimensions. Therefore, optimizing layers such as Conv2d-2 can have a substantial impact on the overall performance of CNNs, especially when scaling to larger datasets or higher-resolution images.

To better understand these performance factors, this report focuses on:

- Estimating the time and space complexity of the Conv2d-2 layer using a pen-and-paper method.
- Measuring the actual performance of the layer in terms of floating point operations (FLOPs) and memory usage on a GPU using PyTorch.
- Exploring the impact of varying batch sizes on performance, and comparing the estimated and measured results, discussing any observed differences.

This study provides insights into the performance characteristics of convolutional layers in practical CNN applications and offers guidelines for optimizing such layers in future work.

II. Methodology

A. Pen and Paper Method

To estimate the time and space complexity of the Conv2d-2 layer, a theoretical approach was used.

1) Time Complexity: The time complexity is determined by the number of operations required to process the input for each batch. For the Conv2d-2 layer, the time complexity is given by:

$$O(B \times 2 \times K_h \times K_w \times C_{in} \times H_{out} \times W_{out} \times C_{out})$$

Where:

- B is the batch size, as the operation is performed for each input in the batch.
- K_h and K_w are the kernel height and width.
- C_{in} and C_{out} are the number of input and output channels.
- H_{out} and W_{out} are the height and width of the output feature map.

2) Space Complexity: The space complexity can be broken down into two components:

- **Parameters**: These contribute to the model's static memory usage. Parameters represent the learned weights and biases of the model and are independent of the batch size. The number of parameters in the Conv2d-2 layer is:

$$\text{Parameters} = (K_h \times K_w \times C_{in} \times C_{out}) + C_{out}$$

- **Activations**: These contribute to the dynamic memory usage. Activations represent the intermediate feature maps generated after each layer, and their size is proportional to the batch size. The space required for activations is given by the size of the output feature map multiplied by the batch size B:

$$\text{Activations} = H_{out} \times W_{out} \times C_{out} \times B$$

Thus, the total space complexity can be calculated as the sum of the parameters and activations.

B. Experimental Setup

The empirical study was conducted using PyTorch on the MNIST dataset, where the Conv2d-2 layer was analyzed in terms of time and space complexity. The PyTorch MNIST example code, available at https://github.com/pytorch/examples/tree/master/mnist, was used as the basis for this study.

- **Hardware**: As shown in Figure 1, the experiments were conducted using GPU acceleration on the NYU Greene cluster, utilizing NVIDIA V100 GPUs. These GPUs provided high computational power, which was essential for running the deep learning models and profiling GPU kernels.

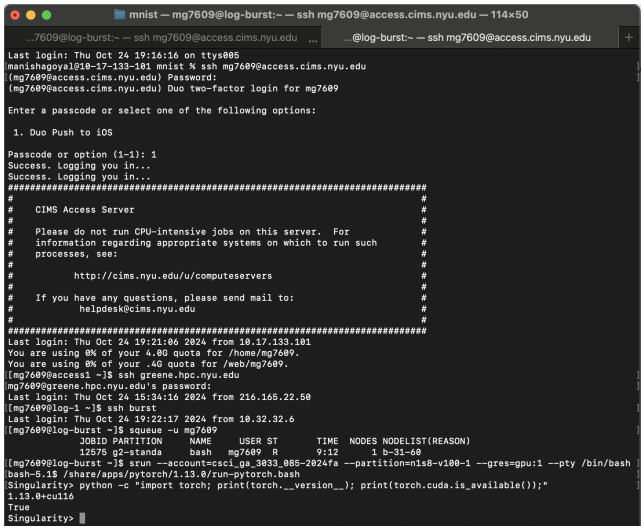

Fig. 1. Logging into the NYU Greene HPC cluster

- **Software**: PyTorch (version 1.13.0 with CUDA 11.6) was used for implementing and testing the Conv2d-2 layer. A custom virtual environment was set up for the experiment, and the `torchvision` library was used for data preprocessing and augmentation.
- **Batch Sizes**: The impact of varying batch sizes (16, 32, 64, 128, 256) was measured to understand how they influenced memory usage and processing speed.
- **Model Overview**: The `TorchSummary` utility was used to provide an overview of the model's architecture, which facilitated the verification of the theoretical estimates for time and space complexity.
- **Time Profiling**: NVIDIA Nsight Compute was used to profile the execution of CUDA kernels during training. Specific metrics related to FLOPs were collected to analyze the time complexity of the Conv2d-2 layer. The profiling command utilized the metrics `smsp__sass_thread_inst_executed_op_fadd_pred_on.sum`, `smsp__sass_thread_inst_executed_op_ffma_pred_on.sum`, `smsp__sass_thread_inst_executed_op_fmul_pred_on.sum`. These capture the number of addition, fused multiply-add, and multiplication operations, respectively.
- **Memory Profiling**: Memory usage was measured before and after the Conv2d-2 layer using PyTorch's CUDA memory utilities. The memory metrics `torch.cuda.memory_allocated()` and `torch.cuda.memory_reserved()` were used to determine the dynamic memory allocated for activations and reserved GPU memory during each pass.

C. Metrics and Evaluation

To measure the time complexity, the number of floating-point operations (FLOPs) required for the Conv2d-2 layer was recorded. For space complexity, memory usage was tracked based on the size of the parameters and activations. Both time and space complexity were evaluated as a function of the batch size.

The results were compared to the theoretical estimations obtained from the pen-and-paper method, and the discrepancies were analyzed to identify any differences due to hardware acceleration or implementation optimizations.

III. Results and Analysis

A. Theoretical Estimations

Using the pen-and-paper method, the theoretical time and space complexity of the Conv2d-2 layer were calculated. The parameters for the model were verified using the `torchsummary` utility as shown in Figure 2.

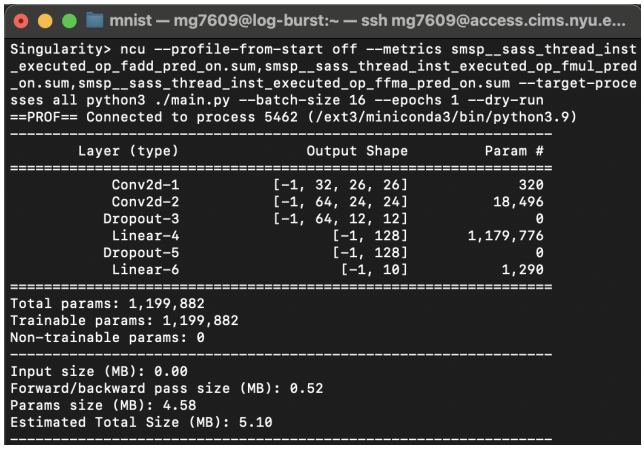

Fig. 2. Summary of model parameters and size

1) Time Complexity: The time complexity is proportional to the number of operations performed by the convolutional layer. For the Conv2d-2 layer (`nn.Conv2d (32,64,3,1)`) with the following parameters:

- Kernel size: $K_h = 3$, $K_w = 3$
- No. of input and output channels: $C_{in} = 32$, $C_{out} = 64$
- Stride: $S = 1$
- Padding: $P = 0$ (assume no padding as not defined)
- Batch Size: B

The FLOPs for a convolution operation is given by:

$$\text{FLOPs} = B \times (K_h \times K_w \times C_{in} \times H_{out} \times W_{out} \times C_{out}) \times 2$$

Where H_{out} and W_{out} represent the height and width of the output feature map, respectively, and are computed as:

$$H_{out} = W_{out} = \frac{H_{in} - K_h + 2P}{S} + 1$$
$$= \frac{26 - 3 + 2(0)}{1} + 1 = 24$$

Substituting the values, the FLOPs equation becomes:

$$\text{FLOPs} = B \times (3 \times 3 \times 32 \times 24 \times 24 \times 64) \times 2$$
$$= B \times 21,233,664$$

The estimated time complexity with different batch sizes is theoretically calculated as shown in Table I.

Batch Size	Theoretical FLOPs
16	339,738,624
32	679,477,248
64	1,358,954,496
128	2,717,908,992
256	5,435,817,984

TABLE I
THEORETICAL FLOPS FOR DIFFERENT BATCH SIZES

2) Space Complexity: The space complexity includes the number of parameters and activations. For the Conv2d-2 layer (`nn.Conv2d (32,64,3,1)`) with the following parameters:

- Kernel size: $K_h = 3$, $K_w = 3$
- No. of input and output channels: $C_{in} = 32$, $C_{out} = 64$
- Stride: $S = 1$
- Padding: $P = 0$ (assume no padding as not defined)
- Batch Size: B

The parameters in the Conv2d-2 layer is given by:

$$\text{Parameters} = (K_h \times K_w \times C_{in} \times C_{out}) + C_{out}$$
$$= (3 \times 3 \times 32 \times 64) + 64$$
$$= 18,496$$

The activations in the Conv2d-2 layer is given by:

$$\text{Activations} = H_{out} \times W_{out} \times C_{out} \times B$$
$$H_{out} = W_{out} = 24 \text{ (as computed before)}$$
$$\text{Activations} = 24 \times 24 \times 64 \times B$$
$$= 36,864 \times B$$

The memory required is the sum of the space for parameters and activations. Assuming 32-bit precision (4 bytes per value):

$$\text{Memory (bytes)} = \text{Parameters} + \text{Activations}$$
$$= (18,496 \times 4) + (36,864 \times B \times 4)$$

The estimated space complexity for different batch sizes is theoretically calculated as shown in Table II.

Batch Size	Theoretical Memory (Bytes)
16	2,433,280
32	4,792,576
64	9,511,168
128	18,948,352
256	37,822,720

TABLE II
THEORETICAL MEMORY USAGE FOR DIFFERENT BATCH SIZES

B. Experimental Results

The actual performance metrics were collected by running the experiments on the NYU Greene cluster with a Conv2d-2 layer implemented using PyTorch. The results for one batch size is shown in Figure 3.

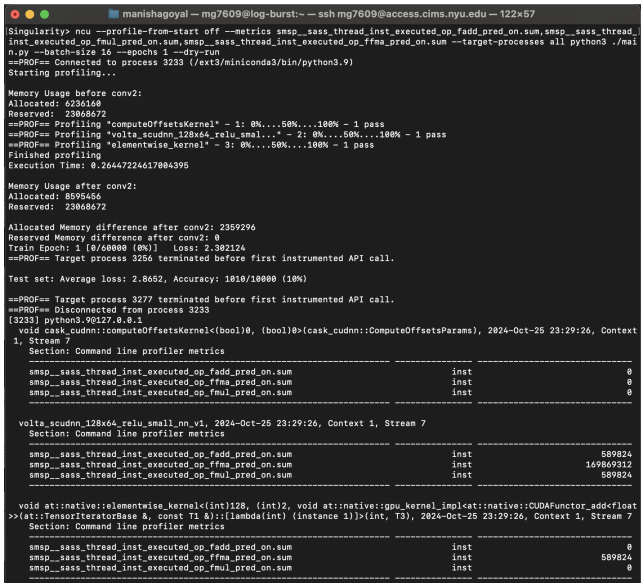

Fig. 3. Output showing FLOPs and memory usage metrics

1) Time Complexity: The FLOP count was measured by summing metrics for addition, fused multiply-add (doubled for the dual operation of multiply and add), and multiplication operations, as shown in the equation below.

$$\text{FLOP count} = \text{fadd\_pred\_on.sum} + \text{fmul\_pred\_on.sum}$$
$$+(2 \times \text{ffma\_pred\_on.sum})$$

Table III and Figure 4 show the measured FLOPs for each batch size:

Batch Size	Add	MultiplyAdd	Multiply	Total FLOPs
16	589,824	169,869,312	589,824	340,918,272
32	1,179,648	339,738,624	1,179,648	681,836,544
64	2,359,296	679,477,248	2,359,296	1,363,673,088
128	4,718,592	1,358,954,496	4,718,592	2,727,346,176
256	9,437,184	2,717,908,992	9,437,184	5,454,692,352

TABLE III
MEASURED FLOPS FOR DIFFERENT BATCH SIZES

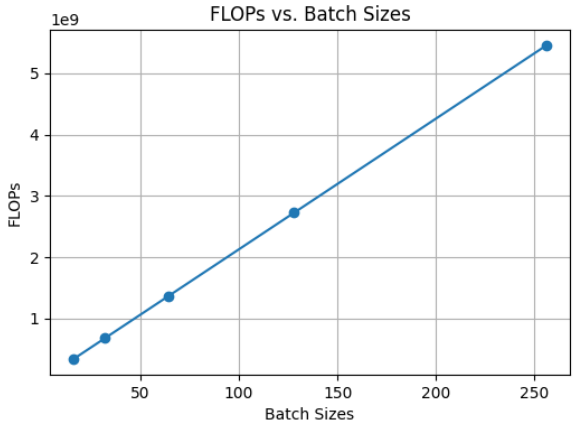

Fig. 4. FLOPs vs batch size

2) Space Complexity: The space complexity was measured by summing both static and dynamic memory requirements (`torch.cuda.memory_allocated()` and `torch.cuda.memory_reserved()`) immediately before and after the Conv2d-2 layer. The measured memory for each batch size is shown in Table IV and Figure 5

Batch Size	Allocated Memory (Bytes)	Reserved Memory (Bytes)
16	2,359,296	0
32	4,718,592	0
64	9,437,184	20,971,520
128	18,874,368	18,874,368
256	37,748,736	37,748,736

TABLE IV
MEASURED MEMORY USAGE FOR DIFFERENT BATCH SIZES

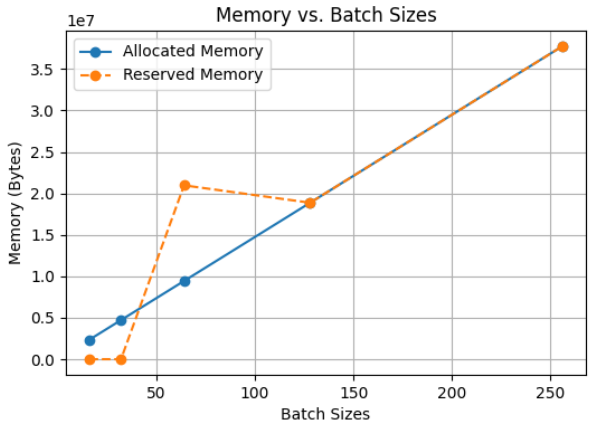

Fig. 5. Memory (Bytes) vs batch size

C. Comparison of Theoretical and Measured Results

The results show close alignment between the theoretical estimates and empirical measurements for both FLOPs and memory usage across various batch sizes, though minor discrepancies arise due to several factors inherent to practical implementations on GPU hardware.

1) Time Complexity Comparison: As shown in Table V and Figure 6, the measured FLOPs closely match the theoretical estimates across all batch sizes. This consistency suggests that the theoretical calculation effectively captures the core operations performed by the Conv2d-2 layer.

Batch Size	Theoretical FLOPs	Measured FLOPs
16	339,738,624	340,918,272
32	679,477,248	681,836,544
64	1,358,954,496	1,363,673,088
128	2,717,908,992	2,727,346,176
256	5,435,817,984	5,454,692,352

TABLE V
COMPARISON OF THEORETICAL AND MEASURED FLOPS

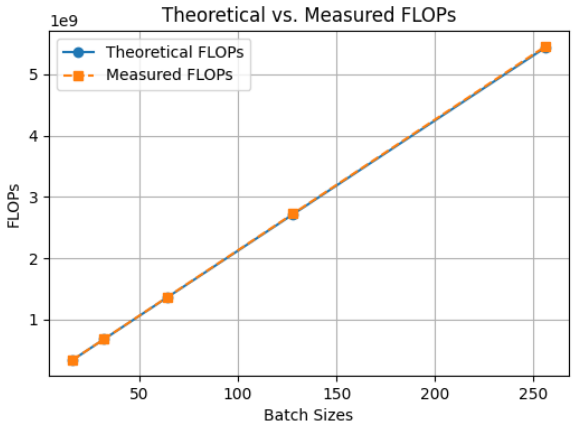

Fig. 6. Flops vs batch size (theoretical vs measured)

However, minor discrepancies are observed, with the measured FLOPs slightly exceeding the theoretical estimates. This deviation may be attributed to:

- **GPU-Specific Optimizations**: The NVIDIA V100 GPU uses specialized tensor cores and CUDA optimizations, such as instruction-level parallelism and efficient handling of FMAs, which slightly alter the raw FLOP count.
- **Profiling Overhead**: The profiling tool, NVIDIA Nsight Compute, introduces a slight overhead when recording metrics. This profiling overhead could contribute to minor variations, especially as batch size increases, where more operations are profiled.
- **Round-Off Errors and Accumulation**: The accumulation of floating-point operations across large batch sizes can introduce round-off errors, subtly affecting the FLOP count observed by the profiler.

These factors together lead to slight but consistent increases in measured FLOPs relative to theoretical predictions. Despite these minor differences, the results confirm that the theoretical FLOP model is a reliable predictor of computational load, particularly for understanding scaling behavior with batch size.

The results show a clear trend of increasing FLOPs as batch size grows, which aligns with theoretical expectations. Since the total FLOPs for a convolutional layer are directly proportional to the batch size, increasing the batch size linearly

increases the computational workload. Each additional batch element requires the same set of convolutional operations, which scales the total FLOPs in a linear fashion. This trend is critical for performance tuning in deep learning, as it highlights that larger batch sizes can lead to a significant increase in computation time, especially on GPU-accelerated systems where large-scale parallelization is possible. Consequently, batch size should be chosen carefully to balance the computational load with the available hardware resources, ensuring that the GPU remains efficiently utilized without introducing excessive latency.

2) Space Complexity Comparison: Table VI and Figure 7 provide a comparison between theoretical and measured memory usage.

Batch Size	Theoretical Memory (Bytes)	Measured Memory (Bytes)
16	2,433,280	2,359,296
32	4,792,576	4,718,592
64	9,511,168	9,437,184
128	18,948,352	18,874,368
256	37,822,720	37,748,736

TABLE VI
COMPARISON OF THEORETICAL AND MEASURED MEMORY USAGE

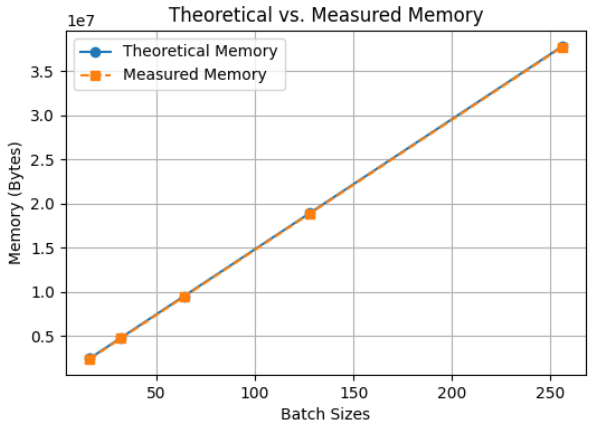

Fig. 7. Memory (Bytes) vs batch size (theoretical vs measured)

The measured memory usage closely aligns with theoretical estimates, with minor variances that become more evident at larger batch sizes. Several factors explain these differences:

- **Memory Optimizations**: PyTorch includes dynamic memory management features such as memory caching, tensor reuse, and pre-allocation. For example, PyTorch's CUDA memory allocator reserves blocks of memory that are reused in subsequent operations, which can result in reduced memory allocation needs for each batch.
- **Reserved vs. Allocated Memory**: While the theoretical model focuses on the memory strictly required for computations, the measured memory includes reserved memory, which PyTorch allocates in advance to prevent fragmentation and reduce allocation time during computation.
- **Precision Handling and Memory Fragmentation**: GPUs often use precision-specific optimizations that reduce memory overhead by compressing intermediate activations or leveraging half-precision for less critical calculations. These optimizations could lead to memory usage lower than theoretical expectations, especially in reserved memory allocations.

Overall, the empirical memory usage remains close to theoretical estimates, confirming the accuracy of the initial model. However, the subtle optimizations in PyTorch's memory handling reveal that the theoretical space complexity can be an upper bound, with actual implementations potentially requiring less memory.

The measured memory usage exhibits a linear growth pattern with batch size. This trend stems from the fact that each additional batch element requires storage for its activations (intermediate feature maps), which increases the overall memory footprint proportionally. Memory usage scales with batch size due to the need to hold multiple feature maps in memory during the forward and backward passes. In practice, this trend imposes a constraint on the maximum feasible batch size, especially on memory-limited hardware like GPUs, where excessive memory consumption could lead to out-of-memory errors or reduced computational efficiency due to frequent memory swapping. Understanding this trend is vital for configuring batch sizes that fully utilize the GPU's memory capacity without surpassing its limits, thereby optimizing both computation and memory usage in deep learning applications.

IV. CONCLUSION

This study provided a detailed analysis of the Conv2d-2 layer's time and space complexity in a CNN trained on the MNIST dataset, with results confirming the close alignment between theoretical estimates and empirical measurements. The trend of linearly increasing FLOPs with batch size emphasizes the need for careful batch size selection to manage computational load effectively, especially when leveraging GPU acceleration. Similarly, the trend of increasing memory usage highlights the direct relationship between batch size and memory demands, an important consideration for optimizing memory utilization in resource-limited environments. Minor discrepancies observed between theoretical and measured results were attributed to GPU-specific optimizations and PyTorch's memory management, underscoring the efficiency of modern deep learning frameworks. This analysis provides a foundation for understanding how CNN performance scales with batch size and highlights the critical role of optimizing both computation and memory usage. Future work could expand this study to examine additional layers and architectures, offering a broader perspective on performance scaling across diverse deep learning applications.

REFERENCES

[1] NVIDIA. (2024). Nsight Compute Documentation. Retrieved from https://docs.nvidia.com/nsight-compute
[2] PyTorch. (2024). PyTorch Documentation. Retrieved from https://pytorch.org/docs/stable/index.html