Fall 2023 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Implementing Convolutional Neural Networks in Parallel

Harsh Sahu (hsahu@wisc.edu)
Varan Shukla (varan.shukla@wisc.edu)

December 13, 2023

Abstract

Our project details the development and implementation of a parallelized forward pass for a custom Convolutional Neural Network (CNN) using CUDA in C++, aimed at optimizing runtime performance in deep learning models. With a focus on enhancing inference speed, this project endeavored to distribute computation tasks using GPUs to expedite the processing of large datasets and complex CNN architectures.

As we have established in our project proposal, we delved into the creation of optimized kernels utilizing the CUDA API in C++ for key CNN operations such as Convolution, Pooling, Padding, Dense Layers, and Activation Functions. Rather than relying solely on existing libraries like cuBLAS, our approach involved crafting custom kernel functions to enable a finer level of control over parallelization. The methodology included the segmentation of tasks into distinct functional units, each responsible for a specific operation within the CNN pipeline. Through a streamlined job processing workflow, the input data is traversed through a skeleton CNN model to generate the final output. We conducted benchmarking experiments comparing the runtime of our parallelized implementation (C++ GPU) against non-parallel implementation (C++ CPU) and prevalent state-of-the-art deep learning framework, PyTorch.

In our study, we are able to establish our approach's superiority over PyTorch, in inference speed, in all configurations of the CNN. We are also able to get as much as 10X speedup, in some cases, compared to C++ CPU (non-parallel) implementation.

Link to Final Project git repo: https://git.doit.wisc.edu/VARAN.SHUKLA/hpc-finalproject.git

Contents

1. General information

1. Name: Harsh Sahu
2. Email: hsahu@wisc.edu
3. Home department: Computer Sciences
4. Status: MS Student
5. If applicable: Varan Shukla. His report is identical to this.
6. Choose one of the following two statements:
   I am not interested in releasing my code as open-source code.

2. Problem Statement

The primary aim of this project is to develop and implement a parallelized forward pass for a custom and configurable Convolutional Neural Network (CNN) using CUDA. This endeavor is motivated by the essential need to optimize runtime performance in deep learning models. By parallelizing key operations within a CNN, such as Convolution, Pooling, Padding, Dense Layers, and Activation Functions, we aim to significantly reduce inference time.

Our motivation stems from active involvement in deep learning research, where enhancing the efficiency of deep learning models' runtime stands as a fundamental task. The ability to distribute computational tasks across multiple processors or GPUs offers the potential to process large datasets and intricate CNN architectures more rapidly.

By comparing the runtime of our parallelized solution against the non-parallel approach and existing state-of-the-art PyTorch library, we seek to demonstrate the efficiency gains achieved through our custom parallelization strategy.

Ultimately, this project aligns with the broader goal of optimizing deep learning model runtime, contributing to our research objectives, and gaining comprehensive insights into algorithmic optimizations for CNN operations on GPU architectures.

3. Solution Description

In this project, we implement a single forward pass of a configurable custom CNN. The architecture follows a standard CNN architecture, a Convolution module that includes a bunch of PADDING - CONV - ACTIVATION layer stack (CONV stands for convolution), followed by the Pooling module which consists of some pooling layers, followed by the Dense module containing dense layers. The outermost dense layer contains one neuron (or unit) outputting a single value for each input instance. The user can define and configure the following hyperparameters of the architecture:

1. 3-D Input Shape: (Batch Size, Height, Width, No. of Channels)
2. Convolution module:
    a. No. of PADDING - CONV - ACTIVATION layer stack
    b. Filter size of CONV layer
    c. No. of filters in the CONV layer
    d. Stride of filters
    e. Activation function
3. Pooling module:
    a. No. of pooling layers
    b. Kernel size of pooling layers
    c. Stride of kernels
4. Dense module:
    a. No. of dense layers
    b. No. of units in each dense layer (except the last one which will always contain a single neuron)

We use the following default values for our hyperparameters:

1. 4-D Input Shape: (1, 32, 32, 3)
2. Convolution module:
    c. No. of PADDING - CONV - ACTIVATION layer stack: 1
    d. Filter size of CONV layer: 3X3
    e. No. of filters in the CONV layer: 32
    f. Stride of filters: 1
    g. Activation function: ReLU
5. Pooling module:
    a. No. of pooling layers: 1
    b. Kernel size of pooling layers: 2X2
    c. Stride of kernels: 1
6. Dense module:
    a. No. of dense layers: 2
    b. No. of units in each dense layer (except the last one which will always contain a single neuron): 128

For the implementation of convolution methods we used *Unified Memory*, so that it was much simpler to manage the images and model weights. This also allowed us to use the same wrapper code for CPU convolution layer methods as well, as we could pass the same pointers around to both CUDA and CPU implementations.

We also relied on using *shared memory* in the convolution method, where we loaded the kernel into the shared memory. Here the same kernel is used by every thread. Each thread in the convolution computes one cell of the output image, so loading the kernel into shared memory allows considerable speedup. Similar parallelization has been used in pooling, activation, dense, and padding layers, i.e. each individual thread works on processing a single cell of the output thread.

We chose a block size of 1024 threads as that allowed us to have 16x16x4 threads in the 3 dimensions of the block. We didn't need more than 4 threads along the Depth as the input or output images are rarely more than 3 in depth (i.e. Number of channels). These blocks would move in 3-D spatially over the output image to compute blocks of cells. Figuring out the number of blocks launched was thus straightforward. The CPU methods were quite straightforward as they didn't require much index manipulation.

4. Experiments and Results

The primary goal of our work is to conduct a comprehensive evaluation assessing the performance gain achieved through the implementation of a Convolutional Neural Network (CNN) in C++ utilizing CUDA. Firstly, our experiments center around the comparison of runtime efficiencies between two implementations in C++: one utilizing CUDA for parallel processing on the GPU and another relying solely on CPU computation.

Furthermore, we also benchmark these C++ implementations against PyTorch's CNN implementation, both using CPU and GPU. Through these experiments and comparative analyses, our objective is to get comprehensive insights into the runtime efficiencies offered by C++ CUDA, benchmarked against C++ CPU and PyTorch implementations.

Note: In each of the following experiments, every hyperparameter apart from the varying hyperparameter(s) assumed the default value (unless otherwise mentioned).

Input Size

Here, we compare runtime over increasing Input Size of the image. For example, if the input size is 2^5, then the input shape would be [1, 32, 32, 3]

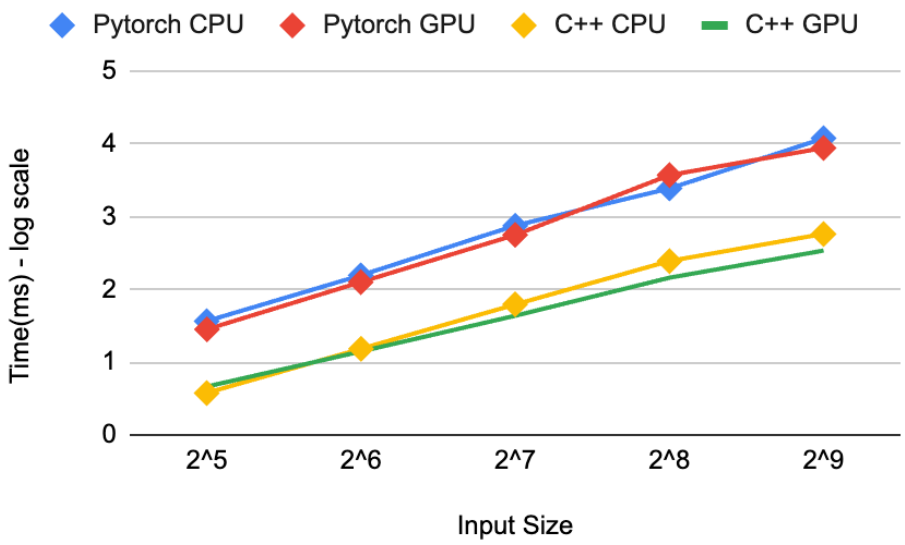

Figure 2

As we can see, C++ implementation beats pytorch implementation in both CPU and GPU settings. This is something we expected for the following reasons:

- **Low-Level Optimization:** C++ with CUDA allows for low-level optimization, enabling us to finely tune and optimize specific parts of the code for the GPU architecture. This level of control often leads to more efficient memory management and computation, compared to higher-level frameworks like PyTorch, which may not exploit every hardware-specific optimization.
- **Framework Overhead:** PyTorch, being a high-level deep learning framework, provides flexibility and ease of use but introduces an additional layer of abstraction. This abstraction comes with certain overheads such as dynamic graph creation, automatic differentiation, and other abstractions that might slightly slow down computations compared to a more tailored and optimized C++ CUDA implementation.
- **Kernel-Level Parallelism:** Our C++ CUDA implementation exploits parallelism at the kernel level, allowing for simultaneous execution of operations across thousands of GPU cores. This level of parallelism can be tightly controlled and optimized in C++, potentially leading to more efficient utilization of GPU resources compared to higher-level abstractions present in PyTorch.

Further increasing the input size caused the RAM exhaustion and consequently crash of the session on Google Colab. So, we could not run PyTorch on higher input sizes. Therefore, as a subsequent step, we analyzed higher input sizes for just C++ CPU and C++ GPU implementations

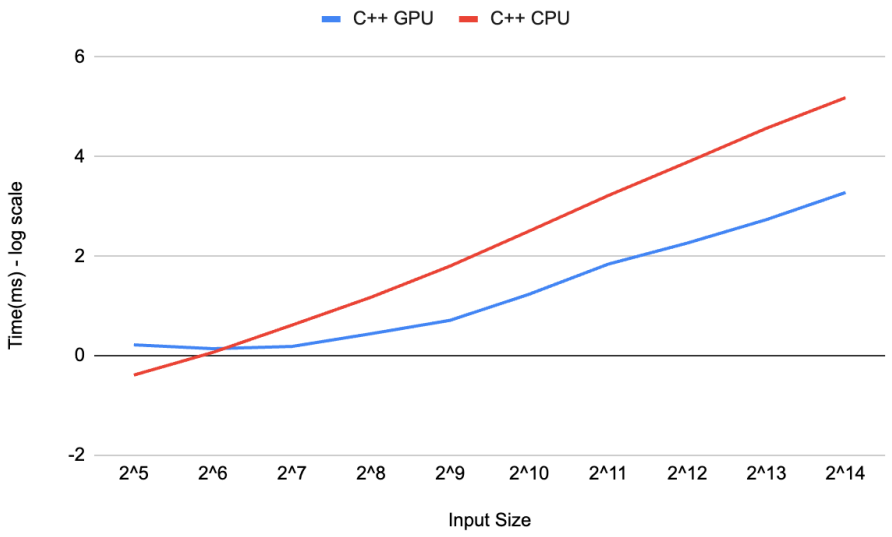

Figure 3

We observe in Figure 2 as well as in 3, that the C++ GPU (or CUDA) implementation beats the CPU one on higher input sizes. The gain in performance also increases with increasing Input Size.

Filter Size

We increase the size of a NXN filter in a convolution layer to assess the increase in running time.

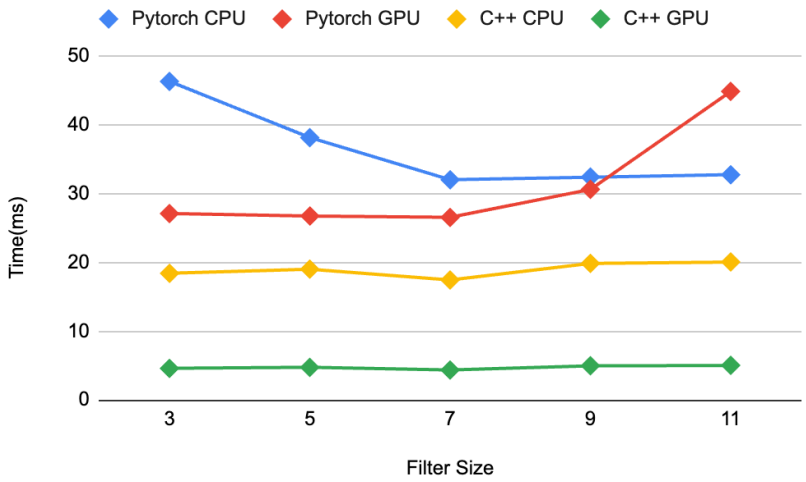

Figure 4

Note: We have performed this experiment, and all the following ones, with a limited number of hyperparameter values as Google Colab's RAM could not handle larger values.

We do not see the runtime increase as we increase the filter size except for PyTorch GPU implementation. Additionally, overall, C++ GPU runs the fastest followed by C++ CPU.

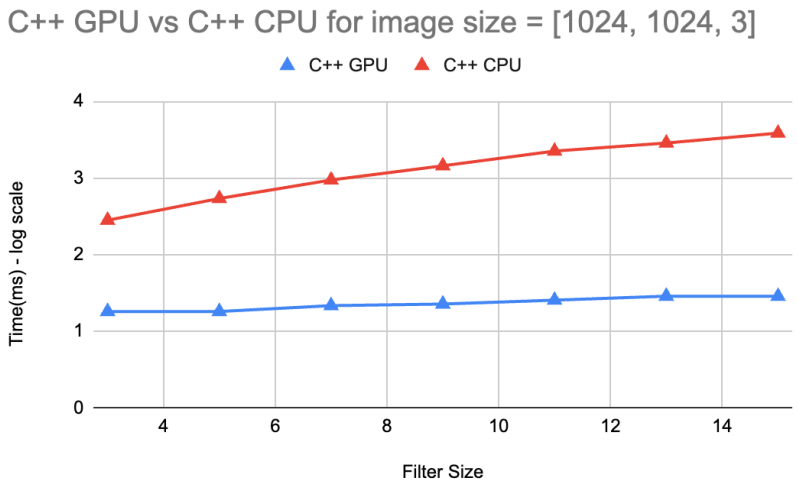

Figure 5

We also compare C++ runtime for varying filter sizes for higher image dimensions (1024, 1024, 3) where we notice a linear increase in runtime as we increase the filter size.

No. of Filters

We evaluate the runtime compared to the varying number of filters within a convolutional layer

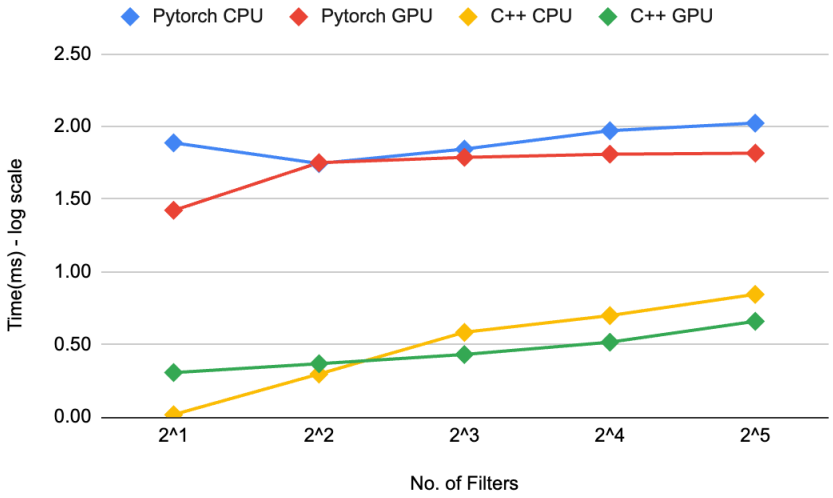

Figure 6

Increasing no. of filters in the convolution layer linearly increases the runtime of all the implementations. In this experiment, we see the same order of runtime where C++ GPU < C++ CPU < Pytorch CPU < Pytorch GPU

No. of Layers

Here we perform 3 experiments, varying the number of layers in the convolution, pooling and dense modules individually to evaluate the runtime. The size of the convolution module implies the number of PADDING - CONV - ACTIVATION layer-stack.

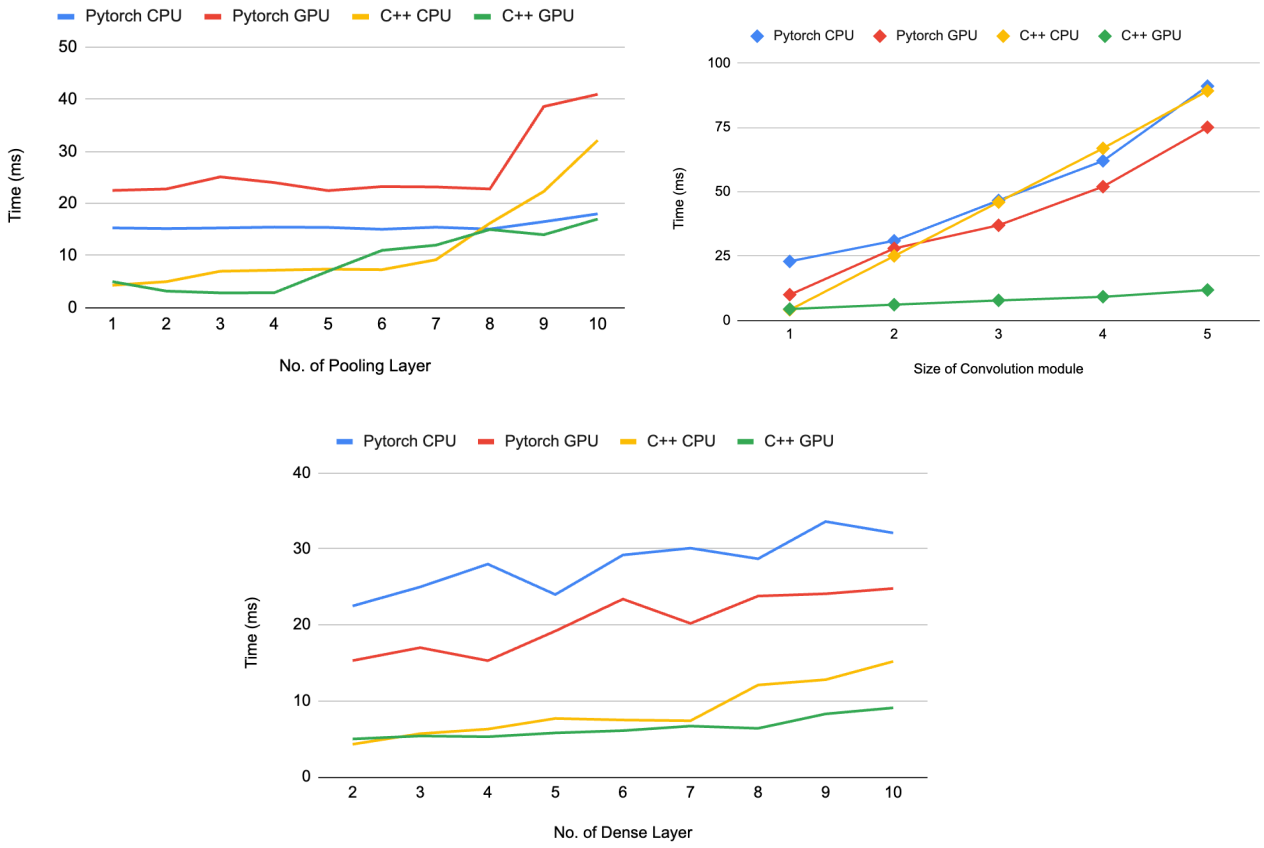

Figure 7

Convolution module: We see an almost linear increase in the RunTime as we increase the number of layers. C++ GPU is observed to beat the other implementations clearly, which seemed similar. Pytorch GPU has done slightly better than Pytorch CPU.

Pooling module: We observe a general linear trend for all implementations. C++ GPU is performing best for large numbers.

Dense module: Pytorch's runtime is increasing for an increasing number of layers, whereas C++ remains almost constant with a slight uptick. C++ GPU has the lowest runtime as the number of layers increases.

5. Deliverables

Here, we provide the description of all the submitted scripts and files in our repo:

config.json

- This is the JSON file that contains the default values of all our hyperparameters used to run our experiments for CNN. We take these parameters and iterate over them while varying one of the parameters to get our results. The rest of the parameters are fixed to values given in the json.

CNN_pytorch.py

- This Python script contains the code for running the PyTorch implementation of our CNN on a single image.
- We use *torch.cuda.Event()* to time the inference of Pytorch's model.
- We have run all our Pytorch CPU as well as GPU experiments on Google Colab

convolution.cu

- This is the most important file that contains most of our C++ and CUDA.
- It contains all the methods for the implementation of all CNN layers in the CPU as well as GPU, these include: convolution, dense, pooling(max), relu(activation), and padding. Each of these layers also has a corresponding CPU implementation. The kernels for global functions are also defined in this file, which include: convolution_kernel, pooling_kernel, add_padding_kernel, relu_kernel, and dense_kernel.

convolution.cuh

- This contains the structures definition for image(img) and kernel(ker) for use with CNN. It also contains the declaration of the convolution methods.

benchmarkModels.cu

- This file contains the main wrapper function that defines the CNN for a given set of hyperparameters, initializes the input and the weights, and runs the CNN model with the help of methods defined in convolution.cu

benchmarkModelsCPU.cu

- This file contains a similar wrapper as above for initialization and running of the network but instead uses the CPU-only implementation of layer methods.

cnn.cu, cnn.sh

- It contains the main function to initialize and run a single layer of padding followed by convolution followed by ReLU activation, and ends with max pooling. The shell script compiles and executes the layer based on the input parameters provided, so we can vary input image sizes and kernel sizes.

bench.sh, benchCPU.sh

- It contains the scripts to compile the benchmarking code and run it with varying input parameters. For our experiments, we fix the parameters based on those given in the config file, followed by varying a single parameter using the shell script which is provided as input to the benchmarking code. A similar script is written for both CUDA-accelerated and CPU-only versions.

We have performed our C++ experiments on the Euler cluster using slurm and the timing results provided are derived after running on the same, although we did occasionally use Google Colab to run our CUDA program to easily debug the issues faced by directly running, instead of using slurm. When the experiments were run, the results were directly copied into Excel, so the .out files in Repo don't necessarily contain meaningful timing outputs.

Running Pytorch implementation:

CNN_pytorch.py reads the hyperparameter values from *config.json.* In order to run the PyTorch's CNN for a given configuration, run the following command from the terminal:

```
python CNN_pytorch.py
```

Running C++ CUDA and CPU implementations:

Use the shell scripts to run the models and layers based on the description on the .sh files given above. The final committed files describe the usage of the executable files in the comments of the script, so one can vary the input parameters in a for loop and get the timings by running the experiments for models and varying the parameters. The parameters can be changed accordingly, and using the same as the ones in the results should ideally give the same timing outputs.

6. Conclusion and Future Work

From the conducted experiments, it is evident that the performance of CNN implementations varies significantly based on the hardware utilization and design specifics. The following conclusions can be drawn from our experiments:

- **The C++ implementation leveraging CUDA for GPU parallel processing outperformed C++ CPU** across varying input sizes, number of layers (or sizes of modules), filter sizes, and number of filters.
- We observe an exponential increase in the runtime for all 4 implementations as we increase the Input Size and No. of filters
- We observe a linear increase in the runtime for all 4 implementations as we increase the number of layers in any of the modules (convolution, pooling as well as dense).
- **Our C++ GPU (using CUDA) implementation significantly and consistently beats the state-of-the-art deep learning library Pytorch, across various experimental parameters**. We can attribute this performance superiority to low-level optimization, minimized framework overhead, efficient memory handling, and kernel-level parallelism achievable with C++ CUDA.

The findings underscore the advantages of leveraging low-level optimization and GPU parallel processing in C++ CUDA, resulting in enhanced CNN performance. Additionally, limitations in PyTorch's handling of larger input sizes were highlighted, which may impact its scalability in certain scenarios. Overall, these findings emphasize the significance of hardware-specific optimizations and parallelism in achieving superior CNN performance.

One of the likely future works would be to train a convolution neural network from scratch in C++ using CUDA instead of just using the model to get a prediction. It would require implementing the entire training pipeline, including backpropagation, optimization, and iterative updates of network parameters on the GPU. This work will require a deep understanding of both CNN architectures and CUDA programming and would be significantly harder to implement. As training of a CNN is more computationally expensive, it will also give us more opportunity to increase efficiency and we expect even larger speed gains with C++ CUDA implementation.

Learnings from ME759

Through the course, we learned how to leverage GPU threads to parallelly perform the same operation (convolution, pooling, padding) on different regions of the input matrix. We also leveraged the concept of shared memory (for example in storing a filter in a convolution operation) to load all the necessary variables just one time, required by all threads, in order to get further speedups. This course also taught us about synchronization mechanisms in CUDA, which we used to handle concurrent operations. We were also able to efficiently manage the memory and its Transfers between the device and the host along with effectively timing computations.

7. References

[1] https://cs231n.github.io/convolutional-networks/
[2] https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
[3] https://blog.paperspace.com/writing-cnns-from-scratch-in-pytorch/
[4] https://machinelearningmastery.com/building-a-convolutional-neural-network-in-pytorch/
[5] https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
[6] https://www.simplilearn.com/tutorials/deep-learning-tutorial/convolutional-neural-network
[7] https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/