**CSCI GA 3033 Cloud and Machine Learning  
Homework 3: Performance study of (a small part of) Neural Network**

Name: Aashiq Mohamed Baig  
Net ID: amb1558

***Introduction***

In this report, we will be covering performance measures of Machine Learning scripts using a profiler. The 2 main metrics to measure are the time complexity via number of flops and space complexity via memory. We will be using Nvidia NCU to verify if our calculation for the flops via pen and paper is indicative of the true flops computed on the system for 1 epoch in a 2D convolution layer. Similarly, we will use PyTorch profiler to verify if the memory we calculated is indicative of the true memory used by the system. Using PyTorch profiler, we will extract logs for our Machine Learning job which we will further analyze in Tensorboard. We will write a python Machine Learning script which will run on the MNIST dataset; a dataset that has over 60,000 images of handwritten digits; and classify them into a digit from 0 to 9. There are 2 variations to our Python script; 1 for the flop calculation and one for the memory calculation i.e. NCU and PyTorch profiler.

***MNIST Database***

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It 60,000 training images and 10,000 testing images. Each image is a grayscale image of size 28x28. Researchers have experimented with many different models and have achieved a near human level of accuracy with these models.

A picture containing arrow

Description automatically generated

Fig 1. Sample of images from the MNIST database

***Background***

Before we dive into computing the time and space complexity, let us understand the terminology and the shorthand being used. Here is the representative code of the 2D convolution in the Neural Network. The 2D convolution is a simple operation; you start with a kernel, which is simply a small matrix of weights. This kernel “**slides**” over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

**Conv1**: nn.conv2d (1, 32, 3, 1)

**Conv2**: nn.conv2d (32, 64, 3, 1)

The order of the parameters passed is (input filters, output filters, kernel size, stride)

We will refer to the following parameters with the below shorthand

* input filters/channels as Cin
* output filters/channels as Cout
* kernel size as k
* stride as s

The input image size is 28x28x1. We will follow the output produced in both convolution steps. Additionally, we will be using Hout and Wout to represent the output height and width of the image after convolution.

**Conv1**: (1 x 28 x 28) -> (32 x 26 x 26)

**Conv2**: (32 x 26 x 26) -> (64 x 24 x 24)

For our calculations, we will be taking the 2nd convolution layer.

***Time Complexity Pen and Paper Method***

In this method [1], we are trying to calculate the total number of additions and multiplications done to complete a single convolution. The entire number is multiplied by 2 to account for multiplications and additions since for a 3\*3 kernel, we have 9 multiplications and 8 additions. This is approximated up to 2\*9. The time complexity is indicated of the FLOPs measurement.

***Space Complexity Pen and Paper Method***

Some things to note in the following method [2]. The “Params” here refers to the weights that need to be stored for the kernels and which are used in the actual convolution. The first term is the output of the convolution. The entire sum is multiplied by 4 to account for each value being a floating-point number and each floating-point number is 4 bytes.

The approximate calculation for memory is 0.22 MB.

***Machine Learning Model***

In Fig 2, we can see the exact model operations being used to identify the images in the MNIST dataset. There are 6 operations in total. However, these are not the exact order of operation calls during forward propagation. This is how the operations are initialized in the init function of the Neural Network Model.

Text

Description automatically generated

Fig 2. Machine Learning model for classifying images in MNIST

As we have seen above, conv1 and conv2 handle the actual convolutions. The dropout layer randomly sets some of the input units to 0 with a frequency of p (in Fig 2) which is 0.25 and 0.5 respectively. This is done during the training phase and it’s helps to prevent overfitting. Finally, we have 2 fully connected layers, a.k.a. linear layers, which convert the input if it is in 3D format to a 2D linear layer. This is exactly what happens in the first fully connected layer. The 2nd fully connected layer reduces the number of features. Here, we can see the 2nd layer reduces the number of features to 128.

Text

Description automatically generated

Fig 3. Neural Network Class and functions for init and forward propagation

Fig 3 details the exact steps and order of all operations in the neural network. Note that there are 2 additional parameters here. We have a ReLU layer [3] which implements non-linearity and a 2D max pool layer [4] which reduces the dimensions of the matrix by 2. It does this to down sample a representation of the input such that we may still make assumptions about the features contained in the sub-regions binned [5].

Fig 4 shows us the summary of the model. It is achieved by calling summary from the “torchsummary” package. We specify the model and the dimensions of the input. It gives us a detailed view of the model, the resulting dimensions after each operation and the number of parameters required/learned/stored for that layer. It also returns the total number of trainable and non-trainable parameters as well as the size required to store the model outputs and parameters in memory/disk.

Text

Description automatically generated

Fig 4. Summary of the Machine Learning model

***FLOPs Calculation***

We will be running our code on NYU’s Greene cluster to calculate the number of floating-point operations for a single convolution layer. We will be using NVIDIA Nsight Compute to get the fadd, fmul, and ffma parameters which can be used for FLOPs calculation.

A picture containing text

Description automatically generated

**Phase 1: Setting up a container with latest dependencies**

Step 1: SSH into access on the CIMS server

Text

Description automatically generated

Step 2: SSH into the Greene cluster. This is where we will run our scripts

Text

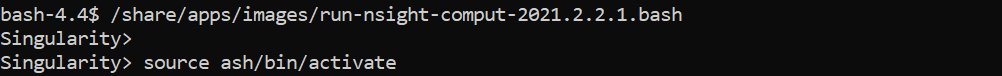
Description automatically generated

Step 3: Do a module load to use only use Python 3.8. Following that do an ssh burst and start a slurm interactive job on the cluster.

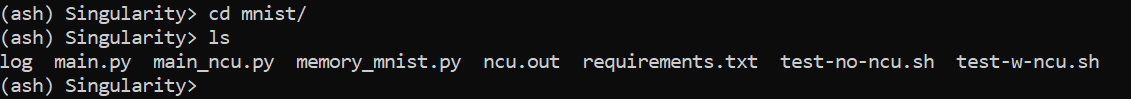
Graphical user interface, text

Description automatically generated

Step 4: Create a python virtual environment. Following that, create a Singularity container and activate the environment inside the container.



Step 5: I have created a folder called mnist with all my required files. Enter the file and install the packages inside requirements.txt.



Text

Description automatically generated

Step 6: Verify that torch can be used inside a python script and check the version

Shape

Description automatically generated with medium confidence

**Phase 2: Analyzing the FLOP count with and without NCU**

Step 1: First we will run a simple MNIST script (Vanilla) which will do a simple dry run for 2 epochs. The code is not unlike to the model we have seen above. It uses the file main.py which does not have any NCU code. Using this, we can verify if our script can run on the Greene cluster.

A picture containing shape

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Step 2: Now that we have verified that our vanilla script runs on the cluster, we will make changes to the code to get the flop parameters from NCU. We have passed the epoch number as a parameter and are only running the NCU computation for epoch 4 over the 2nd convolution layer.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Text

Description automatically generated with medium confidence

The above images show the outputs from the NCU for the 2nd comvolution for epoch number 4. It is necessary to make the calculations for a single epoch.

**Flop Calculation from NCU:**

smsp\_\_sass\_thread\_inst\_executed\_op\_fadd\_pred\_on.sum +

smsp\_\_sass\_thread\_inst\_executed\_op\_fmul\_pred\_on.sum +

smsp\_\_sass\_thread\_inst\_executed\_op\_ffma\_pred\_on.sum \* 2

= 1384448 + 18432 + 2359296 + 2359296 + 2\*2359296

= 10,840,064

**Manual Flop Calculation:**

Here is a comparison graph for how the FLOPs calculation changes with different batch sizes.

Chart, bar chart

Description automatically generated

Chart, line chart

Description automatically generated

In most cases, the pen and paper method for calculating the FLOPs is greater. This is because the system does some caching or optimization to reduce the number of actual operations done. There is also a huge increase in the number of computations for small sizes since GPUs are not designed to handles smaller batch sizes. We can see more standard behavior for higher batch sizes. System calculated FLOPs is measured and Pen and Paper FLOPs is estimated.

***Memory Calculation***

We will be using the PyTorch profiler to compute the amount of memory required for 1 convolutional layer. I have already described the method to estimate the amount of memory required in the pen and paper method for space complexity. Here, we will modify the vanilla code to insert a profiler step before and after the convolution layer. The code will generate a call stack log which can be used to analyze the code steps including the forward and backward propagation. There are many more blocks to analyze.

Text

Description automatically generated

Text

Description automatically generated

Snippet of the generated log:

Text

Description automatically generated

Generated Log Trace:

We use the chrome tracer to upload the log in chrome://tracing.

Timeline

Description automatically generated with medium confidence

In order to better analyze the logs, we will use tensor board.

Step 1: Copy the logs to your PC and open in Jupyter Notebook

Graphical user interface, application

Description automatically generated with medium confidence

Step 2: Run the following snippet and then open it up in tensor board; localhost://6006



Graphical user interface, application

Description automatically generated

Step 3: Navigate to the Memory View and access the table at the bottom. This is a snippet of the memory utilization of batch size 1.

Graphical user interface, application

Description automatically generated

Using out pen and paper method, we estimated the required memory would be 0.22 MB. This is the calculation for a batch-size of 1. If we multiply it by the batch size, we have the memory requirement for a specific batch size.

From the tensor board image attached above, we can see that it uses 40.6 MB for a batch size of 64. By multiplying 64 with 0.22 MB we get 14.08 MB which is the estimate for the memory computation. There is a marked difference between the measured memory and the estimated memory because there are many kinds of overheads that are actually involved when doing convolutions. The compiler uses many different temporary variables for value storage and often times, extra memory is used in PyTorch to optimize the run-time. To avoid errors, we may also need to save the values temporarily in order to recover them in case of node failure (due to multiple GPUs being used).

Here is a comparison graph for how the memory changes with different batch sizes for both measured and estimated.

Chart, bar chart

Description automatically generated

Chart, line chart

Description automatically generated

***Changing the Conv Layer dimensions***

This is one example of changing the dimensions of the 2D convolution. Here, we are changing the output filters of the first and 2nd layer Additionally, we need to change the input parameters for the linear layer. Output of the first layer will be 26x26x20 and the output of the 2nd layer will be 24x24x40. After passing through a pooling layer, the output will become 12x12x40 which is 5760.

Below, we can also see a snippet of the training happening for this new dimensions of the model. If we do not change the linear layer dimensions it will give us a matrix multiplication error.

Text

Description automatically generated

Text

Description automatically generated with medium confidence

***Conclusion***

From this report, we have gone over how to PyTorch profiler to record the memory required for a convolution layer. We have also seen the usage of NVIDIA Nsight (NCU) to calculate the FLOPs for a convolution layer. We use different formulas to estimate the number of flops and the memory used for this calculation/ However, the estimated flops and memory is often different from the measured values, due to the architecture of the cloud which is not taken into consideration. There are overheads and other methods used for optimizations which can deviate from the estimated value. Overall, it is still a good method for estimation.

***References***

[1] <https://www.thinkautonomous.ai/blog/deep-learning-optimization/#calculating-the-flops-in-a-model>

[2] <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>

[3] <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>

[4] <https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html>

[5] <https://computersciencewiki.org/index.php/Max-pooling_/_Pooling>