High Performance Computing for Machine Learning
Spring 2018 - Assignment 1

Daniel Rivera Ruiz
Department of Computer Science
New York University
`drr342@nyu.edu`

1 Inference in C and Python

1.1 Dimensions and memory size (Q1 and Q2)

The following table shows the dimensions and memory size for the two layers of the neural network:

	Dimension			Memory Size		
Layer	x	W	z	x	W	z
0	[1 × 50,176]	[50,176 × 4,000]	[1 × 4,000]	392 KB	1.50 GB	31.25 KB
1	[1 × 4,000]	[4,000 × 1,000]	[1 × 1,000]	31.25 KB	30.52 MB	7.81 KB

The memory size calculations consider the standard size for the double precision type in `C` (8 Bytes).

1.2 Memory bandwidth measurement (C1)

The micro-benchmark designed and implemented in `lab1.c` uses an array of randomly generated `double` values. Given an array length of 367,001,600 and `double` values of 8 Bytes, the size of the array in memory is 2.8 GB. The minimum time it took to fetch the array from memory was 0.523 seconds, which yields a **memory bandwidth of 5.23 GB/s**.

1.3 Compare to Intel specifications (Q3)

According to the specifications of the reserved CPU in the Prince cluster, *Intel® Xeon® Processor E5-2660 v3*, the **maximum memory bandwidth is 68 GB/s**. However, the experimental memory bandwidth measured with the benchmark from the previous exercise only represents **7.7%** of this value.
The main reason for the poor performance of the code (memory-wise) is that we are creating a data structure that is much bigger than the cache size of the CPU (25 MB) and performing very few floating point operations on it. Under these conditions the process becomes memory-bound and a bottleneck occurs due to the latency inherent to fetching the data from memory. In order to improve this situation, the following measures can be taken:

- Divide the data structure into smaller blocks the size of the cache and perform operations per block. By doing this, the data blocks can be stored in cache and the number of accesses to main memory will be reduced considerably.
- Optimize the code using vectorization and/or parallelism in order to perform more operations at the same time. Implementing these optimizations will make the bottleneck move towards the CPU-bound region in the roofline model of the code.

1.4 Python (C2)

In `lab1.py` there is an implementation in `Python` of the network described in exercise Q1 using the definition provided in the assignment to initialize x_0, W_0 and W_1. The following results where obtained for one full step of inference on the network:

- Execution time (excluding initialization): **144.660640 s**
- Checksum value $S = 11,901,764,160.000206$

The checksum value S is defined as the sum of all the elements of the output vector z_1.

1.5 NumPy (C3)

Also in `lab1.py`, an optimized implementation of the network using the `NumPy` package is provided. The results for the inference step on this network are the following:

- Execution time (excluding initialization): **0.089247 s**
- Checksum value $S = 11,901,764,160.000221$
- Speed up with respect to the execution time of C2 $= \frac{t_{C2}}{t_{C3}} = \frac{144.660640}{0.089247} = 1,620.90$

1.6 C (C4)

In `lab1.c` there is a third implementation of the network in plain `C`. The results for the inference step on this network are the following:

- Execution time (excluding initialization): **3.769837 s**
- Checksum value $S = 11,901,764,160.000206$
- Speed up with respect to the execution time of C2 $= \frac{t_{C2}}{t_{C4}} = \frac{144.660640}{3.769837} = 38.37$

1.7 Intel MKL library (C5)

The final implementation of the network is also in `C`, but it is optimized using the *Intel MKL Library*. The results for the inference step on this network are the following:

- Execution time (excluding initialization): **0.123293 s**
- Checksum value $S = 11,901,764,160.000206$
- Speed up with respect to the execution time of C2 $= \frac{t_{C2}}{t_{C5}} = \frac{144.660640}{0.123293} = 1,173.31$

2 Training in PyTorch

2.1 PyTorch code (C6)

The `PyTorch` implementation of the neural network and data loaders described in the assignment is in the file `lab1.pytorch`.

2.2 Time measurements (C7)

The `time.monotonic()` function was used throughout the code from the previous section to measure the time it took to complete certain tasks within the program. The following are the measurements obtained for a training process of 5 epochs using a `DataLoader` with 1 worker:

- Aggregate DataLoader I/O time: **1,405.8912 s**
- Aggregate DataLoader preprocessing time: **66.5703 s**
- Aggregate data loading time during training: **1,435.5756 s**
- Overall training time: **1,475.6217 s**

2.3 I/O optimization (C8)

To optimize the I/O performance in the code from the previous step, we reduce the time required to fetch the batches during training time by incrementing the number of workers the `DataLoader` utilizes. The following table shows the aggregate waiting times for different amounts of workers:

Workers	Waiting Time (s)
0	1511.56
1	1435.58
2	877.62
3	453.00
4	306.54
5	**238.05**
6	250.97
7	263.32
8	240.91

As we can see, for this particular process **the optimal number of workers is 5**. Beyond this number, the aggregate waiting time stops decreasing.

2.4 Training optimization (C9)

Finally, we explore the optimization of the whole training process by GPU-enabling the code using `CUDA` and trying different optimizer algorithms. With the optimal number of workers found in the previous exercise and training the model for 10 epochs, the following table summarizes the results of these experiments:

Processor	Optimizer	Average Epoch Time
CPU	SGD with Momentum	23.10 s
GPU	SGD with Momentum	9.58 s
GPU	SGD	9.55 s
GPU	ADAM	9.54 s

As it was expected, taking advantage of the GPU processing capabilities results in smaller values for the average epoch time.