Programming Assignment 5: Memory, Parallelism, and Precision

CS4787/5777 — Principles of Large-Scale ML — Fall 2023

Project Due: Monday, December 4, 2023 at 11:59pm

Late Policy: Up to two slip days can be used for the final submission.

Please submit all required documents on gradescope.

This is a group project. You can either work alone, or work **ONLY** with the other members of your group, which may contain AT MOST four people in total. Failure to adhere to this rule (by e.g. copying code) may result in an Academic Integrity Violation.

Overview: In this project, you will be evaluating the impacts of memory allocation and parallelism on a learning algorithm. Since we are close to the end of the semester, I have planned this project to be relatively short so that you can finish it **within one week**, but please don't use this as a reason to put off starting on the project until the last day!

Instructions: This project is split into three parts: the implementation and evaluation of a version of SGD that doesn't allocate memory in its inner loop, the evaluation of the performance of the built-in multithreading capabilities of numpy, and the implementation of a multi-threaded version of SGD in Python.

Note that for the multi-threading section of the assignment, you will need to run on a CPU with more than one core to get interesting results. If you are running on a VM (most of you are not), you will need to make sure that it is set up to allow for more than one thread of execution. To do this, you may need to increase the number of cores available in the VM settings.

Note: There is no autograder for this project, since you can verify that your code outputs the correct thing by just comparing it to the baseline methods provided. Grading will be performed based on your lab report and on manual inspection of your code.

Also note: The experiments in this project can take some time to complete. On my computer, they take about 7 minutes to run in total (across all experiments). Please be sure to leave enough time to complete the experimental exploration.

Part 1: Memory.

Background: Most numpy operations by default allocate a new array in which to return their result. This extra allocation can be costly, and can cause harmful memory effects that slow down computation. Fortunately, numpy has a mechanism that lets us avoid this. Most numpy operations have an optional argument `out` which lets you specify an already-allocated array into which the result of the operation will be written. This lets us perform a numpy operation without allocating any new memory. For an algorithm like SGD, this means that we can perform all the allocations we need before entering the inner loop, and then just do all the computations in the inner loop with allocation-free numpy operations. This can speed up the execution of the loop, as we'll see here. (For this part of this assignment, you should run experiments with the number of threads set to 1.)

1. Adapt your function `sgd_mss_with_momentum` (SGD + momentum with minibatching and sequential scan) from Programming Assignment 3 to return only the model arrived at at the end of all the iterations (rather than returning a list of models produced every so often).
2. Implement a function `sgd_mss_with_momentum_noalloc` that runs the same computations as your `sgd_mss_with_momentum` function, but avoids any allocations in the inner loop by pre-allocating any needed arrays.
    - To do this, the inner loop should be entirely numpy functions called with a `out` parameter that is a pre-defined array.

- Additionally, be sure that your inner loop does not contain slices in it (snippets like `Xs[:,ii]` where `Xs` is a numpy array). If you want to use a slice, pre-allocate it outside the inner loop by running something like `numpy.ascontiguousarray(Xs[:,ii])`, which allocates that slice as a dense array.
- To validate your implementation, you should check that the output of this new function is close to the output of your original `sgd_mss_with_momentum` function.

3. Run both of your functions (`sgd_mss_with_momentum` and `sgd_mss_with_momentum_noalloc`) on the MNIST dataset using the following hyperparameters:
   - Step size $\alpha = 0.1$.
   - Momentum hyperparameter $\beta = 0.9$.
   - Minibatch size $B = 16$.
   - L2 regularization parameter $\gamma = 0.0001$
   - Run for 20 epochs.

   Measure the amount of wall-clock time it takes to run all 20 epochs using each function. Which one, if any, is faster? How much faster is it?
4. Now repeat the previous experiment for multiple values of the minibatch size, $B \in \{8, 16, 30, 60, 200, 600, 3000\}$. Plot the resulting wall-clock times for both functions together on a single figure with x-axis the minibatch size and y-axis the wall clock time. How does the relative performance of the two functions compare as the minibatch size is changed? Can you explain why you observed this?

Part 2: Parallelism in Numpy.

1. Identify how many cores are on the CPU you will be using for this programming assignment.
2. Change the number of cores used implicitly by numpy for this part by setting the environment variables

```
os.environ["OMP_NUM_THREADS"] = "n"
os.environ["MKL_NUM_THREADS"] = "n"
os.environ["OPENBLAS_NUM_THREADS"] = "n"
```

   where `n` is replaced by the number of threads you want to use (some number larger than 1). You will need to restart python to make this change.
3. Re-run the experiment in Part 1.4 using this automatic multi-threaded parallelism and plot the results on the same figure as in Part 1.4. How does the speed of the algorithm using multithreading compare to the speed when no multithreading is used?
   - Note that there should now be four series on this figure, two for 1-thread execution and two for multi-threaded execution.

Part 3: Multithreading in Python. So far we have looked at parallelizing implicitly using numpy multithreading (supported on top of OpenMP and controlled by those environment variables). There is another way we can take advantage of the multi-core capabilities of our CPU: to explicitly use python multithreading capabilities. This is what we will do in this part.

1. Implement a function `sgd_mss_with_momentum_threaded` that runs the same computations as your `sgd_mss_with_momentum_noalloc` function, and uses python multithreading functionality from the `threading` standard library. Your function should use multithreading to parallelize across subsets of the minibatch. That is, if you are running on M machines with a minibatch of size B where $B = M \cdot B'$, then to compute the minibatch SGD update

$$W_{t+1} = W_t - \alpha \cdot \frac{1}{B} \sum_{m=1}^{M} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(W_t).$$

   I have provided some starter code that gives a skeleton for an implementation that does the following.

- First, on each parallel worker c, compute

$$g_m \leftarrow \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(W_t)$$

  in parallel.
- Then, wait on a **barrier**, a parallel synchronization construct that waits for all workers to arrive at the barrier before allowing any worker to proceed past the barrier.
- Next, on a single thread only, compute the update to the model

$$W_{t+1} \leftarrow W_t - \alpha \cdot \frac{1}{B} \sum_{m=1}^{M} g_m.$$

- Have all the threads wait on the barrier again.
- Repeat for the next iteration of SGD.

However, you are free to modify this starter code as you think is appropriate. (If you are looking for more of a challenge, and/or you want to learn more about parallel programming, you can try to implement this function from scratch.)

2. Run your function `sgd_mss_with_momentum_threaded` on the MNIST dataset using the following hyperparameters:
   - Step size $\alpha = 0.1$.
   - Momentum hyperparameter $\beta = 0.9$.
   - L2 regularization parameter $\gamma = 0.0001$
   - Run for 20 epochs.

   for each of the minibatch sizes in Part 1.4, i.e. $B \in \{8, 16, 30, 60, 200, 600, 3000\}$. Plot the wall-clock time to run all the epochs on the same figure as for the other experiments. In order to make sure that your cores are not overloaded, you should set the number of cores used implicitly by numpy back to 1 (allowing the cores to be used explicitly by your implementation).
   - Now there should be five series on this figure, the four from before plus the new series for manual multithreading in python.

   How does the wall-clock time of your manually multithreaded code compare to that of the other algorithms? Can you give an explanation for any trends you observed?

Part 4: The Effect of Precision.

In class, we talked about how using lower-precision arithmetic can improve the throughput of machine learning applications. Here, we will test whether this is true for this training task. Numpy uses 64-bit floating point arithmetic by default: we will see here whether using 32-bit floating-point arithmetic (which is the standard precision used for most ML applications and ML frameworks) can improve the training speed.

1. Implement two functions, `sgd_mss_with_momentum_noalloc_float32` and `sgd_mss_with_momentum_threaded_float32`, which work the same as the similarly named functions you implemented earlier, except that they do their computations using 32-bit floating point numbers. You should do this by copying your implementations from Parts 1 and 3, and adapting them to use single-precision floats. You may find the numpy datatype `numpy.float32` and the `dtype` argument to be useful here.
   - Hint: the implementation for this function should require only a copy-paste of your existing 64-bit implementations, and adding `dtype=numpy.float32` and sometimes `.astype(numpy.float32)` as needed.
   - It is simplest to implement these functions to take 64-bit numbers as input and produce 64-bit parameters as output, and have them convert to/from 32-bit on entry to/exit from the function. But, since there is no

autograder, you are free to use whatever semantics you find convenient for whether the inputs/outputs of these functions are 32-bit or 64-bit numbers.

2. Run both your functions on the MNIST dataset using the following hyperparameters:
   - Step size $\alpha = 0.1$.
   - Momentum hyperparameter $\beta = 0.9$.
   - L2 regularization parameter $\gamma = 0.0001$
   - Run for 20 epochs.

   for each of the minibatch sizes in Part 1.4, i.e. $B \in \{8, 16, 30, 60, 200, 600, 3000\}$. Plot the wall-clock time to run all the epochs on the same figure as for the other experiments. In order to make sure that your cores are not overloaded, you should set the number of cores used implicitly by numpy to 1 (allowing the cores to be used explicitly by your implementation).
3. Run your function `sgd_mss_with_momentum_noalloc_float32` using implicit numpy multithreading (as in Part 2) on the MNIST dataset using the following hyperparameters:
   - Step size $\alpha = 0.1$.
   - Momentum hyperparameter $\beta = 0.9$.
   - L2 regularization parameter $\gamma = 0.0001$
   - Run for 20 epochs.

   for each of the minibatch sizes in Part 1.4, i.e. $B \in \{8, 16, 30, 60, 200, 600, 3000\}$. Plot the wall-clock time to run all the epochs on the same figure as for the other experiments. Now there should be eight series on this figure:
   - The baseline method, which uses 64-bit floats, with no implicit numpy multithreading.
   - The no-allocation method, using 64-bit floats, with no implicit numpy multithreading.
   - The no-allocation method, using 32-bit floats, with no implicit numpy multithreading.
   - The baseline method, which uses 64-bit floats, with implicit numpy multithreading.
   - The no-allocation method, using 64-bit floats, with implicit numpy multithreading.
   - The no-allocation method, using 32-bit floats, with implicit numpy multithreading.
   - Your explicitly multithreaded method, using 64-bit floats (no implicit multithreading).
   - Your explicitly multithreaded method, using 32-bit floats (no implicit multithreading).

   How does the wall-clock time of your 32-bit code compare to that of the other algorithms? Can you give an explanation for any trends you observed?

What to submit:

1. An implementation of the functions in `main.py`.
2. A lab report containing:
   - A one-paragraph summary of what you observed during this programming assignment.
   - The (single) figure displaying the wall-clock time results from all parts.
   - Wall-clock time measurements from Part 1.3, and a short explanation of which function is faster and how much faster it is.
   - Answers to questions from Part 1.4.
   - The number of cores on your CPU in Part 2.1 and the method you used to identify that, and answers to questions from Part 2.3.
   - Answers to questions from Part 3.2.

- Answers to questions from Part 4.3.

Setup:

1. Run `pip3 install -r requirements.txt` to install the required python packages