Parallel vs. Serial Performance of Hyperparameter Gridsearch on Gradient Descent

Eddie Shim
MPCS 52060 Parallel Programming
eddieshim@uchicago.edu

06/09/2020

1 Overview

In this project I'll be presenting an implementation and analysis of parallelized grid search of hyperparameters in gradient descent. Gradient descent is a popular optimization algorithm used in statistics to calibrate parameters numerically. Roughly speaking, it allows us to find the minimal point in a convex equation using an iterative approach that allows us to slowly "descend" towards a minimum with each step with a distance that's calculated by taking the gradient of the cost function. In each iteration, we update our parameters with the following equation:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} cost(\theta)$$

where θ represents a vector of parameters we wish to optimize, α is the learning rate, and $cost(\theta)$ represents the total cost of our model (eg: sum of the residuals squared). This iteration is repeated until we reach an error lower than a threshold tolerance, or until we have iterated more than a predefined maximum number of iterations. In batch gradient descent, we pass over the entire set of training samples per iteration. Thus the larger our training set, the more expensive our calculation gets. This can be a limiting factor in big data. Take for example calibrating a logistic regression on a training set of $n = 1,000,000$ with $25,000$ independent variables, perhaps predicting a disease based on data from 25,000 genes. Each iteration of gradient descent

would require calculating 25 billion terms, and if our convergence takes 1000 steps, then in total this problem requires calculating 2.5 trillion terms. Performing hyperparameter grid search exponentially increases our computational difficulty, as it requires an independent calibration of the models for each permutation of hyperparameters from which we wish to select the optimal hyperparameter set from.

2 Linear Regression

Since the focus of this project is on parallel computation techniques, I decided to keep the statistical framework as simple as possible. Thus we'll be testing the results of our gradient descent on a univariate regression model with an intercept, which is defined as:

$$\hat{y} = \beta x + \mu + \varepsilon$$

where $\hat{y}$ represents our prediction (dependent variable), β is the linear coefficient, x is our input data (independent variable), μ is our intercept, and ε is random noise, assumed to be normally distributed $\varepsilon \sim N(0, \sigma^2)$, independent, and homoskedastic. For our loss function, we'll use the standard mean squared error:

$$cost(\beta, \mu) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

$$cost(\beta, \mu) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta x_i + \mu))^2$$

From here we can derive the gradients of our two parameters:

$$\frac{\partial cost(\beta, \mu)}{\partial \mu} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})$$

$$\frac{\partial cost(\beta, \mu)}{\partial \beta} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - \hat{y_i})$$

Thus with each iteration, we'll be updating our parameters by an amount of (α * gradient). In order to tame our α value and ensure that each step of our descent takes an efficient step, we first normalize the data with the following:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Normalization of our independent variable is a feature scaling technique that allows gradient descent to converge must faster by changing the shape of our cost function. It's a necessity in our experiment, as without it we would need to calibrate different α values that scale with sample size n.

3 Hyperparameters

Given our framework on calibrating linear regression through gradient descent, we require a few hyperparameters in order to control the learning process. These include the following:

- α: The learning rate which restrains the magnitude of our step with each iteration in our descent. If α is too high then gradient descent will not converge towards the minimum and if α is too low then it will converge too slowly towards the minimum
- $numEpochs$: The number of steps we take in our descent. The loss function of linear regression is convex, so there's no need to worry about converging towards local minimas that aren't the global minimum
- λ: The regularization term which restrains the magnitude of β to prevent overfitting
- $batchSize$: Describes the size of our training subset we use in iteration of the descent. In plain batch gradient descent, $batchSize$ is simply n, the full training set. Other varieties of gradient descent take advantage of training on randomly selected smaller subsets, namely stochastic gradient descent ($batchSize = 1$), and mini-batch gradient descent ($batchSize = m << n$, eg: $m = 10$).

4 Data

To proceed with our experiment I wrote a script `generator.go` which simulates sets training data with various sample sizes with the following parameters:

$$n = 100\text{k, } 300\text{k, } 1\text{m, } 5\text{m}]$$
$$\beta_{true} = 5$$
$$\mu_{true} = 100$$
$$\varepsilon \sim N(0, 25)$$

where ε is random error noise drawn from a normal distribution of mean 0 and variance 25. By simulating data directly with true parameters, we can ensure the accuracy of our experiment by comparing our calibrated parameters to objective truth.

5 Parallel Computing

There are several components to calculating gradient descent that are independent and thus present themselves as opportunities to be optimized using parallel computing.

5.1 Reader

First to process our JSON inputs, we spawn reader goroutines which can work with each task independently (see section 6.3 Input for more details on input format). The number of readers spawned is:

$$numReaders = \lceil numThreads * (1.0)/(5.0) \rceil$$

Each individual reader will attempt to grab *blockSize* amount (a user defined parameter) from the available JSON tasks available. If there are less than *blockSize* tasks left, the reader will grab all tasks available. These JSON tasks are read in from Stdin and processed into `struct Hyperparameters`, which are then stored in a channel and passed into a worker. This allows us to work on each JSON task in parallel. Since multiple readers must access Stdin in a thread-safe manner, we use a coarse-grained lock synchronization where we lock and unlock the Stdin each time a reader pulls tasks from it.

5.2 Worker

Each reader spawns an individual worker goroutine. The worker is where the bulk of our parallelization occurs. In order to process the *blockSize* number of tasks passed through reader's channel, we loop through and process each task sequentially. Within an individual task we first preprocess our hyperparameter grid by calculating all permutations of available hyperparameters (eg: a task with inputs `alpha = [.05, .1], numEpochs = [10, 100]` will generate a grid of 4 permutations). We then split the grid array (*workArray*) into *numThreads* equal sized chunks, where each chunk is size:

$$len(subworkArray) = \lceil len(workArray)/numThreads \rceil$$

We spawn *numThreads* goroutines to work individually on each chunk, since every model calibration is independent of one another. Each time a calibration finishes, we check if its hyperparameter set was the most optimal calibration. To do so, we use atomic variables $globalOptimalHyperParams$, $globalOptimalModelParams$, $globalOptimalMSE$. If our calibration's MSE is lower than than the $globalOptimalMSE$, we lock the atomic variables, update them to the new optimal parameters, and unlock the atomic variables. At the end, we use `sync.WaitGroup` to ensure all threads complete and have a chance to communicate their hyperparameter set.

5.3 Writer

Once each worker completes all *numThread* goroutines, we've calculated to optimal global hyperparameters. To output this to the user, we spawn an individual writer goroutine that writes our global optimal parameter set into a user designated csv file (for more detail on output data, see section 6.4).

6 Implementation

6.1 Overview

Below are the main components of the code:

- `proj3\calibrate\calibrate.go`

  This contains our main function, and performs serial or parallel hyperparameter grid search

- `proj3\data\generate.go`
  This is a helper class that contains functions related to handling and generating our data
- `proj3\regression\regression.go`
  This is a helper class that contains functions related to our statistical model framework
- `proj3\hyperparams.txt`
  This is the filepath of our input JSON tasks which contain the hyperparameter values we wish to grid search on
- `proj3\training_data\trainingData_[n].csv`
  This are the filepath(s) of our input training data
- `proj3\output\outputParams[n].csv`
  This are the filepath(s) of our output csv files, the optimal hyperparameter sets
- `proj3\gridsearchPerformance.ipynb`
  This is a Python notebook containing the code used for speedup analysis

6.2 Program Specifications / Readme

There are two parts to running this program (note: I used Windows Powershell terminal but I've translated the below commands into Linux commands).

1. First, we have to generate and cache our training data as described in section 4. The following terminal command line will generate a dataset of size n and cache it into a csv file named `trainingData_[n].csv`

   ```
   go run calibrate.go -g=[n]
   ```

2. Second, we can run our gradient descent in serial or parallel using the following terminal command:

   ```
   go run calibrate.go -i="[filepathData]" -t=[numThreads] -b=[blockSize] < [filepathJSON]
   ```

   Where `-i` is the filepath of our cached training data generated from the first step (eg: `-i="trainingData_100000.csv"`), `-t` is the number of threads to run with (if set to 0 or not specified,

then defaults to serial run), and `-b` is *blockSize*, defined as the number of JSON tasks each reader should attempt to chunk and grab. The program also pipes in input JSON tasks which describe which hyperparam values we want to grid search on (eg: `hyperparams.txt`)

An example command looks like the following:

```
go run calibrate.go -i="C:\Users\eddie\Documents\MSCS\ParallelProgramming\eddieshim\proj3\
training_data\trainingData_100000.csv" -t=2 -b=2 < C:\Users\eddie\Documents\MSCS\
ParallelProgramming\eddieshim\proj3\hyperparams.txt
```

6.3 Input

There are two main inputs to `calibrate.go`:

- Training data, stored as a cached csv file
- JSON tasks file with user-input hyperparameter values

  Example task:

```
{"outpath": "outputParams1.csv", "alpha":
[".01", ".05", ".1"],"lambda": [] , "numEpochs": ["10", "100", "500"], "miniBatchSize": []}
```

6.4 Output

The output of `calibrate.go` is a csv file per JSON task, where each file details the optimal hyperparameter set and the corresponding model parameters calibrated with those hyperparameters. These filepaths are detailed in our JSON tasks (eg. `C:\Users\eddie\Documents\MSCS\ParallelProgramming\eddieshim\proj3\output\outputParams1.csv`) and should be unique per task. *Note: if α values are too large, and gradient descent explodes away from the minimum, then the expected behavior are NA hyperparameter values. Furthermore, if no values are specified for a hyperparameter in the JSON task, then the expected output for that hyperparameter is NA.

7 Conclusion / Performance Analysis

7.1 Testing Machine

Execution performance was measured on a Lenovo Thinkpad laptop with the following specifications:

- Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz
- Number of Cores: 4
- Operating System: Windows 10 64-bit
- RAM: 16.0 GB

7.2 Analysis

Using the Python notebook I wrote (`gridsearchPerformance.ipynb`) we'll be testing the performance of our parallel results by running the program on a set of different training data sets as described in section 4. To give a rough sense of the magnitude of the data, n =5m is 101 mb of data and n =300k is 6mb. Each timing measured is an average of 5 executions in order to prevent outliers and ensure results are more accurate. From each timing, we can calculate the speedup which is defined as:

$$speedup = \frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

As we can see in figure 1, there is a clear but somewhat limited to using the parallel version with more threads. We're able to achieve upto a nearly 2 times speedup when using 8 threads. There's a sharp performance increase at $numThreads = 2$, but performance staggers even with the use of additional threads beyond that. Furthermore, we see the largest performance gains at smaller sample sizes.

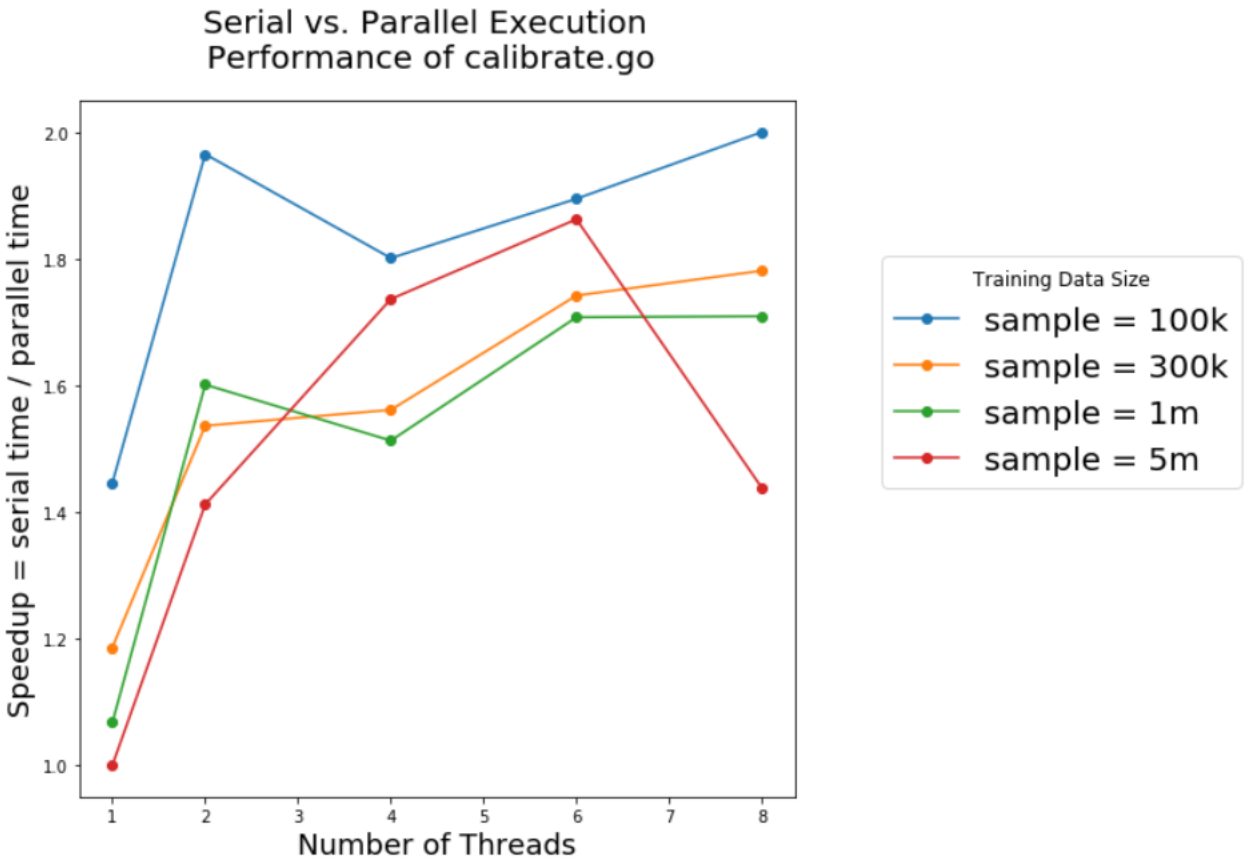

Figure 1: Performance Results

- What are the hotspots and bottlenecks in your sequential program? Were you able to parallelize the hotspots and/or bottlenecks in your parallel version?

  Hotspots are defined as areas in the program doing the most work, whereas bottlenecks are areas of the program causing the slow down in performance. As training sets get larger and larger, it's clear that the computation required to descend per epoch demands the most work thus it's both a hotspot and a bottleneck. By parallelizing calibration of models, we can take advantage of training multiple independent models simultaneously. In our parallel version, another bottleneck is the reader's ability to process JSON inputs from Stdin. Since multiple readers must access Stdin in a thread safe manner, we use a coarse-grained lock synchronization where we lock and unlock the Stdin stream each time a

reader pulls tasks, which forces it to work sequentially. Across each of our methods, worker contains the most computationally heavy portion of the overall work and thus is a hotspot for the parallel version of our program.

- What is limiting your speedup? Is it a lack of parallelism? Communication or synchronization overhead?

  The largest impact is training data size. Due to the fact we're using batch gradient descent, each thread performing gradient descent must train on the entire dataset before iterating through their epochs. Tools to tackle this bottleneck include switching from batch to stochastic or mini-batch gradient descent, and also vectorizing our computation instead of working purely iteratively.

- Does the training data size being processed have any effect on the performance?

  Yes, we can see that with larger training sets, there's a deterioration in performance gain due to the fact that individual model calibrations take up the majority of the bottleneck effect.