Parallelization of Machine Learning

Samuel Briceno, Adam Cook, Daniel Foster
Patrick Maley, Joshua Tobler

***Abstract*—This research proposes that a concurrent implementation of the gradient descent algorithm in machine learning is much more efficient than a sequential version. Provided a dataset containing the garage names at the University of Central Florida, occupancy rates, days of the week, and times, the algorithm will generate quadratic regression models of the data. While the sequential version is successful in accomplishing the creation of these gradients, they consume more time and memory than desired when testing at a higher number of epochs to obtain a more accurate model. Combining the mini-batch stochastic gradient descent approach along with Python's multiprocessing module to achieve parallelism yields faster results, while also allowing the user to create more accurate models.**

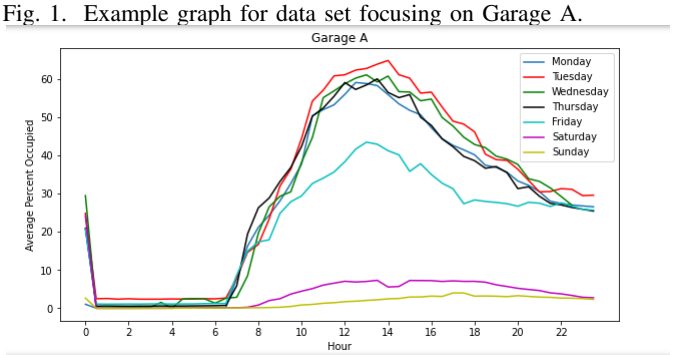
I. Introduction

MACHINE learning is a field of computer science that use statistics and data to find patterns or trends. Machine learning and artificial intelligence algorithms take in various inputs and expected outputs, and use those datasets to make predictions on unknown data. Often times, these algorithms can be applied to real-world scenarios. However, these can sometimes be too slow or costly to memory, but can be mitigated using parallel programming. We present a concurrent implementation of a quadratic regression that uses a dataset to predict the occupancy rate of a garage at the University of Central Florida on a given day and time.

II. What the Problem is

Our dataset gives us four columns of data: the garage, the occupancy represented as a percent value, the day of the week, and the time of day in military minutes since midnight, EST. The overall objective for this project is to, through the use of machine learning, predict how occupied a given parking garage will be at a given time and day. However, given the amount of data that the set provides, it could take a considerable amount of time to sequentially train the model, especially with a higher number of epochs. This is likely to result in significantly large execution times for training the model if the data is processed sequentially. Thus, this downside can be mitigated through utilizing multithreading or multiprocessing, we would be able to improve the performance of our model training that uses a large number of iterations,

Fig. 1. Example graph for data set focusing on Garage A.



which would yield better predictions than running a sequential training with less epochs.

There are several options for choosing the programming language to use to perform the concurrent machine learning. The three most popular choices to choose from are C++, Java, and Python, but we eventually decided to develop our experiments in Python. Python is not only an easy to understand language, but is "easily extensible and provides many high-quality mathematics packages" [1]. Namely, we would use the NumPy library to create vectors from the dataset that can be used to perform gradient descent. Python is also very flexible, and is useful for developing both large and small-scale projects.

III. State-of-the-art in this Research Area

Python has various frameworks and models to use for multiprocessing or multithreading in tandem with machine learning. Some of these frameworks are used exclusively for concurrent programming, while others have a broader use in machine learning with select modules for parallelization.

A. PyTorch

PyTorch is a popular deep learning library used in "defining layers, composing models, loading data, running optimizers, and parallelizing the training process..." [2]. Primarily developed by the AI Research Lab at Facebook, PyTorch provides tensor computing with "strong acceleration via GPU and deep neural networks built on tape-based automatic differentiation systems" [3]. Normally, the global interpreter lock (GIL) prevents a Python program from concurrently operating on multiple threads, acting as a universal mutual exclusive lock. The integrated multiprocessing module of Python mitigates this problem by effectively creating multiple GILs, one for each running thread, such that each thread no longer conflicts with one another or the original GIL. PyTorch further extends the Python multiprocessing library into its own torch.multiprocessing module.

B. Dask and Numba

A pair of frameworks that are often interchanged or used together are Dask and Numba. Dask is a library that parallelizes existing machine learning tools like Pandas and Numpy [4], whereas Numba is used to broadcast data into a vector. Dask is a naturally parallel framework due to how it approaches DataFrames from NumPy. By utilizing methods similar to divide and conquer, it effectively transforms any implementations into a concurrent version [5]. As for direct machine learning use, Dask has its own library called dask-ml, which is used in tandem with other libraries. Numba is a library that allows programmers to use vectors to perform mathematical operations over scalars, in a way similar to NumPy. However, once fitted with Dask, the parallelized algorithms are now broadcasted and can complete calculations faster and now over a broader range. Dask and Numba by themselves have relatively similar performance, but the two combined outperform the individual libraries [6]. The individual libraries perform at a linear, but significantly smaller rate than standard Python. But when combined, their runtime reaches a near constant value when operating on up to one million values.

C. pomegranate

pomegranate is an open source machine learning package for probabilistic modeling in Python that supports parallelizing batches for model fitting. These probabilistic models allows pomegranate to accelerate it's computation. It "divide(s) the data into several batches and calculate(s) the sufficient statistics for each batch locally" [7]. These local batches are then added back together to minimize the execution time of the model fitting. Python's global interpreter lock (GIL) would ordinarily prevent this, but pomegranate is able to bypass this by writing computationally intensive parts in Cython.

D. Qjam

Qjam is a framework that is focused on the parallelization of machine learning algorithms. To accomplish this, Qjam utilizes 3 type classes: a single Master class, several Worker classes, and several RemoteWorker classes. This Master-Worker relationship allows Qjam to execute multiple partitions of a program. The master class creates multiple instances of a RemoteWorker, which acts as a "proxy between the Master and the Worker" [1]. The Worker alleviates the stress of computation by executing the actual code once it receives the work from its RemoteWorker via ssh. Afterwards, the RemoteWorker receives the results from its Workers and relays them back to the Master class. The Workers also receive data chunks necessary for execution from the Master, and caches them if the appropriate data cannot be found. According to Beniz-Benet's benchmark of Qjam, it "performs better than the single-core every time" [1], achieving up to a 7.2x speedup on a load of 100,000 data patches using 16 Worker classes.

E. Parsl

Parsl is a parallel scripting library for Python. Parsl "augments Python with simple, scalable, and flexible constructs" which allow Parsl to "construct a dynamic dependency graph of components that it can then execute efficiently on one or many processors" [8]. Parsl identifies certain sections of code as opportunities for concurrent execution. Parsl can also construct dynamic dependency graphs of the tasks to aid in handling concurrent execution.

F. Joblib

Joblib is a popular Python library used to help achieve nested parallelism, which is defined as "hiding latencies of synchronization and serial regions" which are generally a downside of typical programs using NumPy [9]. Joblib does this through "transparent disk-caching" and "lazy re-evaluation" [10]. This memoization of pipeline jobs allows Joblib to execute code faster at the cost of minor disk space. This speedup is most prominent when implementing Joblib on code that will be rerun several times. Joblib's caching technique prevents the same code from executed more than necessary, saving computation time.

IV. Algorithms

A. Gradient Descent

For model training we will primarily use gradient descent, an algorithm that is used to predict coefficients of a regression curve for the dataset and updates the coefficients based on the loss function. With each update of the coefficients, the loss function is updated. The graph of this loss function with respect to a respective coefficient is called the gradient. The goal is to iteratively change coefficients so that the coefficients produce the least error, i.e. minimizing the gradient. We update the coefficients by moving down the error vs weight graph (descending the gradient) to find the optimal weights (coefficients). Coefficients are updated by taking the partial derivative of the loss function with respect to a certain coefficient. Once this gradient is computed, it is back-propagated to update coefficients by the factor of

a chosen learning rate. A higher learning rate will make bigger steps toward the local minimum of the function but might miss the minimum entirely, while a smaller learning rate is much more likely to reach the local minimum but will do so in more steps. [11] For this project, we have chosen to use unique learning rats for each parameter and a quadratic equation as your model. Our quadratic model looks as such:

$$\hat{y} = ax^2 + bx + c$$

The loss function we will use is called Mean Squared Error. This function is important as it puts higher weight on larger loss by squaring the difference between expected and created values.

$$Loss = \frac{1}{n}\sum((y - \hat{y})^2)$$

There are gradients for the error function for each of the three coefficients. These are simply the partial derivative of the loss function with respect to the considered coefficient (or parameter). They are as follows:

$$D_a = \frac{-2}{n}\sum(x^2(y - \hat{y}))$$

$$D_b = \frac{-2}{n}\sum(x(y - \hat{y}))$$

$$D_c = \frac{-2}{n}\sum(y - \hat{y})$$

After these losses are calculated, the coefficients are modified by an amount equal to the loss times the learning rate.

$$a = a - L1 * D_a$$

$$b = b - L2 * D_b$$

$$c = c - L3 * D_c$$

B. Mini-Batch Stochastic Gradient Descent

Gradient descent has the potential to be extremely slow for large datasets as regression requires that we compute the sum of loss and gradient for all data points each pass. As a result, we will train our data using stochastic mini-batch gradient descent. Stochastic gradient descent is a method for training data in which gradient descent is performed on only a singular random sample of a data set being used. Stochastic gradient descent was introduced as a way to get the correct gradient descent for far less computation, but because of only using one sample each pass, the results will end up being very noisy. As a middle ground between full-batch iteration and stochastic gradient descent, mini-batch stochastic gradient descent was introduced in which small batches of randomly sampled data are collected and gradient descent is performed in these small batches [12]. As an effect, mini-batch stochastic gradient descent is able to compute gradient descent more efficiently while still maintaining less noise than regular stochastic gradient descent.

Our proposed implementation would be to split the training our model by parellelizing Mini-Batch SGD. To parallelize it, we plan to split the number of epochs used in a sequentially trained model by a factor of N. N will represent the number of threads. by doing this we create N models with unique parameters which we will average at the end.

V. Approach

A. Python Libraries

To begin our approach, we first import five Python libraries to handle the dataset and perform multiprocessing. First, NumPy is used to vectorize the data so that pieces of the given dataset can be manipulated and operated on. As previously mentioned in Section II, the newly created vectors are used to calculate the gradient descent. Provided columns of data will be converted to an array and further sorted using the array() method.

Second, Pandas is used to both load and sort the dataset. While the program is capable of reading either a .csv or .xlsx file, we chose to use our .csv. file. Pandas is further used to sort the dataset in two ways: by column index and by condition. The columns are sorted by garage, occupancy, day, and time using the iloc() method, in order to split the dataset into four possible categories. Then, the loc() method is used to sort those columns further by certain conditions, such as specific days or times. For example, a dataset representing occupancies of Garage A on Fridays would use data2.loc[(day == 'Friday') AND (gar == 'Garage A')].

Third, Matplotlib is used to generate plots of data to visualize the curve of the dataset and the corresponding quadratic equation from the gradient descent. Though this library does not have major influence on calculations, it is useful for confirming the regression and testing input values. The scatter() method plots datapoints along the graph, and show() displays the graph to the screen. An example of a graph generated using Matplotlib can be seen in Figure 1.

Fourth, the Multiprocessing module was chosen to perform concurrent operations. We had some trouble finding a library that could provide true concurrent processes, however, we were able to find the multiprocessing library which is a Python library that is able to overcome the Python Global-Interpreter-Lock (GIL).

Finally, Time is used to track time intervals so that we can measure the efficacy of parallelized quadratic regression.

B. Gradient Descent

Our approach to Gradient Descent was completed as a python function that takes in the arguements [dataset, epochs, batchsize, n = float(batchsize), a, b, c, L1, L2, L3] where dataset is the scubbed dataset to be considered during the descent. Epochs is the number of iterations that the gradient descent will take. Batchsize represents the number of random sample to be taken from the dataset - this signifies our 'mini-batch'; n is the float version of batchsize used to divide the summation when computing gradients. a, b , and c are our initial parameter values. The last 3 arguments are the individual learning rates for parameters a, b, and c - L1, L2, and L3. Each parameter has a unique learning rate because we ran into some situations where the learning rate was too high and caused our function to diverge exponentially; or too low which cause our function to converge slowly. Thus, we decided to use individual learning rates to give higher precedence to parameters that have less effect on the overall function and vice versa.

The function used begins by starting a for loop that loops through the number of epochs. Then we take a slice of the dataset from the first entry to the size of the batch (for our implementation we mostly used a batch of 100) to use as the x and y variable. After computing x and y, we shuffle the dataset so that when we compute x and y in the next epoch we are using random values. Next the predicted value of y is computed using the current values of parameters a, b, and c. Then the gradients are computed with respect to each parameter. Finally, the parameters are updated using the values of their gradients multiplied with the individual learning rates. Once all epochs are completed, we return the final computed values for a, b, and c.

Fig. 2. Python code for Mini-Batch SGD Gradient Descent.

```
def Gradientdescent(garageAmanip, epochs = 5000, batchsize = 100, n = float(100) ,
                    a = -1, b = 0, c = -50, L1 = 0.00001,L2= 0.00005,L3 = 0.00008):
    for i in range(epochs):
        #set x and y and shuffle
        x = garageAmanip['Military_Time'][:batchsize]
        y = garageAmanip['Percentage_Occupied'][:batchsize]
        garageAmanip = garageAmanip.sample(frac=1).reset_index(drop=True)
        y_pred = a*(x**2) + b*(x) + c
        d_a = (-2/n) * sum(x**2 * (y - y_pred))
        d_b = (-2/n) * sum(x * (y - y_pred))
        d_c = (-2/n) * sum(y - y_pred)
        ##updating paramters using specific learning rates for each parameter.
        a = a - (L1 * d_a)
        b = b - (L2 * d_b)
        c = c - (L3 * d_c)
    return a,b,c
```

C. Multiprocessing

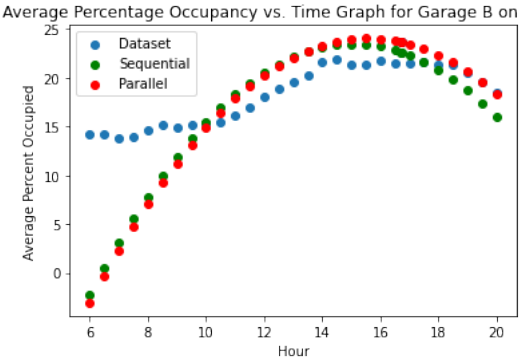
For implementing multiprocessing, we make use of the multiprocessing library. Specifically we use the Pool class from this library that when ran, completes functions using a pool of processors/threads. The Python multiprocessing library was the simplest method to overcome the Python Global-Interpreter-Lock (GIL). The Pool class is also interesting in that processes under execution are kept in memory. This is very important for reducing overhead as once a process finishes and the parameters are computed, we do not have to worry about copies of our dataset being kept in memory and potentially causing problems later on. With this library, we can split the gradient descent function into N processors that will each computer gradient descent with epochs/N epochs. After each thread finishes and returns the computed parameters, we will take the average of all parameters from each thread to compute the average a, b, and c parameters. Using the Time library from python, we will track the time it takes for all threads to complete gradient descent. Then we will run gradient descent sequentially with the normal amount of epochs and also track the time it takes to complete. Finally we will compare the time taken to complete between the sequential version and the concurrent version of gradient descent.

VI. Experimental Results

To test the results of our experiment, we compared the regression models of the gradient descents calculated both sequentially and concurrently. We initialized used the maximum number of CPU cores to test in the Google Colab environment, which happened to be 2. A total of 49 different data subsets were tested, one test for each garage along with each day of the week. Each test outputs several pieces of information: the time allotted for each individual test, the parameters calculated for each regression, and a graph to display both regression models along with the original data. For example, Figure 3 below shows a generated graph correlating to Garage B with Monday. The blue trend line represents the original data loaded in, the the green and red trend lines display the sequential and parallel models respectively.

Through extensive testing we were able to find starting parameters that worked best for this dataset. For our quadratic function, the best performing initial parameters were: a = -1, b = 0, c = -50. A is important to initialize to -1 because sometimes our model would diverge when the gradient descent would update a to be positive. We also found that initializing the learning rates L1 = 0.00001, L2= 0.00005, L3 = 0.00008 for a, b, and c respectively worked best for this implementation. Using learning

Fig. 3. Regression models from sequential and parallel computations



rates above 0.0001 for any of the parameters would cause our model to diverge and eventually cause errors due to creating numbers outside of Python's integer range. Parameter a should have the smallest learning rate because it has the largest affect on the quadratic function. b and c have even larger learning rates due to them having less of an impact with larger parameter change on the quadratic function.

After testing all possible combinations of garages and days, we can analyze the time consumption between both experiments and calculate an average speedup. Table I displays all of the allotted times for Garage B from Sunday to Saturday, generated from both the sequential model and the parallel model using two CPU cores. Using the average time consumption for each column, the calculated average speedup from sequential to concurrent is 1.14x, which indicates a moderate performance boost using parallel programming. Similarly, Table II showcases the performance of the gradient descent tested on 4 CPU cores. The average time consumed for both trials is higher, due to testing at 20,000 epochs instead of 10,000 epochs when increasing the number of cores. The speedup for this experiment 1.16x, indicating a similar speedup when tested on different numbers of cores. This speedup is good as we wanted a way to train models quicker than sequential versions, however, the speedup we are observing is too small considering we are splitting up the work amongst threads. The desired speedup should give us a speed that is constant among all different core situations. We believe this may be caused by some sequential bottleneck slowing down the multi threaded performance. There could be several factors coming into play for this. There can potentially be some operation in the gradient descent function that is causing the bottleneck. Maybe the shuffle operation that we use from pandas is causing the Python Global-Interpreter-Lock (GIL) to reactivate in the Pools class thus creating a sequential bottleneck. Or maybe the summation operation is causing this bottleneck. Regardless we had expected the time to remain constant for concurrent quadratic regression model training in 2-core and 4-core instances.

As for the accuracy for our models, there a quite a few observations we can make. Firstly, we can see that quadratic regression fits nicely for quite a few of the garage / day of week combinations but generally speaking the content of the dataset would have fit better using a polynomial regression implementation. As for the difference in accuracy between parallel (2 - core, and 4 - core) models and the sequential models; the general trend is that for most of the situations where the data is more quadratic in shape, the parallel implementation for quadratic regression worked better. Interestingly, the dataset portions with less quadratically shaped graphs (Libra garage, or most Sundays) performed worse for the parallel implementation. In addition to this, we observed that threads working on training the same model would produce very close parameters. The reason for this can be that our model converges quickly enough that after a certain number of epochs, the parameters will start becoming very similar. This can also explain why the sequential model more often performed worse - we were simply iterating over more epochs than needed.

It was also interesting to see that the choice to implement Mini-Batch Stochastic Gradient Descent as our quadratic regression training algorithm was the optimal choice. Through several iterations of training we found that batch size after 10 units made little to no difference in the accuracy of our graphs. Batch sizes smaller than 10 would sometimes cause greater shift in performance for our models. This confirms the efficacy of Mini-Batch SGD in that by computing our model trainings with batches greater than 10 we observer better performance than models trained using SGD (batch of size 1) and with less memory overhead of having to perform computations across the entire dataset for each epoch in gradient descent.

Due to having low overall occupancy most instances with garage Libra did not fit our model well. We also noticed that most garages on Sunday had similar results due to low overall occupancy. Perhaps using polynomial regression would have worked better with these outliers.

REFERENCES

[1] J. B.-B. et al., "Parallelizing machine learning algorithms," *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012.

[2] A. Paszke, "Pytorch: An imperative style, high-performance deep learning library," *33rd Conference on Neural Information Processing Systems*, Dec. 2019.

TABLE I
TIME CONSUMPTION FOR GARAGE B (2-CORE)

Day	Sequential (s)	2-Core Parallel (s)
Sunday	18.604	15.940
Monday	18.748	16.450
Tuesday	18.845	16.446
Wednesday	18.632	16.142
Thursday	18.499	16.243
Friday	18.354	16.046
Saturday	18.643	15.441
AVERAGE	18.332	16.103

TABLE II
TIME CONSUMPTION FOR GARAGE B (4-CORE)

Day	Sequential (s)	4-Core Parallel (s)
Sunday	35.948	30.593
Monday	35.227	30.696
Tuesday	35.501	30.519
Wednesday	35.805	31.300
Thursday	35.833	31.510
Friday	35.693	31.400
Saturday	35.947	29.691
AVERAGE	35.708	30.816

[3] M. Mishra. (2020, May) Best python libraries for machine learning in 2020. [Online]. Available: https://medium.com/analytics-vidhya/best-python-libraries-for-machine-learning-in-2020-top-6-99e2e9e31694

[4] G. Puneet. (2019, June) Speed up your algorithms part 3 - parallelization. [Online]. Available: towardsdatascience.com/speed-up-your-algorithms-part-3-parallelization-4d95c0888748

[5] A. Amidi and S. Amidi. (2018) A detailed example of how to generate your data in parallel with pytorch. [Online]. Available: stanford.edu/ shervine/blog/pytorch-how-to-generate-data-parallel

[6] E. Kim. (2019, Mar.) Dat pre-processing in python: How i learned to love parallelized applies with dask and numba. [Online]. Available: towardsdatascience.com/how-i-learned-to-love-parallelized-applies-with-python-pandas-dask-and-numba-f06b0b367138

[7] J. Schreiber, "pomegranate: Fast and flexible probabilistic modeling in python," *Journal of Machine Learning Research*, pp. 1–6, 2018.

[8] Y. B. et al., "Parsl: Pervasive parallel programming in python," *HPDC '19: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[9] A. Malakhov, "Composable multi-threading for python libraries," *Proceedings of the 15th Python in Science conference*, 2016.

[10] M. Space, "Use joblib to run your python code in parallel," *Medium*, Nov. 2020. [Online]. Available: measurespace.medium.com/use-joblib-to-run-your-python-code-in-parallel-ad82abb26954

[11] D. L. Wang, "Lecture 03. gradient descent and linear regression example," February 2021.

[12] S. Patrikar, "Batch, mini batch & stochastic gradient descent," *Medium, Towards Data Science*, 2019.

VII. APPENDIX

Some challenges we've encountered so far are extracting the data from our dataset in a way that allows us to easily access necessary components, tuning our gradient descent algorithm to fit our needs, and getting some of our group members up to speed with the scope of our project. We've been able to overcome the first challenge and can access parts of the dataset that we need access to. In regards to getting our members all on the same page, some of our members met up to go over the key components of machine learning and what we intend to do with our project. By doing this, all of our members were able to obtain the necessary level of understanding for the project to proceed smoothly. Our gradient descent algorithm was completed , however, we noticed that by limiting ourselves to quadratic regression as opposed to polynomial regression, we severely limited the chance for completely accurate models to predict Garage Occupancy. If we were to restart this project from scratch with the knowledge we have now we would certainly aim to create a polynomial regression model instead. When it came to multiprocess implementation we were mostly happy with the results being consistently close (and sometimes better) than sequentially trained models. This at the very least means we were able to obtain performance that is near to sequential versions of quadratic regression - this is great in that our project could be used in a real world setting to achieve similar performance with work split among many processors. Despite this, we were displeased with the time performance of our concurrently trained 2-core and 4-core models with them being only slightly faster than sequentially trained models. With more time or python familiar coders, it could have been possible to track down the source of this sequential bottleneck.

In summary, we were happy that the few challenges in regards to getting the code process started were quickly cleared. We are also content that little to no obstacles appeared with regards to our research paper besides the initial push to get everyone confident with the material. We are pleased with the fact that our concurrently trained model achieved similar and sometimes better accuracy with the dataset, but, we are not so pleased with the time performance compared to sequential model training.