Distributed Asynchronous SGD

Liamarcia Bifano
liamarcia.bifano@epfl.ch
EPFL

Roman Bachmann
roman.bachmann@epfl.ch
EPFL

Michael Allemann
michael.allemann@epfl.ch
EPFL

Abstract— **This project is part of the course CS-449 "Systems for Data Science" taught as a part of the EPFL Data Science core courses in the academic year 2017-2018. The goal of this project is to implement synchronous and asynchrous distributed stochastic gradient descent (SGD) used for learning support vector machines (SVM). The main reference for this project is the HOGWILD ! [1] paper, whereby the key insight is that SGD can be implemented without any locking, when the associated optimization problem is sparse.**

I. INTRODUCTION

Stochastic Gradient Descent (SGD) is a widely used algorithm in current data science applications and has proven to be well suited for data-intensive machine learning tasks [1]. However as the authors of the Hogwild! paper mention SGDs scalability to a distributed setting is limited by its inherently sequential and iterative nature, which leads to large message overhead for synchronization (or locking). The principle idea of the Hogwild! strategy is to ensure scalability of data analysis by minimizing the overhead of locking, which is done by running SGD in parallel without locks [1].

In the asynchronous distributed SGD that we implement in this project, workers send messages to each other and to the coordinator, which will be written to memory without synchronization, potentially overwriting. The authors of the Hogwild! paper prove, that memory overwrites are rare and that they introduce barely any error into the computation, when they occur [1]. The theoretical guarantees can be studied in the original Hogwild! paper and will not be developed in this paper. The focus of this project lies on the design, implementation and experimentation of synchronous and asynchronous distributed SGD. Before getting to the main part of the paper it must be mentioned that the original Hogwild! paper was implemented on a multicore system with low latency, high throughput and shared memory. In contrast we are using a Kubernetes cluster managed by EPFL (IC cluster) and the nodes don't share the same memory or can be even in different machines however they share a common volume. In the next chapter we will discuss the design choices that were taken and afterwards we move to the implementation and the experiments that were conducted. Finally a short conclusion will tie up what was learned and what could be done to improve the running and the train loss convergence.

II. DESIGN

A. Definitions

Before coming to the actual design choices taken, some terms need to be defined:
Synchronous: Synchronous means that the state of the vector to be learned after each epoch of SGD is the same on all machines.
Epoch: For the synchronous case, one epoch corresponds to one iteration of all workers calculating the gradient of their subset, sending it to the coordinator and receiving the accumulated weight updates.

B. Setup

The first step of both the synchronous and the asynchronous implementations is to setup the cluster, whereby the parameters like number of workers, etc. get set. The coordinator creates the train and validation set split and proceeds to tell to the workers the indices of the validation set.

C. Synchronous Design

1) Synchronous Worker: In the synchronous design the workers randomly subsample the train data, then they calculate the gradient and send the updates to the coordinator, after which they wait for the message of the coordinator containing the collected weight updates of all workers, before subsampling the train data again and continuing.

2) Synchronous Coordinator: The coordinator waits for the weight updates of all workers, when it has received them, it calculates the validation loss and then sends the updates of the current epoch to the workers. Figure 1 shows the high level design choice.

3) Expected performance: On a cluster, message passing is the bottleneck, and our design choice guarantees the minimal amount of messages passed while still guaranteeing synchronicity. We nevertheless expect this design to be fairly slow, because all workers remain idle, while they wait for the slowest worker to terminate its calculations and we do not assume the workers to be of identical hardware.

D. Asynchronous Design

1) Asynchronous Worker: In the asynchronous design the workers randomly subsample the training data, calculate the gradient and then send the updates of the weight vector to all workers and the coordinator. The worker then gets the updates, which arrived while the gradient was being calculated, updates his own weight vector, subsamples the training data again and continues.

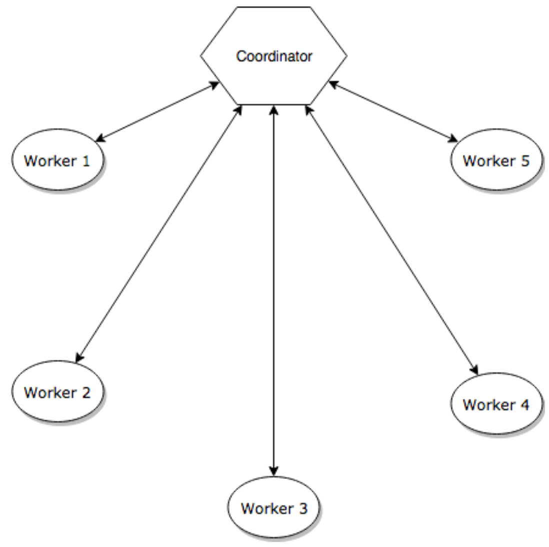

Fig. 1. Synchronous design

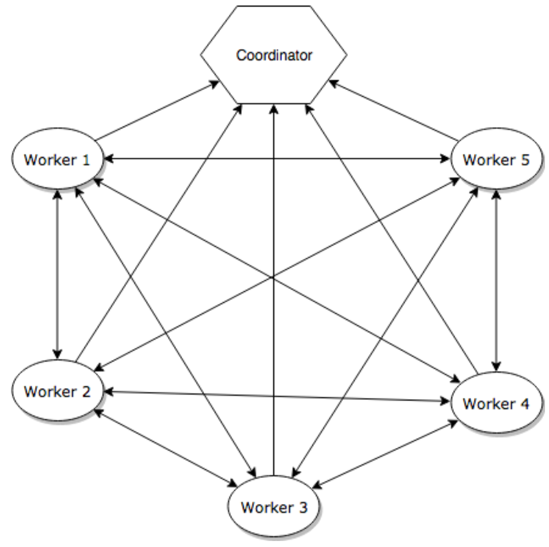

Fig. 2. Asynchronous design

2) Asynchronous Coordinator: In the asynchronous design the coordinator collects the updates of the workers and after a specified number of received updates calculates the validation loss. Figure 2 shows the high level design choice for the asynchronous version.

3) Expected performance: This time quadratically many messages get passed, but the workers can calculate iteration after iteration without waiting for a signal from the coordinator, which we expect to significantly speedup the calculation.

III. IMPLEMENTATION OF DISTRIBUTED SGD

A. Reuters RCV1 Dataset

The dataset used to evaluate our implementation is the *Reuters RCV1*, which contains pre-classified English news. The train contains 23.149 and the test 781.265 rows, the maximum number of columns that a row can have is equal to 47.237 however just 77 of them are filled on average which turns it into a very sparse problem. [1]

B. Support Vector Machine (SVM)

The machine learning algorithm that is optimized using SGD is the Support Vector Machine in our case. We will use the SVM in a supervised binary classification task of detecting whether or not a given Reuters news article belongs to the 'CCAT' class. The SVM will try to find the maximum-margin hyper plane dividing the group of points labeled -1 and 1. The associated soft-margin loss is

$$\mathcal{L}(w) = \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i(wx_i - b)) + \lambda||w||^2 \tag{1}$$

where y_i is the label of data point x_i, w is the vector of weights and b the bias. The bias is removed from the formula by adding a constant new dimension of ones to the data. A regularization factor λ is chosen to be 10^{-5} allowing the SVM to find a solution even in non linearly separable data sets.

1) Data handling: Every worker is reading and parsing the whole data set. Since the data sets are stored in a text file, we have to read the whole file in any case, so every node will sample from the full training set. For calculating the test accuracy, only the coordinator will read the large test set once in the end of the SGD calculations.

2) Loss calculation: The coordinator is in charge of calculating the validation loss. In the synchronous case, it calculates this loss after every epoch. In the asynchronous case however, we wait until the coordinator got approximately as many updates as in the synchronous case before we calculate the loss. All the workers are responsible for sending the training loss over their sub-sample to the coordinator with every calculated gradient. The training loss is highly stochastic and is not as stable as the validation loss.

For proper comparison between the validation and training losses, we always consider the scaled down loss per data point.

3) Learning Rate: The baseline learning rate for the SVM is set to $\eta = \frac{0.03}{|W|}$, where $|W|$ is the number of worker nodes. We scale the learning rate inversely with the number of workers, such that when accumulating the subsampled gradient updates from many workers, we do not just add them all up in total and walk too far. The value of 0.03 with subset size 100 has been found to work well during the implementation of the system because it was observed that values bigger than that often lead to divergence in the validation loss.

4) Stopping Criterion: To decide when to stop the SGD algorithm, we opted to using a stopping criterion where we check if the validation loss stops improving. We implemented this by having a sliding window over the last 20 (persistence) validation losses and if the loss did not decreased in the meantime, we assume convergence and stop the training. If

[1]Less than 1% of the input matrix is filled.

the stopping criterion is not reached, 2000 epochs will be run.

C. Communication between nodes using gRPC

For sending messages between all the workers, we use gRPC on top of Protobuff. Each worker first creates a gRPC server and adds a `HogwildServicer` class on top which is responsible for handling incoming messages. The `HogwildServicer` keeps information about certain flags and locks needed for the synchronous and asynchronous implementation. It also stores a buffer of incoming weight updates that will be used by the coordinator and workers to update their own SVM weights.

Workers communicate with the servers of other workers by first opening a gRPC channel and then creating a client stub on that channel. gRPC messages can then be created and sent to a worker using their corresponding stub. Sent messages must always return an answer message. In our use case however, we do not require a response from other workers, which is why we send an empty answer.

D. Multiprocessing and Multithreading

To take advantage of the fact that modern CPUs usually have multiple cores, we decided to run the listener process parallel to the SVM process that calculates the weight updates. During execution, the coordinator and workers spawn the SVM process using the Python `multiprocessing` library and pass a thread safe task queue and response queue to it. The task queue is used by the main coordinator and worker loops to tell the SVM processes to calculate the gradient, update the weights and calculate the loss. The response queue is used by the SVM process to return gradients or losses to the parent process. When trying to pop from an empty queue, the process will wait until an element arrives.

IV. Deployment on the Cluster

In order to run the code in the Kubernetes cluster, two different *services* and *statefulsets* will be created, one for the workers and another for the coordinator. The *Dockerfile* for both types of machines will be the same so the difference between them will be the command used when the container is started and it will be specified in the *yaml* files to spin the machines and services in Kubernetes. *Statefulsets* were considered to make the *pods / containers* talk to each other through fixed host names that are more stable compared to IP addresses that can change once some of the *pods* are restarted. Variables like the number of workers, synchronous and asynchronous mode and the data path will be passed as environment variables to the containers and those variable will be gathered from the *ConfigMap*.

To run the job in Kubernetes these main steps will be considered:

1) Check if there are some remaining machines in the cluster and shut them down
2) Build the image and push it to *Docker Hub*
3) Start the workers service and all pods replicas in parallel
4) Once all machines in the worker service are up and listening, start the coordinator
5) The coordinator will send the message to start the job to all the workers
6) Once the job is done, the coordinator will write a log file inside its container
7) The log file will be synced with the local machine that started the process
8) Shut down all the infra (workers and coordinator)

V. Experiments

It is important to evaluate the running time and that the validation loss converges in order to analyze and compare our implementation. Three main experiments were run testing our implementation. The first one evaluates the timing performance for synchronous and asynchronous over a different set of workers. Secondly, the loss convergence for each version (synchronous and asynchronous) will be evaluated over time. Lastly, the subset size will be tested for the following subset $\{1, 10, 50, 100, 500, 1000\}$.

All experiments were run in the IC cluster and it was setup with the same amount of memory for each of the containers to not consume all the memory available in the cluster and also in order to make the trials comparable. For the workers we setup $4Gi$ and for the coordinator $20Gi$. The reason for this high memory for the coordinator is to calculate the final accuracy in the test set that is about $14GB$ uncompressed.

A. Number of Workers

The execution and accuracy performance for different number of workers $\{1, 2, 5, 10, 15, 20, 23\}$ were collected for synchronous and asynchronous implementation. In the Figure 3 it is possible to observe that the execution time for synchronous is increasing very fast with the number of workers [2] because all workers have to wait until all updates reach the coordinator to proceed in each epoch so the execution time will be always at least the execution time of the slowest worker. On the other hand, the execution time of asynchronous remains almost the same and tends to decrease as long as the number of workers increases (at least it is expected taking in consideration that the train loss will converge faster due to the number of workers as can be seen in 4).

Even though the number of messages passed over the network for asynchronous is quadratically higher than in the synchronous case because all workers send their updates to everyone, it doesn't hurt the execution performance. However this scenario might be different if the machines are not in the same geographical region because the messages' travel time can take to much to reach others workers thus the loss could take longer to converge and then increase the total running time.

[2]If it is assumed that the distribution of time execution is exponential then the maximum will have an exponential distribution as well however it is hard to infer it from the curve because there are not enough data points.

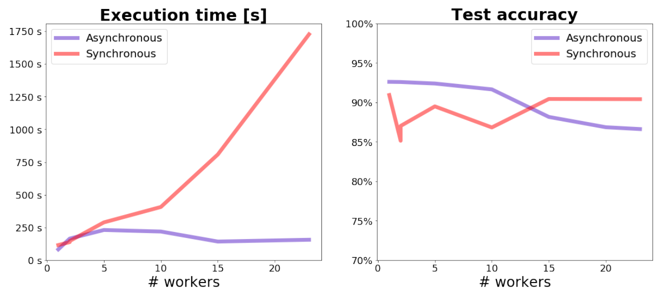

Fig. 3. Execution and accuracy performance by the # Workers

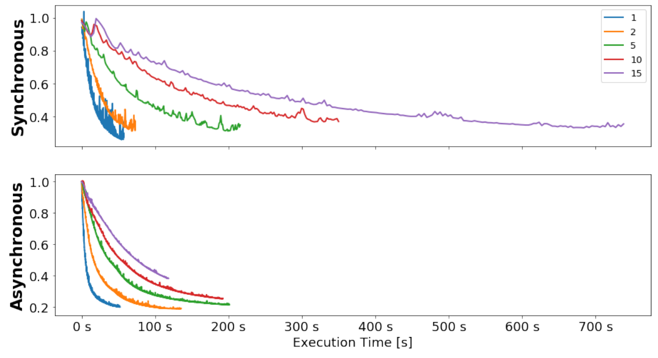

Fig. 4. Validation Loss Convergence synchronous and asynchronous by the number of workers

Regarding the test accuracy it is not possible to assume that both curves are statistically different or that synchronous gives better results than asynchronous because each trial was run only once and the algorithm is not deterministic. Besides that, just one trial was run for each number of workers, due to time constraints and a sometimes seemingly overloaded cluster.

The performance was just tested for a small set of workers because for a number of workers greater than 25, some of the containers were having difficulties to setup and be available to execute the job in the cluster. In addition, to start all machines and get all machines down in the cluster took about 10 minutes which made executing more trials difficult.

B. Validation Loss Convergence

Another important aspect to be analyzed is the validation loss convergence and in Figure 4 the losses are shown for both implementations and for a different number of workers over time.

It can be seen that the convergence time for asynchronous case is faster when compared with the synchronous implementation however it tends to be even faster when the number of workers increases. Even though some workers process faster than others and can be stuck for a while, the coordinator is responsible for evaluating the overall convergence and then eventually the fastest workers will receive the updates from their partners.

C. Changing the subset size

For measuring the effect that the sample size has on the convergence, we conducted an experiment with 10 workers

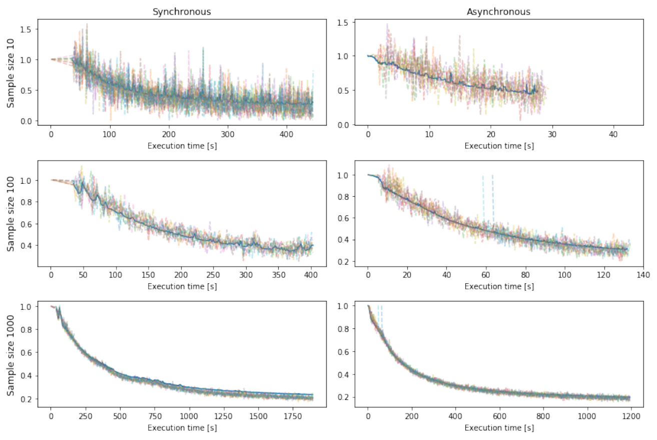

Fig. 5. Comparison of the validation loss and the training loss (dashed) of all 10 workers in synchronous and asynchronous execution with sample sizes 10, 100 and 1000.

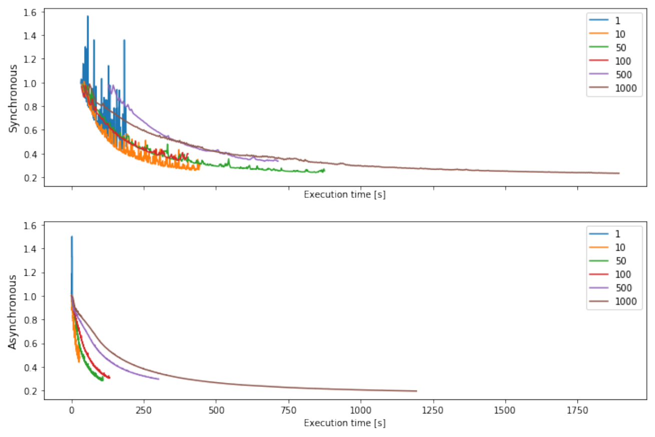

Fig. 6. Comparison of the validation losses using different subset sizes.

and sample sizes 1, 10, 50, 100 500 and 1000. As for each round of updates, we are receiving more or fewer gradient calculations from each worker depending on the subset size, we scaled the learning rate accordingly. Our baseline learning rate of 0.03 with baseline sample size 100 will be scaled by $\frac{100}{|S|}$, where $|S|$ is the sample size. If we did not adapt the learning rate to the subset size, we are running into the risk of the loss not converging.

Figure 5 highlights the validation loss and the 10 training losses (dashed lines) for three selected subset sizes of 10, 100 and 1000. We clearly observe that the fluctuations in the training losses get more stable with increased subset size.

Figure 6 shows a comparison of the validation losses between many different subset sizes and synchronous and asynchronous modes. The asynchronous computations are taking much less time than their synchronous counterparts. In the synchronous case, all sizes reached similar performance before the early stopping criterion activated. In the asynchronous case however, only the experiment using a subset size of 1000 reached a very low validation loss.

Looking at the execution time and test set accuracy as a function of the chosen subset size in Figure 7, we notice

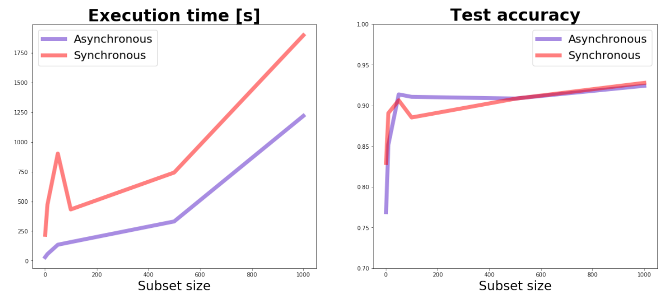

Fig. 7. Execution time and test set accuracy as a function of the subset size.

that the execution time increases almost linearly with the subset size. This might stem from the fact that each node has to calculate more gradient elements in each epoch, while at the same time the adapted learning rate is keeping the convergence at the same speed. We also see that the accuracy on the test set is always very similar and only very unstable for subset size 1.

From this experiment, we take away that smaller subset sizes are converging faster but less stable and that asynchronous versions are usually taking less than half the time to reach the same loss as their synchronous counterparts. Because of the stochastic nature of taking smaller batches, we have to increase the early stopping persistence for those cases.

VI. Conclusion

In this project we learned that for SGD on a distributed system the strong notion of synchronicity, which doesn't scale well, can be relaxed, while still resulting in similar accuracy. The asynchronous version, which we implemented, performed significantly better in the convergence time than its synchronous counterpart, while reaching comparable test losses and accuracies.

Throughout this project we learned to handle the complexity of communication between multiple computers and deal with a large data set while having to investigate closely the performance and execution times of the code running on the cluster.

For future work, it would be interesting to experiment with different optimization algorithms in a distributed and asynchronous environment or implement this project using multiple GPUs and try recent proposals [2] like increasing the batch size instead of decaying the learning rate.

References

[1] Benjamin; Re Christopher; Wright Stephen J. Niu, Feng; Recht. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. eprint arXiv:1106.5730, 2011.

[2] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.