Distributed Asynchronous SGD

Timon Zimmermann, Thibault Viglino, Bernat Esquirol

May 28, 2018

1 Introduction

This project implements a distributed, synchronous or asynchronous, computation of the stochastic gradient descent to optimize a support vector machine. There were two main parts to our work. First we had to overhaul our milestone 1's synchronous code to make it more efficient and be able to work with the RCV1 dataset. Then we had to implement the asynchronous version using insight from the *Hogwild!* paper. Special care was given to our ad hoc sparse matrix implementation and to the corresponding modification of our SVM implementation.

2 Design and Implementation

In order to implement the given problem we decided the best approach was to create one coordinator (`sgd_server.py`) and multiple workers (`sgd_client.py`). The number of workers is decided at the beginning by the user. Each worker trains on a subset of the whole dataset given at random by the server. When the computation of the SGD (`svm_sparse.py`) by the client ends, the weights are then sent back to the server which modifies its own global weight vector. The new vector is then sent back to the workers so that they can continue their work. In the following subsections we develop more on how we accomplish each part. Note that unless specified, both the sync and async version use the same code and thus act the same way. All the configuration, such as number of workers, learning rate, etc. are set in a configuration file.

2.1 Deployment

The algorithm deployment is done through Docker containers deployed on a Kubernetes cluster. This way, we have a fully system agnostic deployment pipeline which is very convenient to encourage ease of installation and reproducibility. To this end, we create a Kubernetes configuration file with a StatefulSet of *num_workers+1* replicas (the workers plus the coordinator) and we also launch a Kubernetes Service to allow networking and communication between all the StatefulSet clients. The Kubernetes pods created are deploying a Docker image which runs the client script for the workers and the server script for the coordinator.

2.2 Reading and sending the data

At initialization, the server reads the data files and transforms it into our custom made sparse matrix format. We used the CSR format that consists of three lists: `A, IA, JA`. `A` contains the non-zero values, `IA` is recursively defined as the number of non zero value on a row plus its preceding ones. `JA` gives for each element of `A` its column. This implementation allows us to do fast computation on very sparse matrices.

When the data is loaded on the server, we sent a part of it to each client. The part of the dataset sent to each client is each time randomly selected (we had to create a random row getter function on our sparse matrices). The size of the part is $size\_of\_dataset/(2 \cdot number\_of\_workers)$. This was chosen experimentally to reduce the transfer time to the clients, while maintaining accuracy and convergence time.

Here are the other parameters sent to each workers at initialization:

- w: the initial weight, a vector of zeros by default
- *eta*: the learning rate, fixed by default

- *lambda_*: the regularization parameter
- *iterations*: the number of SGD iteration that the worker needs to do before sending its new weights
- *gradient_thresh*: the threshold above which a gradient value is considered meaningful

2.3 Communication

Here is a summary of the exchange between the different actors during runtime:

- A worker will do stochastic gradient descent on its dataset during the defined number of iterations.
- The worker sends back the last computed gradient, but just the part of it where it was above a certain threshold (another hyper-parameter defined at launch).
- The server updates its weights. It is there that we had the most difference between the synchronous and asynchronous parts. We break down the difference in the following.

  Synchronous: We had to modify the way it was working in milestone 1 in order to make it compatible with the rest of the asynchronous code. Whenever a worker yields back its new weights, the server adds it to a weight list and puts the worker on hold (by not replying right away). When it has all the weights, it computes the mean an makes it the new weight. Then it is sent back to the workers that do a new iteration.

  Asynchronous: In this case, we use the *Hogwild!* approach, i.e. only the changed weights are updated asynchronously from the gradient coming from all workers. Numpy arrays being thread safe on insertion, we do not have to fear concurrency issues. The workers don't wait for the server to compute loss or anything and continue to work right away with the new weights.

This whole process goes on until convergence, as explained in section 2.4.

2.4 Convergence

We used the accuracy as a measure of the state of convergence of our system. It is computed at fixed time steps (of 2 seconds) in order to give meaningful comparison between the different methods. We set a threshold of *90%* (modifiable in the configuration file), above which the server state will change to "shutdown". When the server is in this state, whenever a worker tries to ask for the new weights, the server will send it the additional information that it has to stop. The server keeps track of which worker has stopped. When all the workers have stopped, the server will exit, returning the final weights.

3 Experiments

3.1 Preprocessing

As a preprocessing step, we've added a bias feature with value 1 to each sample as is common place with SVM. We did not find it to improve the result though, neither in convergence time nor in accuracy. The data where already normalized so we did not have to scale them.

3.2 Hyper-parameters

3.2.1 Learning rate

We tried a few powers of ten for the learning rate. We ended up using 10^{-3}. The problem with a bigger learning rate is that the loss wouldn't go down. With smaller learning rate on the other hand, the convergence was way too slow.

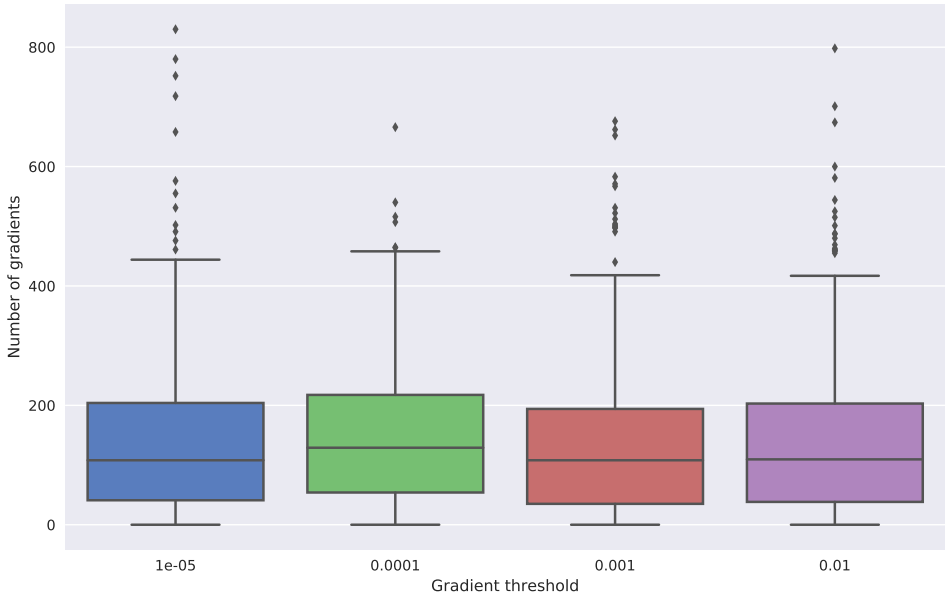

Figure 1: Number of gradient values above threshold.

3.2.2 Iterations

The number of iterations that a worker compute before sending its weight was hard to fine tune. We wanted to have a very fast rate, since it is how it is suggested in the *Hogwild!* paper. A big iteration number would also mean that most of the gradient would change and we would lose the benefit of sending only a few values over the network, also, we'd like the worker to be quickly updated on new weights. The difference between our work and *Hogwild!* resides in the fact that sending new gradients over the network, in an asynchronous context, has some overhead that is not present in a purely multithreaded environment. Thus we set this parameters to 10 and got good results.

3.2.3 Regularization

Our regularization parameter was set to zero to achieve best performances. We believe that this is due to the fact there is a low chance of over fitting in such a big dataset.

3.2.4 Gradient threshold

This parameter controls how big an element of the gradient vector must be in order to be transmitted over to the server. Our initial idea was that the gradient could have very small changes on some weights and that it would be wise not to send them over on the network. In practice though, as we can see in figure 1, it didn't change much. The extreme sparsity of the dataset probably made so that the only none zero gradient values were big and thus the threshold had no impact.

3.3 Round trips

Figure 2 shows the round trip time (RTT) that a worker experiences when sending its new weights to the server and getting the newly updated weights. As expected, in the synchronous case, the more workers we have, the more time the server has to wait on average, since it waits for all workers to answer. We observe that in the asynchronous case this mean RTT time still grows, but at a much slower pace. This should be related to the fact that the server is busy handling other requests and weight updates, but in a non-blocking fashion.

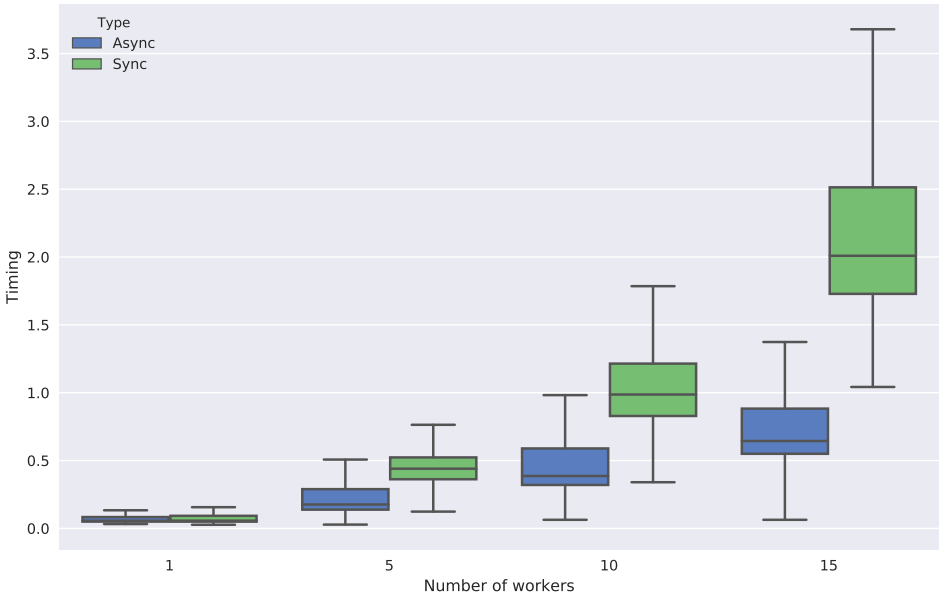

Figure 2: The y axis shows the return trip time between the moment when a workers sends its computed weights and the moment when it gets new weights from the server.

3.4 Convergence time

If we compare figure 3 and figure 4 we can see how even though the asynchronous version takes less steps to reach *0.7* accuracy independently of the number of workers, it converges slower than the synchronous one. That could be caused by the fact that we compute the mean of the weights instead of replacing them in some cases, this effect might occur in the asynchronous part when we have collisions.

4 Conclusion

The *Hogwild!* paper has guided us in the implementation of the SGD, but we needed to address the problem of abstracting to a distributed system when the paper is the multi-core. This project shows how the parallelized implementation of SGD doesn't always improve the overall timing, it mainly depends on the number of workers we have running.

A possible future work would be to compare the two approaches in the extreme case where we have more than 100 clients, we think that the asynchronous algorithm would scale much better and outperform the other one.

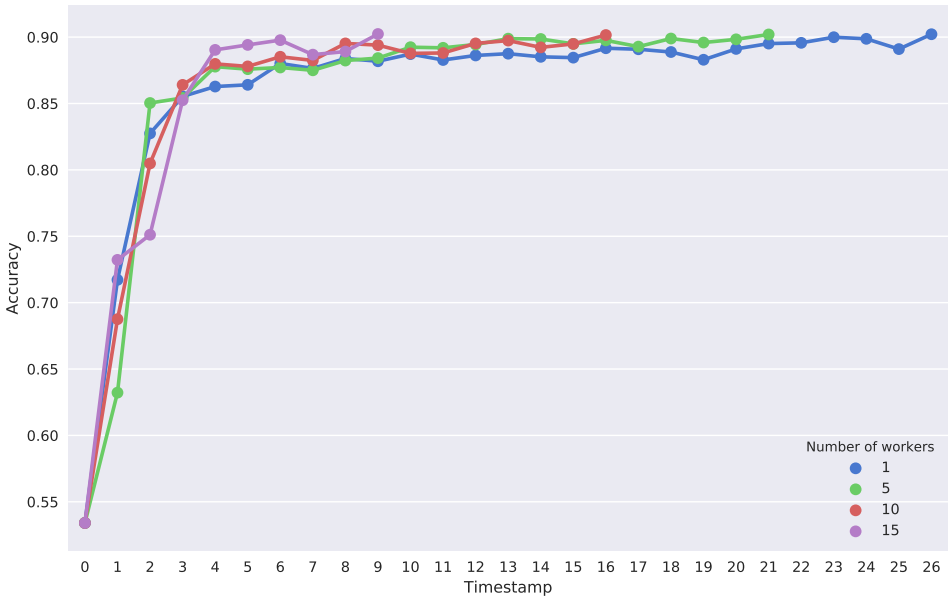

Figure 3: Accuracy over time in the synchronous case.

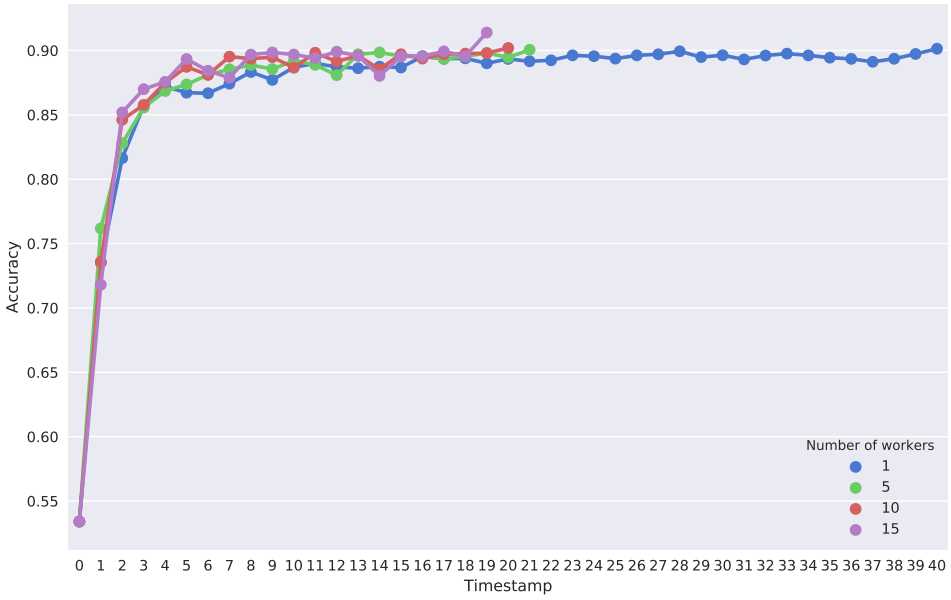

Figure 4: Accuracy over time in the asynchronous case.