Distributed Synchronous and Asynchronous SGD (CS-449)

Romain Choukroun (203917), Augustin Prado (237289), Brune Bastide (23967)

***Abstract*— A single-threaded version of SGD is often not fast enough. There are many ways to implement a multi-threaded version of SGD. However, it is not very straightforward to implement an efficient parallel version of SGD due to its inherently sequential nature. We used HOGWILD ! paper [1] as an inspiration to obtain an efficient and fast solution to this problem.**

I. Machine Learning

A. Support Vector Machine

First, we label the dataset by looking whether each sample contains the word "CCAT" or not, assigning -1 or $+1$ to each. The Support Vector Machine (SVM) algorithm tries to find the best hyperplane that separates our data points by maximising the margin –that is the distance between the hyperplane itself to hyperplane's closest data point.

We have implemented in python a stochastic gradient descent for this binary classification problem. For this aim, we have used the general SVM loss function :

$$L(\mathbf{w}) = \sum_{i=1}^{N} l(y_n \mathbf{x}_n^T \mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||_2 = \sum_{n=1}^{N} L_n(\mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||_2 \tag{1}$$

where $l : \mathbb{R} \mapsto \mathbb{R}, l(z) = \max\{0, 1-z\}$ is the hinge loss function, $\mathbf{x}_n \in \mathbb{R}^D$ is the n-th data example $y_n \in \{0, 1\}$ the n-th output and $\lambda \in \mathbb{R}^+$ the regularization term. In our case, $D = 47'236$ and $N = 23'149$. Instead of using the function in (1), we use at each update the loss $L_n(\mathbf{w})$ which is the cost contributed by the n-th example. We also implemented a mini-batch gradient descent to accelerate the convergence of the algorithm, where each update consider the cost contributed by multiple samples, where the number of samples, or batch size, has to be fixed. Moreover, given that the loss function is not derivable when $y_n \mathbf{x}_n^T \mathbf{w} = 1$, we use a subgradient for the learning part, instead of a classical gradient.

The loss is computed with `calculate_loss` in `svm_function.py` file. The subgradient is computed in function `mini_batch_update` where you can choose to do stochastic gradient descent or mini-batch gradient descent.

1) Learning rate & stopping criteria:

We optimize the learning rate γ in the grid search section and finally find $\gamma = 0.05$ as the parameter that reach best accuracy on the training set. We stop the learning process when the condition $0 < |L(\mathbf{w}_n) - L(\mathbf{w}_{n+1})| < \epsilon$ is satisfied, where the stopping criteria ϵ is computed through the theoretical equation coming from [1] simplified as follow :

$$\epsilon = \frac{2L^* M^2 \Omega}{\phi c}\gamma \tag{2}$$

where the following parameters are defined with respect to the loss function $L(.)$: c is the strongly convex modulus, L^* the Lipschitz constant, M the constant that bound the gradient. Finally, $\Omega = \max_{e \in E} |e|$, with E the set of hyperedges of the *hypergraph* corresponding to our dataset, is a parameter that characterize sparsity of our data and $\phi \in (0, 1)$. In practice $\Omega = 1$ and other constant where estimated manually.

II. Systems decisions

HOGWILD ! can be implemented on a single laptop using multiple CPUs. However, when we are training over a huge dataset with sparse data, why not leverage the power of a whole cluster of nodes ? The IC faculty at EPFL provided us with such a cluster and we explain in the following section how the components connect together as well as differences between our synchronous and asynchronous implementations.

A. Docker/Kubernetes

Kubernetes is a standard choice for large scale deployment, it is able to distribute a task along multiple nodes in a cluster. A few problems arise compared to the laptop version of the program, mainly the communication in between different nodes. Leveraging Kubernetes' services is a great way to solve this issue.

Let us discuss exactly how the resources are allocated on Kubernetes before getting into the nitty-gritty details of the implementation. We have two docker images stored on EPFL's docker images registry, one for the server and the other for the different clients, both containing only the code necessary. There is not much to say about the images, as they are simple python-slim images containing the different python packages needed to run the program (client or server).

From these two images, we created two StatefulSets in Kubernetes, the idea behind StatefulSets was to have a layer of consistency regarding the names of each client, and of the server, but also across the different DNS names. Another highly appreciated feature of the StatefulSet is the persistent link in between itself and the persistent volume associated (the dataset in our particular case). The server is the only StatefulSet claiming access to the dataset, and of course, it is replicated only once, we do not want two servers running in parallel. Considering the size of the dataset, we gave the server access to a minimum/maximum of 2Gi/20Gi of memory, and also as many CPUs as it has clients in order to treat the requests in parallel. The clients however do not have access to the data (yet), and can be replicated as many times as we need to run the experiment. Each client has access to 1 CPU and a minimum/maximum of 4Mi/4Gi of memory, which will depend on how many clients we have running in total. Setup-wise, we are almost done. The last crucial thing to do is to enable the different StatefulSets to communicate together. To do so, we setup a service which opens a port with which all clients can contact the server, and now the setup is up and running !

B. *Workflow*

We now take a closer look at exactly **how** the program runs. Please see appendix 1 to get an overview of both the synchronous and asynchronous versions. For the purpose of simplicity, we will reduce the system to a server and three clients, abstracting how the clients/server are run on Kubernetes.

C. *Starting the training*

First, the server and clients will start by loading the different config variables such as learning rate, batch size, etc... from the environment. The server will then load the training set, which is fast, and then spawn a detached thread running in the background which will load the testing set while the server continues to work seamlessly. All of the weights used by the algorithm will be initialized to zero, and same for the gradients. Once all of this is done, the server will await the client connections.

The clients will also initialize their gradients and weights to zero, then they will try to authenticate to the server. If the server is not up, they will keep trying up until the server is up. The authentication system will not allow a connection from a client if the server's pool of clients is full. Meaning that if the server knows that it should accept a maximum of 10 clients, any client trying to connect when this pool is full will be rejected. When all clients are connected, they ask the server for a chunk of the training set. Each client will receive a number of training samples equal to the total number of training samples divided by the number of clients. When a client received all of its data, it will divide it into a training set and a validation set, according to the given configuration. Past this point, all of the clients are ready to start computing gradient updates, however they will all wait for the server's background process to load all of the testing set.

Now that the testing set is loaded server-side, the clients can start their computation with their own training and validation sets. They will load a batch of training data, the size of which is defined by the configuration, and proceed to compute a gradient update. When a gradient update has been computed, each client sends only the different gradients than previously to the server, and after that we have two different modes, either the synchronous one or the asynchronous one ; let us discuss both :

Synchronous version: In the synchronous case, the server awaits the gradient updates of all clients while summing them up. All clients only send one gradient update, and then wait for the server's final gradient update response. When the server received all of the updates, it sends back to all clients the summed up gradients, and updates its weights according to the learning rate given in the configuration file. The clients, having received the total gradient update will also update their weights using the learning rate.

Asynchronous version: Running the updates asynchronously allows us to really speed up the computation. Knowing the fact that the data is highly sparse, the server will await again all the clients' updates to create a full gradient update, it continues to sum them up until all updates have been received, with one catch. The catch is that the clients will not wait before sending another update. In fact they will continue to send gradient updates, and when the server answers with a full gradient update, they will update their weights using the learning rate. On the server-side, if it receives two updates from the same client, it aggregate them, and if there is a collision on the gradient of a same feature, the server will only keep the latest version. When the server sends back the global gradient update, it will also update its own weights. Preventing the clients from "wasting time" is what speeds up the computation so much. However, the gradient updates will be of a lesser quality since the clients will compute multiple updates with the same weights.

D. *Ending the training*

In both cases (synchronous/asynchronous), the gradient's accumulator is locked using a simple exclusive lock, so as to not have two client updates provoking a race condition over it. We will run a certain number of epochs over the dataset determined by the stopping criteria explained in the first section of the report. When we reach the stopping criteria, in the synchronous case all workers will stop working altogether, and disconnect from the server gracefully. However, in the synchronous side, one worker might be done with its computations, but not the others, which means that we needed to implement a stopping system. The system is quite simple : when a client is done, it tells the server and then disconnects from it. This way, the server keeps track of the number of connected clients. Which means that when a client disconnects, on the next update, the server will wait for one less client before updating its weights. This way, we will not have any problem regarding client disconnections.

When all clients have disconnected, the server will use its weights to compute the accuracy on the testing set, show it to the user, and go to sleep until the experiment is terminated by the launching script.

III. Experiments

We gathered in this section our final results concerning our hyper-parameter tuning, speed-up convergence analysis and asynchronous/synchronous comparison.

A. *Grid search*

To find our best parameters, we decided to use synchronous version, because it is supposed to be more precise in the convergence compared to asynchronous. Each time, the stopping criteria ϵ is tuned with respect to the learning rate γ using our linear equation in (2).

S (batch size)	γ (learning rate)	Accuracy	Number of it.
16	0.01	90.1	22
-	0.005	91.9	29
-	0.001	90.3	34

TABLE I

LEARNING RATE OPTIMIZATION

For learning rate optimization (Table 1), we observed a maximum accuracy at $\gamma = 0.005$ of 91.9% , along with a number of iterations increasing when the learning rate γ decrease, that is coherent with the idea that each "jump" are smaller.

For batch size optimization (Table 2), we reach a best accuracy for $S = 16$ and $\gamma = 0.005$ and 92.3%.

S (batch size)	γ (learning rate)	Accuracy	Number of it.
1	0.005	73	30
2	-	82	29
8	-	91	30
16	-	92.3	30
32	-	89	31

TABLE II

BATCH SIZE OPTIMIZATION

Finally, we will use these best paramaters later on, $S = 16$ and $\gamma = 0.005$.

B. *Speed vs. nb of workers*

In figure 1, we see a speed of convergence approximately 100 sec. faster for asynchronous. Also, in both synchronous and asynchronous designs, we experience a smililar pattern. Time to convergence improves with the number of workers until 6 workers and stagnate after 6, reaching a threshold of time optimization.

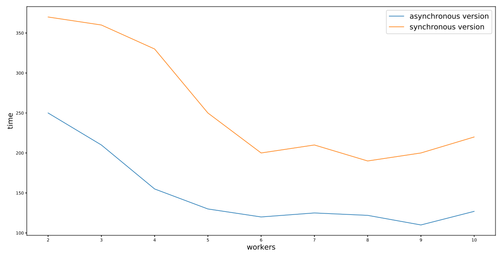


Fig. 1. Workers (x-axis) versus time (y-axis).

C. *Iteration vs. Loss (Synchronous and Asynchronous)*

As we can see from Figure 2, in the synchronous framework, the validation loss converges to a constant value in a smooth manner and the training loss tends to zero.

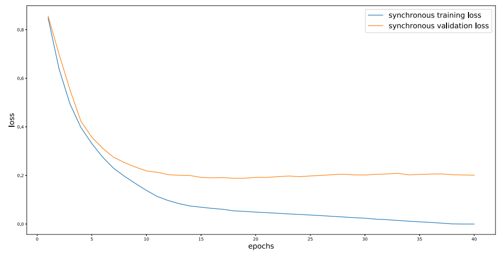


Fig. 2. Epochs (x-axis) versus Loss (y-axis).

However, when we compare to the asynchronous plot, we can see the difference in the number needed of epochs to converge, while each epoch in the asynchronous mode will be faster.

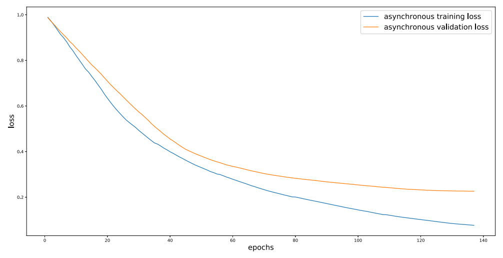


Fig. 3. Epochs (x-axis) versus Loss (y-axis).

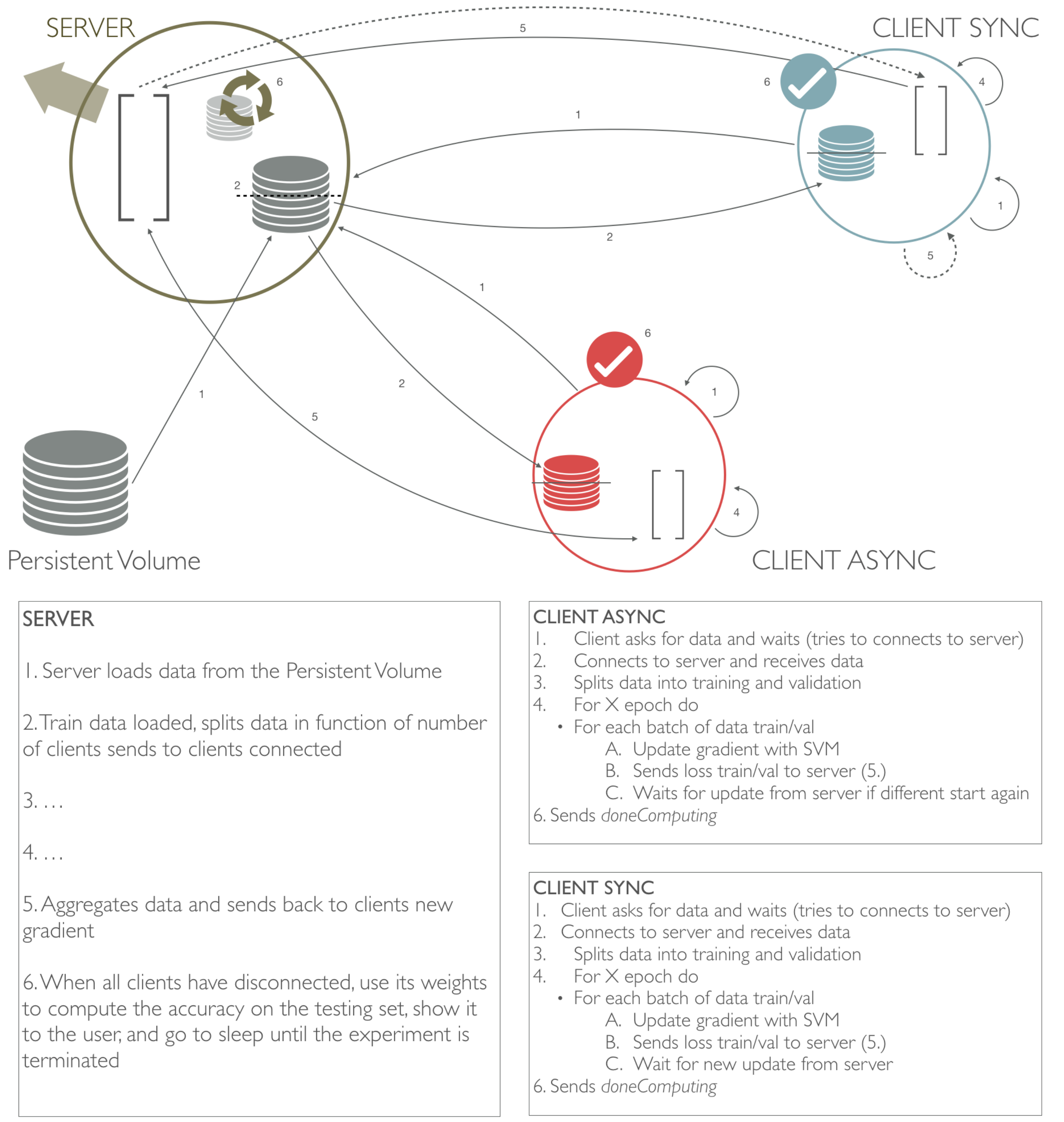
Attachments and deliverables

`run_kub.sh`, `del_kub.sh`, `logging_interface.sh`, `config_file`, `README.md`, `requirements.txt`, `sgd_svm_server.py`, `sgd_svm_client.py`, `sgd_svm.proto`, `data.py`, `sgd_svm_pb2.py`, `sgd_svm_pb2_grpc.py`, `svm_function.py`, `pod-server.yaml`, `Dockerfile-client`, `Dockerfile-server`.

References

[1] Feng Niu, Benjamin Recht, Christopher Re and Stephen J. Wright *Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent*. New Jersey : Prentice Hall.

[2] Robert Burbidge *Stopping Criteria for SVMs* Statistics Section, Imperial College.



SERVER

1. Server loads data from the Persistent Volume

2. Train data loaded, splits data in function of number of clients sends to clients connected

3. . . .

4. . . .

5. Aggregates data and sends back to clients new gradient

6. When all clients have disconnected, use its weights to compute the accuracy on the testing set, show it to the user, and go to sleep until the experiment is terminated

CLIENT ASYNC

1. Client asks for data and waits (tries to connects to server)
2. Connects to server and receives data
3. Splits data into training and validation
4. For X epoch do
   - For each batch of data train/val
     - A. Update gradient with SVM
     - B. Sends loss train/val to server (5.)
     - C. Waits for update from server if different start again

6. Sends *doneComputing*

CLIENT SYNC

1. Client asks for data and waits (tries to connects to server)
2. Connects to server and receives data
3. Splits data into training and validation
4. For X epoch do
   - For each batch of data train/val
     - A. Update gradient with SVM
     - B. Sends loss train/val to server (5.)
     - C. Wait for new update from server

6. Sends *doneComputing*

Figure 1: Workflow schema in both synchronous and asynchronous environments (animated version will be presented in live).