Handling Heterogeneous Clients in Federated Learning
FedHybrid Reproducibility Study

Antoine Bergerault, Felicia Liu, Sophia Zhang

May 6, 2023

1 Introduction and Motivation

Federated learning is a trending research area where we divide training between multiple devices possessing their own local datasets. It is a promising solution to overcome data security and privacy concerns. When communicating, clients involved in the training are not explicitly sharing their dataset information, but rather the result of their local updates on the trained parameters. By aggregating all the updates together, a central server would be able to leverage all these local updates and derive global consensus parameters.

Many past studies have gone over the case of homogeneous clients [2], where classical first order or second order updates have been proved to exhibit different convergence guarantees. Oftentimes, clients within the network do not have the same computational abilities leading to wasted resources. The FedHybrid paper [3] came up with a solution to perform federated learning in these heterogeneous settings. In this study, we summarize how their method works, and show how we were able to implement it using Python in order to reproduce their results.

2 Problem setup and FedHybrid contribution

The goal of FedHybrid is to solve an unconstrained minimization problem using federated learning. To make it possible, only decomposable cost functions of the form $f(x) = \sum_{i=1}^{n} f(x)$ are considered. In addition to that, we need to add a constraint that will enforce a consensus between the clients and the server. If we denote x_0 the server consensus, and x_i the clients primal variables, we get the following consensus constraint, where $W = (1_n, -I_n) \otimes I_d \in \mathbb{R}^{nd \times (n+1)d}$ is the incidence matrix:

$$\underbrace{\begin{pmatrix} I_d & -I_d & 0 & \cdots & 0 \\ I_d & 0 & -I_d & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ I_d & \vdots & \vdots & \cdots & -I_d \end{pmatrix}}_{\doteq W} \underbrace{\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}}_{\doteq x} = 0 \tag{1}$$

This enables us to derive a new constrained optimization problem:

$$\min_x \sum_{i=1}^{n} f_i(x_i) \quad \text{s.t.} \quad Wx = 0 \tag{2}$$

Augmented Lagrangian

In order to derive the updates used by the clients, the problem is restated by the mean of an augmented Lagrangian [1]. We define it as follows, where $\hat{\lambda}$ represents the estimated Lagrange multipliers and $c(x)$ the consensus constraint:

$$\mathcal{L}_\mu(x,\hat{\lambda}) = f(x) + \hat{\lambda}^T c(x) + \frac{\mu}{2}\|c(x)\|_2^2 \tag{3}$$

Applied to our problem where $c(x) = Wx$:

$$\mathcal{L}_\mu(x,\hat{\lambda}) = f(x) + \hat{\lambda}^T Wx + \frac{\mu}{2}x^T W^T Wx \tag{4}$$

$$\nabla_x\mathcal{L}_\mu(x,\hat{\lambda}) = \nabla f(x) + W^T\left[\hat{\lambda} + \mu Wx\right] \tag{5}$$

$$\nabla_{xx}^2\mathcal{L}_\mu(x,\hat{\lambda}) = \nabla^2 f(x) + \mu W^T W \tag{6}$$

Dual function

We define below the dual function that we will utilize for the two types of dual updates:

$$g(\lambda) = \min_x \mathcal{L}_\mu(x,\lambda) = \mathcal{L}_\mu(x^*(\lambda),\lambda) \tag{7}$$

From this definition, we can infer (see 7.2),

$$\nabla g(\lambda) = c(x^*(\lambda)) \tag{8}$$

$$\nabla^2 g(\lambda) = -\nabla c(x^*(\lambda))^T \nabla_{x,x}^2\mathcal{L}(x^*(\lambda),\lambda)^{-1}\nabla c(x^*(\lambda)) \tag{9}$$

Gradient Type method

In order to do a first-order dual gradient ascent update, we need to calculate $\nabla g(\lambda^k)$ following the previous discussion. However, the calculation of the exact value of $x^*(\lambda^k)$ is too expensive. Instead, we implement a gradient-type primal-dual method by taking a gradient descent step on the primal variable. The dual update can be derived from equation (7). We define α and β (both > 0) as the primal and dual step sizes:

$$x^{k+1} = x^k - \alpha\nabla_x\mathcal{L}(x^k,\lambda^k) \tag{10}$$

$$\lambda^{k+1} = \lambda^k + \beta\nabla g(x^k) = \lambda^k + \beta Wx^k \tag{11}$$

Under a distributed setting, the primal and dual update equations at each iteration for all clients with index i that perform first order update become:

$$x_i^{k+1} = x_i^k - \alpha\left(\nabla f_i(x_i^k) - \lambda_i^k + \mu(x_i^k - x_0^k)\right) \tag{12}$$

$$\lambda_i^{k+1} = \lambda_i^k + \beta(x_0^k - x_i^k) \tag{13}$$

The server collects the information and performs the following if all clients uses first order method:

$$x_0^{k+1} = x_0^k - \alpha\left[\sum_{i=1}^n \lambda_i^k + \mu\left(nx_0^k - \sum_{i=1}^n x_i^k\right)\right] \tag{14}$$

Newton Type methods

Clients with better computational capabilities are meant to use this method. We start by first computing the primal update, which solves the root-find problem derived earlier (23). Because the primal Hessian has a nonseparable term W^TW, calculating the exact value of the Hessian inverse is intractable. This leads to the approximation of W^TW with its block diagonal for primal update, i.e. I_d. The hessian $H = \nabla_{x,x}\mathcal{L}_\mu(x, \hat{\lambda})$ can then be approximated by $\nabla^2 f_i(x_i^k) + \mu I_d$. The classical (damped) Newton update is applied using this approximation:

$$x^{k+1} = x^k - \alpha H^{-1}\nabla_x \mathcal{L}_\mu(x^k, \hat{\lambda}) \tag{15}$$

For the dual variables we use the method of multipliers [4] in order to perform an ascent update on the dual function $g(\lambda)$. In particular, we use the following second-order update rule:

$$\lambda^{k+1} = \lambda^k - \beta\Delta\lambda^k \quad \text{s.t.} \quad \nabla^2 g(\lambda^k)\Delta\lambda^k = \nabla g(\lambda^k) \tag{16}$$

Replacing by values in (8), (9), and by using the same hessian approximation H, we get the following dual update:

$$\lambda^{k+1} = \lambda^k - \beta\left[\nabla^2 g(\lambda^k)\right]^{-1}\nabla g(\lambda^k) = \lambda^k + \beta\left[\nabla c(x^k)^T H^{-1}\nabla c(x^k)\right]^{-1} c(x^k) \tag{17}$$

Putting everything together we get:

$$x_i^{k+1} = x_i^k - \alpha(\nabla^2 f_i(x_i^k) + \mu I_d)^{-1}(\nabla f_i(x_i^k) - \lambda_i^k + \mu(x_i^k - x_0^k)) \tag{18}$$

$$\lambda_i^{k+1} = \lambda_i^k + \beta(\nabla^2 f_i(x_i^k) + \mu I_d)(x_0^k - x_i^k) \tag{19}$$

Server update:

$$x_0^{k+1} = \frac{1}{n}\sum_{i=1}^n x_i^k - \frac{1}{\mu n}\sum_{i=1}^n \lambda_i^k \tag{20}$$

Combining gradient- and Newton-type methods

The FedHybrid method tries to maximize the resource efficiency of clients with varying computational capabilities by allowing them to choose either first order or second order updates. Since the first order method is less computationally demanding, it is ideal for clients with limited resources, as it avoids the extended calculation times associated with the second order method. Alternatively, more capable clients can choose second order method and benefit from its super linear convergence rate.

Integrating the above two methods, the final Fedhybrid algorithm looks like this:

for $k \in [n]$ **do**
  Server sends x_0^k to clients
  for gradient-type [LOW] clients **do**
    $x_i^{k+1} = x_i^k - \alpha(\nabla f_i(x_i^k) - \lambda_i^k + \mu(x_i^k - x_0^k))$
    $\lambda_i^{k+1} = \lambda_i^k + \beta(x_0^k - x_i^k)$
  end for
  for Newton-type [HIGH] clients **do**
    $x_i^{k+1} = x_i^k - \alpha(\nabla^2 f_i(x_i^k) + \mu I_d)^{-1}(\nabla f_i(x_i^k) - \lambda_i^k + \mu(x_i^k - x_0^k))$
    $\lambda_i^{k+1} = \lambda_i^k + \beta(\nabla^2 f_i(x_i^k) + \mu I_d)(x_0^k - x_i^k)$
  end for
  Server update:

$$x_0^{k+1} = \frac{1}{n}\sum_{i=1}^{n} x_i^{k+1} - \frac{1}{\mu n}\sum_{i=1}^{n} \lambda_i^{k+1}$$

end for

3 Implementation

The FedHybrid proposal has been implemented in the way of a Python framework, that enables us to compare the results for different scenarios.

Overview of the framework

The framework develops different concepts which permit a great diversity of trials and tests.

Task A task is the combination of an optimization problem with a dataset. The optimization problem is defined by the stopping criteria (maximum number of iterations, tolerance on minimum change in cost, tolerance on minimum change in solution), as well as the definition of the cost function with its corresponding gradient and hessian (that we input ourselves without leveraging auto-differentiation). Importantlty, the cost function should be decomposable and taking arbitrary long non-empty list as inputs, as well as hyper-parameters. The dataset is either loaded or generated, depending on the tasks.

Task configuration We specify the number and types of clients, as well as their learning rates, using configuration objects.

Server The server is in charge of learning the consensus solution by communicating with the clients.

Client A client is either HIGH or LOW, depending on its given (virtual) computational capabilities. They implement the updates mentioned in previous sections.

A minimal usage of our framework can be found in the Annex, as well as some details about our implementation. The code is open-source and can be found on the following page: https://github.com/Antoine-Bergerault/18460-Project.

4 Results

Linear Regression

In this task, we generate data by sampling points along a line and then adding noise to it. The goal is then to determine what the slope and y-intercept of the line of best fit are.

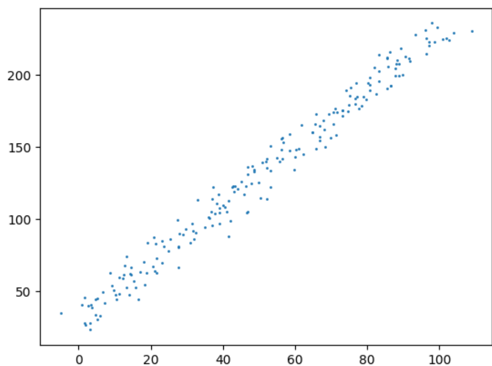

Figure 1: An example dataset (slope = 2, y-intercept = 30)

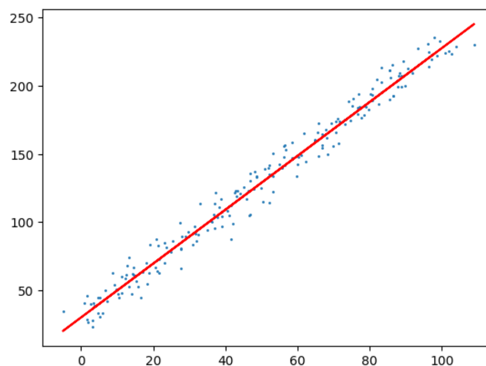

Figure 2: Derived solution to Fig. 1 1xHIGH Successfully converged in 2 iterations.

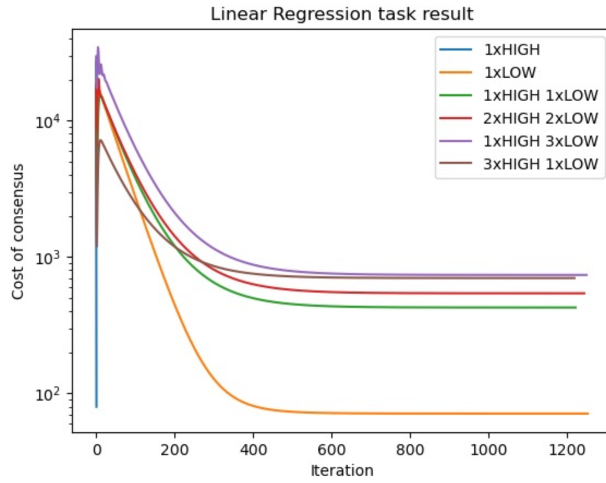

Figure 3: Cost function of different configurations

Fig.2 shows the result of performing linear regression using one HIGH client that performs Newton's method. The result convergences in 2 iterations as expected due to approximation in our algorithm. In Fig.3, we tried different numbers of HIGH and LOW clients and evaluated their costs. It is found that configurations with more LOW clients converge slower than configurations with more HIGH clients. This result matches the expectations and the original paper.

Circle

In this task, we generate data by randomly selecting points that fall within a specified circle. The goal is to determine what the center and radius of the circle.

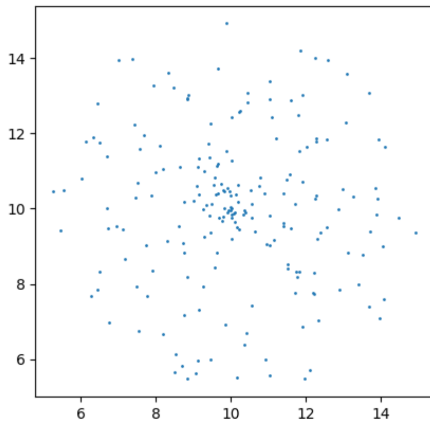

Figure 4: An example dataset (center at (10, 10), radius = 5)

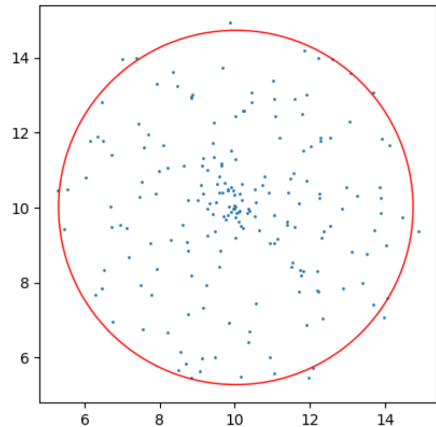

Figure 5: The derived solution to the problem referenced in Fig. 4. Successfully converged in 114 iterations.

UCI Mushroom

In this task, we used the UCI mushroom data set found on Kaggle. This data set includes a variety of mushroom features such as number of rings or stalk length as well as specify whether or not a mushroom is edible or poisonous. Using these features, our goal is to try to determine which mushrooms are poisonous and which ones are edible.

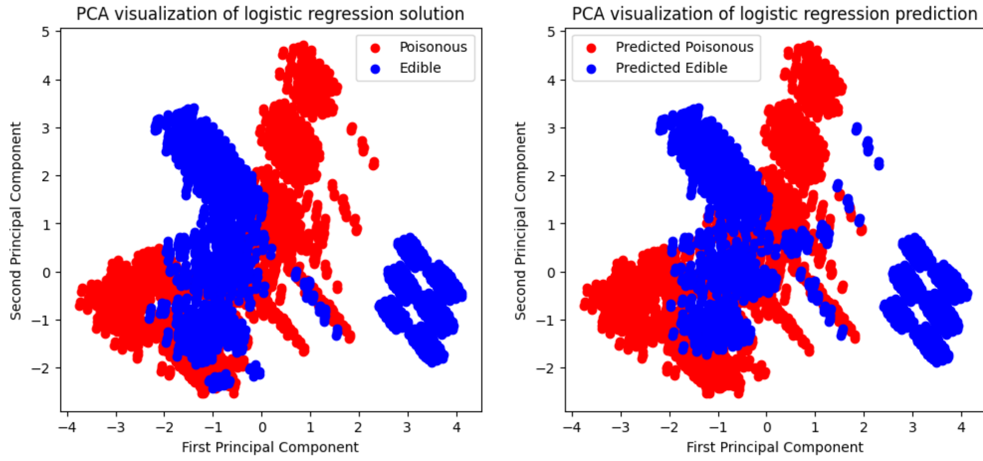

Figure 6: Left hand side is the actual data-set of poisonous vs. edible mushrooms. Right hand side is the predicted poisonous and edible mushrooms using a network of one client with high computational capabilities. Successfully converged in 14486 iterations.

Overall, a general issue we noticed was that once we started introducing more clients into the network, it was unable to converge and derive a consensus.

In the more simple test cases of linear regression and circle, this may be due to issues in the way the data is partitioned and the learning rates. A similar issue could fall in the way data is partitioned for the mushroom task. However as shown in Fig. 7, the network quickly converges to a solution and then afterwards marginally increased per iteration.

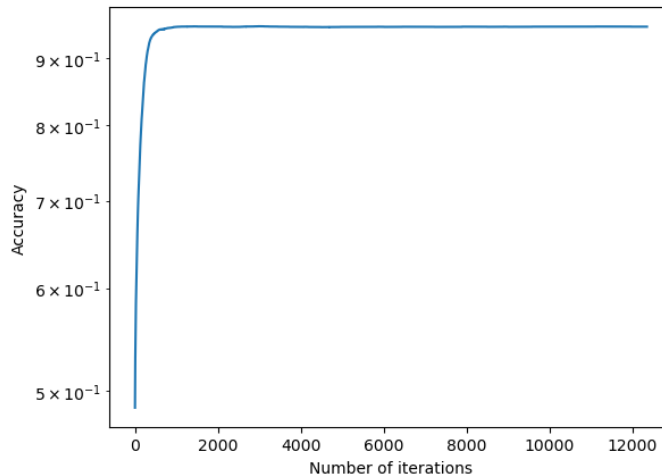

Figure 7: Accuracy over iteration on the mushroom task as defined in Fig. 6

5 Discussion

The FedHybrid method offers a promising solution for optimizing heterogeneous federated learning algorithms. This approach not only ensures data privacy and security but also allows clients with varying computational capabilities to choose between first-order gradient-type or second-order Newton-type methods. This significantly enhances resource utilization efficiency, particularly during a global chip shortage where access to high-performance hardware is limited.

In both the original paper and our reproducibility study, the FedHybrid method was only tested on small datasets. Although the method's linear convergence rate is mathematically proven, its performance on real-life, large datasets remains uncertain. More research is needed to implement the method on hardware. additionally, since the performance of FedHybrid is influenced by the

number of clients and their capability configurations, further investigation should focus on how users can optimize these configurations.

The FedHybrid method is only a starting point of developing optimization algorithms that aim for heterogeneous clients. Future research may refine or build upon this algorithm to achieve even better performance.

6 Contributions of the team members

Antoine

Mainly contributed to the structure of the program from defining the workflow to creating the different classes and replicating the optimization updates from the FedHybrid paper. Was also responsible for both the numerical implementation details (regularization of the hessians, solving linear systems) and the two toy examples, namely linear regression and circle tasks. Wrote about the augmented Lagrangian method and the implementation for the report.

Felicia

Mainly contributed to the data processing and results visualization of the testing tasks. Encoded Mushrooms dataset and contributed to the implementation of the logistic regression task. Helped tune parameters for the best convergence rate for linear regression and logistic regression tasks. Wrote about the details of first order and second order FedHybrid method and discussion.

Sophia

Mainly contributed by implementing concurrency and having the clients compute simultaneously rather than waiting for each node to finish its computation. Also helped with fine tuning parameters for each of the tasks to get the lowest cost in each task with the fewest number of iterations. Wrote about the concurrency implementation and summary of our tasks and results obtained.

References

[1] Magnus R Hestenes. "Multiplier and gradient methods". In: *Journal of optimization theory and applications* 4.5 (1969), pp. 303–320.

[2] Tian Li et al. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60. DOI: 10.1109/MSP.2020.2975749.

[3] Xiaochun Niu and Ermin Wei. "FedHybrid: A hybrid federated optimization method for heterogeneous clients". In: *IEEE Transactions on Signal Processing* 71 (2023), pp. 150–163.

[4] Richard A Tapia. "Diagonalized multiplier methods and quasi-Newton methods for constrained optimization". In: *Journal of Optimization Theory and Applications* 22 (1977), pp. 135–194.

7 Annex

7.1 Augmented Lagrangian

Here we quickly go over how the Augmented Lagrangian works, and how it relates to what we saw in class.

The idea of a classical Lagrangian is to build a function that naturally expresses the fact that, when minimized, the gradient of the cost should be collinear to the gradient of the constraint curve. When multiple equality constraints are imposed, each of them should satisfy this condition, and when inequality constraints are used, we need to distinguish them between active and non-active ones. We assume that the zero vector is collinear to any other vector of the same dimension.

To simplify the discussion, we start by only assuming equality constraints $c(x) : \mathbb{R}^n \to \mathbb{R}^m$ and a given cost function $f(x) : \mathbb{R}^n \to \mathbb{R}$. The lagrangian of this problem is then defined as follows:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T c(x) \tag{21}$$

An augmented Lagrangian will be similar, but adds a penalty term for constraints violations, and uses instead an estimate of the Lagrange multipliers:

$$\mathcal{L}_\mu(x, \hat{\lambda}) = f(x) + \hat{\lambda}^T c(x) + \frac{\mu}{2}\|c(x)\|_2^2 \tag{22}$$

Iteratively, we can then optimize $\mathcal{L}_\mu$ with respect to x, and update our estimate of the duals. Not performed in the FedHybrid paper nor in our implementation, the penalty parameter μ can also be updated geometrically. The first order optimality condition is:

$$\nabla_x \mathcal{L}_\mu(x^*, \hat{\lambda}) = \nabla f(x^*) + \frac{\partial c(x^*)}{\partial x}^T \left[\hat{\lambda} + \mu c(x^*)\right] = 0 \tag{23}$$

And this root finding problem can be easily solved by Newton's method. The extension to a case with inequality constraints $d(x)$ is not too complicated, as it can be rewritten as equality constraints: $d_i'(x) = \max\{0, d_i(x)\}$.

7.2 Dual function gradient and hessian

We adapt a proof derived in [4], which provides solution for the gradient and hessian of the dual function.

With:

$$\nabla_x \mathcal{L}_\mu(x^*(\lambda), \lambda) = 0 \tag{24}$$

$$\nabla x^*(\lambda) = -\nabla c(x^*(\lambda))^T \nabla^2_{x,x} \mathcal{L}(x^*(\lambda), \lambda)^{-1} \tag{25}$$

We get:

$$\begin{aligned}
\nabla g(\lambda) &= \nabla_\lambda \mathcal{L}_\mu(x^*(\lambda), \lambda) + \nabla x^*(\lambda) \nabla_x \mathcal{L}_\mu(x^*(\lambda), \lambda) && (26)\\
&= c(x^*(\lambda)) - \nabla c(x(\lambda))^T \nabla^2_{x,x} \mathcal{L}(x(\lambda), \lambda)^{-1} \nabla_x \mathcal{L}_\mu(x^*(\lambda), \lambda) && (27)\\
&= c(x^*(\lambda)) && (28)
\end{aligned}$$

$$\begin{aligned}
\nabla^2 g(\lambda) &= \nabla x(\lambda) \nabla c(x^*(\lambda)) && (29)\\
&= -\nabla c(x^*(\lambda))^T \nabla^2_{x,x} \mathcal{L}(x^*(\lambda), \lambda)^{-1} \nabla c(x^*(\lambda)) && (30)
\end{aligned}$$

7.3 Minimal usage of the framework

```
import numpy as np
from server import Server
from tasks import lrt

task = lrt.LinearRegressionTask() # will use default configuration
task.visualize()

server = Server(task)
server.connect_clients()

problem = task.get_problem()

k = 0
last_cost = float('infinity')
while k < problem.max_iter and server.delta > problem.tol:
    current_cost = problem.loss(server.consensus.flatten(),
                                task.dataset, problem.hyper_parameters)
    server.run_iteration(k+1)

    if np.linalg.norm(current_cost - last_cost) < problem.ctol:
        last_cost = current_cost
        break

    last_cost = current_cost
    k = k + 1

if k >= problem.max_iter and server.delta > problem.tol:
    raise Exception("Did not converge")

solution = server.consensus.flatten()
task.visualize_solution(solution)
```

7.4 Concurrency

A naive approach for the server to derive the consensus solution is to iterate through all the clients and call their update function.

```
        for client in self.clients:
            client.update(self.consensus, k)
```

However, this method is slow especially in the situations where there are many clients within a network. Not only is it inefficient, but it also does not mimic the actual functionality of a network in which nodes are independent: one node does not need to wait for another node to finish its computation before it can begin computing.

Instead, we can utilize threads to call the update function for each client as follows

```
threads = []
        # create threads for each client
        for client in self.clients:
            x = threading.Thread(target=client.update, args=(self.consensus, k,))
            threads.append(x)
            x.start()

```

```python
        # join threads together once they have finished
        for x in threads:
            x.join()
```

7.5 Numerical implementation details

This algorithm requires us to compute the following primal update for a client i performing Newton-type updates:

$$x_i^{k+1} = x_i^k - a_i \left(\nabla^2 f_i(x_i^k) + \mu I_d\right)^{-1} \left(\nabla f_i(x_i^k) - \lambda_i^k + \mu(x_i^k - x_0^k)\right) \tag{31}$$

Which is of the form:

$$x = A^{-1}b \Leftrightarrow Ax = b \tag{32}$$

Even though computing an inverse is totally possible using NumPy (with `np.linalg.inv`) or other libraries and languages, it should be avoided in general. The first reason for this is that the inversion procedure is not numerically stable. When A is ill-conditioned (i.e. its condition number is high), then the result can blow up due to numerical imprecisions. One solution to this is to perform a QR decomposition $A = QR$, and then solve

$$QRx = b \Leftrightarrow Rx = Q^T b \tag{33}$$

using backward substitution. Other methods exist (e.g, using the SVD decomposition) with their advantages and disadvantages. This is not useful to us to understand which method is the best, so we decided to solve such equations using `np.linalg.solve(A, b)`.

The second reason why inverting a matrix using `np.linalg.inv` should be avoided, but that we do not take into account in our analysis, is that there are more specialized algorithms when the matrix A has a special structure. In particular, sparse matrices can be inverted more efficiently using other algorithms.