DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Asynchronous Optimization Algorithms

René Filip

DEPARTMENT OF INFORMATICS

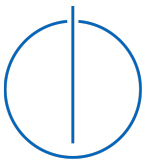
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Asynchronous Optimization Algorithms

Asynchrone Optimierungsalgorithmen

Author:	René Filip
Supervisor:	Prof. Dr. Daniel Cremers
Advisor:	Dr. Tao Wu
Submission Date:	March 16th, 2020



I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, March 16th, 2020 René Filip

Acknowledgments

I would like to thank my advisor, Dr. Tao Wu, who provided excellent feedback and meaningful insights throughout our discussions. In addition, I wish to extend my special thanks to my parents, Gertrud and Georg Filip, who greatly supported me during my studies.

Abstract

Recently, distributed optimization became more and more popular due to the increase in available training data and available computing power. There exist many distributed variants to classical optimization algorithms like Distributed Gradient Descent or Distributed ADMM but their theory often requires that computation nodes run synchronously. That means, for each iteration, all nodes must finish with their local computation and the algorithm must coordinate all nodes. This can become a bottleneck when there are slow nodes within the network, when communication breaks down or when the overhead of coordination becomes too large. *Asynchronous* optimization algorithms have gained attention within the distributed optimization community because they often overcome this bottleneck. In this thesis, we give an overview of centralized, asynchronous algorithms and perform various experiments on computer vision tasks. We look into their convergence theory but also provide examples of how we transform a given problem into a distributed one. Last but not least, we use a real computer cluster of several GPUs to optimize over a function asynchronously.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Centralized and Decentralized Algorithms 3
1.2 Different Kinds of Asynchronicities 4
1.2.1 Synchronous Coordination 4
1.2.2 Partial Barrier & Bounded Delay 5
1.2.3 Update Sets 6
1.2.4 Delays 7
1.3 Theoretical Framework vs. Practical Implementation 8
1.4 Overview on Distributed Algorithms 9

2 Theory of Asynchronous Algorithms **11**
2.1 Synchronous Algorithms 11
2.1.1 Stochastic Gradient Descent (SGD) 11
2.1.2 ADMM and the Consensus Problem 14
2.2 Asynchronous Stochastic Gradient Descent 16
2.3 Asynchronous Block Coordinate Descent 26
2.4 Flexible ADMM 31
2.4.1 Convergence Analysis 31
2.4.2 Order of Updates 37
2.4.3 Delayed ADMM 37

3 Examples and Experiments **39**
3.1 SGD Experiments 39
3.2 Asynchronous ADMM Logistic Regression 41
3.2.1 Derivation 41
3.2.2 Simulated Delays 44
3.2.3 Real Delays 51

3.3 Asynchronous ADMM Image Segmentation 54
3.3.1 Derivation of the Master Update 54
3.3.2 Scaled Projection onto Probability Simplex 57
3.3.3 Derivation of the Worker Updates 62
3.3.4 Solving the Huber-ROF Problem 62
3.3.5 Final Algorithm 65
3.3.6 Simulated Delays 66
3.3.7 Real Delays 69
3.4 PyTorch's `distributed` Package 73

4 Conclusion **76**
4.1 Further Research 76

List of Figures **78**

List of Tables **79**

List of Algorithms **80**

Bibliography **81**

1 Introduction

Today, data plays a more and more important role in research, industry, and society. Over time, advances in hardware made it possible to solve problems by retrying existing methods or developing new ideas that were not possible in the past. Due to hardware advances in faster and larger storage, it became possible to store data that could not be stored previously because of high costs, data decay or slow reading/writing speed. Considering "Industry 4.0", many (industrial) machines started to persist their sensor data in order to make them available for analytics [Mob02]. Not only data storage but also processing power increased, especially if we look at Graphics Processing Units (GPUs). This made it possible to revisit existing methods in Machine Learning and apply them again. For instance, it is well known that "Neural Networks" (NN) were proposed by [Ros58] in 1958 but could not solve many problems efficiently with the available hardware back then. Only recently, with the growth of data and computing power, Neural Networks started to outperform well-established methods such as the Support Vector Machine (SVM) in areas like Computer Vision, as famously shown in [KSH12]. Additionally, faster internet connections and general internet availability allow us to process data from various sources and distribute it again. For example, GPS systems within smartphones make it possible to detect traffic jams and can help the driver to find a faster route.

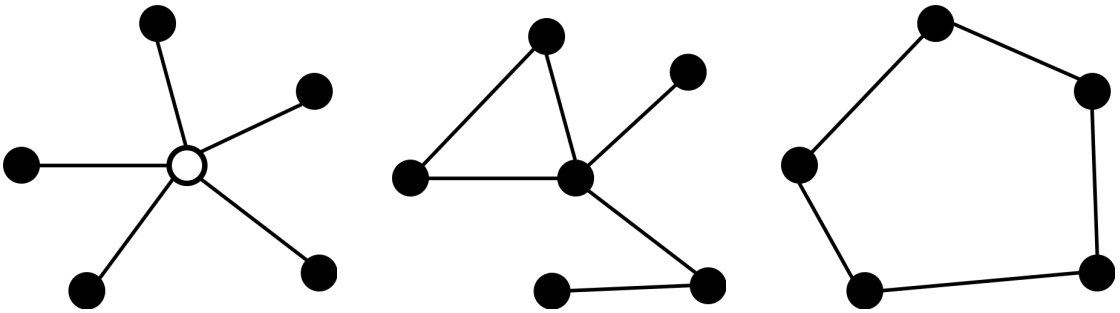
In order to solve a certain problem, we need to collect the related data, give it to a model and train it until it solves our problem sufficiently. But, depending on the size of the data, its structure, and the available hardware, the training or optimization process can vary from seconds to even months. Often, training time is constrained and must not exceed a certain threshold so one needs to find a way to speed up the process. A very common and natural way to do so is by distributing the total work to multiple machines and splitting the full optimization problem into smaller parts. This makes sense when available hardware resources are not fully utilized yet or when the problem is so big that one machine cannot solve it on its own. Or, the training data might be already distributed and cannot be easily collected.

For this reason, many distributed optimization algorithms have been proposed in the literature and we refer to the surveys [Ber15], [FSS15], [BCN18a] and [Yan+19] for an

introduction to optimization and distributed optimization. However, we need to be aware of the additional complexity that comes with it. First, we need to verify that a given problem can be solved in a distributed way or we need to reformulate an equivalent problem that enables distribution. Not seldom, this reformulation makes the problem more difficult by introducing additional parameters so one needs to estimate whether distributed computing makes sense or not. Second, distribution requires sending and receiving data from other nodes. In general, nodes could be threads, CPUs or different machines. In this thesis, we mostly focus on distribution onto different computers. Therefore, we must establish connections between them and coordinate them with each other to send the right information at the right time. Third, because communication networks are often fragile, we need to be resilient against errors like communication failures, node failures or slow responses from machines. Fourth, there are different communication topologies that need to fit with the optimization algorithm itself. For instance, an algorithm that uses a master-worker architecture performs very badly (or not at all) if machines cannot communicate efficiently in a star-shaped topology. All this contributes to slowing down the final distributed algorithm or making it fail.

In our thesis, we are interested in the effects of weakening or even removing the coordination requirement of centralized distributed algorithms between nodes, i.e. when nodes run asynchronously to each other. In specific, we look into the consequences when not all machines finished their task, when communication fails or when workers can only use outdated information. We introduce different categories of this asynchronous behavior and evaluate their usefulness in real life. Not only, we present the theory of existing asynchronous algorithms but we also show how we can distribute a problem and utilize an asynchronous algorithm and how it affects the performance and final results.

Continuing in this Chapter 1, we provide an overview of asynchronous algorithms in general and discuss differences between them. Starting in Chapter 2, we highlight three asynchronous versions of centralized algorithms, namely asynchronous Stochastic Gradient Descent, asynchronous Block Coordinate Descent and of asynchronous Alternating Direction Method of Multipliers and reproduce their convergence proofs to see different proof techniques in dealing with asynchronicity. In the practical Chapter 3, we show how we can solve a logistic regression and an image segmentation problem by using a distributed algorithm. Last but not least, in Chapter 4 we conclude our thesis.



(a) centralized, star shape topology

(b) decentralized topology

(c) ring topology

Figure 1.1: Examples of common communication topologies

1.1 Centralized and Decentralized Algorithms

When we speak of distributed algorithms, one way to categorize them is to look at their global topology, i.e. how nodes are connected with each other. We represent the communication by a graph where nodes represent physical machines and edges represent a possible flow of information. Figure 1.1 shows some common topologies.

On the one hand, a centralized algorithm consists of a master node coordinating several worker nodes. The master node sends out its current state, delegates new tasks to its workers and collects the results from them at a later point. Worker nodes solve the given task and send the result back to the master. Notice that communication is limited between only the master and worker. We refer to it as the "star" shaped topology.

On the other hand, a decentralized algorithm does not have a master node and nodes communicate freely with each other depending on the given network. Even though the focus of this thesis is on centralized algorithms, we want to name a few decentralized methods from the literature.

The authors in [NO09][YLY16][ZY18] considered the algorithms "DGD" and "Prox-DGD" and introduced the analysis for a distributed gradient descent algorithm where the topology is given by a mixing matrix. Their method is inspired by the "Distributed Averaging" algorithm [XB04][BDX03] but can optimize over convex and nonconvex problems. In EXTRA [Shi+15b], PG-EXTRA [Shi+15a] and Asynchronous EXTRA [Wu+18] the initial DGD method has been extended to achieve a faster convergence. In Hogwild! [Niu+11] and AD-PSGD [Lia+17] the topology is represented directly by a graph and for each update iteration, a random edge is sampled to simulate the flow of communication.

All previous algorithms collect and aggregate gradients from their neighbor nodes to perform an update on the optimization variable. In contrast to that, we can also solve a local optimization problem directly and aggregate the resulting models which are common in algorithms like ADMM [Boy+11]. Decentralized ADMM [Shi+14], Async ADMM [WO13] and ASYMM [Far+19] encode the topology by adding new constraints to the problem. For example, [Shi+14] introduces an auxilary variable x_i for each node i and z_{ij} for each edge between node i and j. Then, it enforces equality $x_i = z_{ij}$ and $x_j = z_{ij}$ for all edges.

Due to hardware or software errors, not seldom communication between two nodes can break down. This inspired [NOS17] and [HC17] to consider time-varying graphs that model these real-life scenarios better.

1.2 Different Kinds of Asynchronicities

Before we explore various distributed, asynchronous algorithms, we first need to get a basic understanding of the term "asynchronicity" because it can refer to different kinds of degrees of the same idea. In this section, we introduce these different degrees and point out their shortcomings in modeling. Needless to say, one needs to be careful and distinguish between the theoretical framework and the real application. While in theory, an (asynchronous) algorithm might converge, in practice it might not be the case because the model does not capture the real world.

We start with the lowest kind of asynchronicity, namely no asynchronous behavior at all.

1.2.1 Synchronous Coordination

In a synchronous (distributed and centralized) application, the master node always waits until all workers report back with a result. The master proceeds with the most recent results from the workers and the workers always have the most recent information from the master. Assuming that all nodes always have access to the most recent information is a very common assumption since it often simplifies the analysis, as we see in Section 2.1. In addition, it is not unreasonable to think that having always the most recent data will provide good results. Figure 1.2 visualizes a very general centralized, synchronous algorithm. Colored boxes represent calculation time.

On the downside of this approach, collecting results from all workers or distributing new tasks to all workers might not always be fast or even possible. The whole calculation

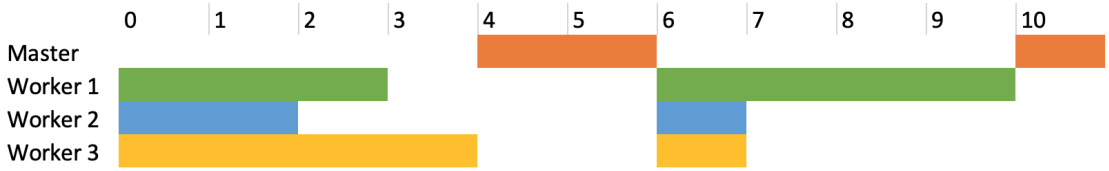


Figure 1.2: Example calculation time of master and worker nodes

time is bottlenecked by the slowest worker in the network. Notice that the calculation time of the master in Figure 1.2 only starts once the calculation of workers 1, 2 and 3 have finished. Then, the workers stay idle until the calculation of the master node finishes. This can result in big idle times for faster nodes. Even worse, due to physical network issues, not seldom, messages with results get lost. Without a meaningful mechanism of recovery, a communication error could stop the whole optimization procedure.

1.2.2 Partial Barrier & Bounded Delay

One method to cope with the bottlenecks mentioned above is by introducing a "partial barrier" and a "bounded delay" as it is done in [ZK14]. Instead of waiting for *all* K workers to finish a given task, we could also just wait for $B \leq K$ workers to finish. That way, we reduce the idle time of fast workers because the master node does not have to wait for a slow worker to finish in order to define and send out a new job. Setting $B = K$ gives us a synchronous version.

Problems might arise when one worker is constantly slower than other ones. If we only define a partial barrier B, then fast workers could skew the final results because they were considered much more often than slow ones. To prevent this, we also define a "bounded delay" τ that ensures within τ master iterations, all workers have contributed their results to the master at least once. Figure 1.3 shows how a partial barrier and a bounded delay work together.

For the first two master iterations, only the master uses the results from worker 1 and 2, because worker 3 is still busy. But, because we defined a low partial barrier, in the third master iteration we actually have to wait for worker 3 to finish before the master can continue. Notice how the calculation of the master and of a worker (here worker 3) can run in parallel now.

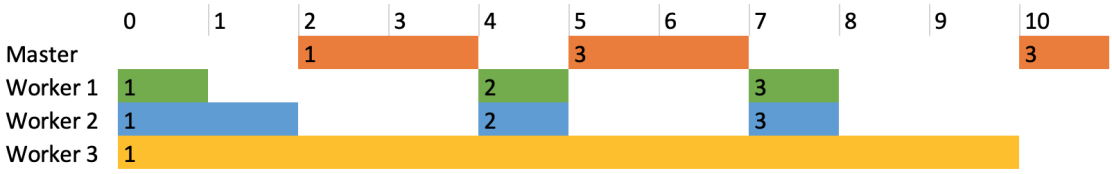


Figure 1.3: Partial barrier and bounded delay example

1.2.3 Update Sets

A very similar but slightly more general concept is "update sets". Instead of waiting for B workers to finish their calculations, for each iteration t we define a subset of nodes $\mathcal{C}^t \subseteq \{0, \dots, K\}$ that perform an update where $k = 0$ represents the master node and $1 \leq k \leq K$ represents worker nodes. To enforce synchronicity, we simply set $\mathcal{C}^t = \{0, \dots, K\}$ for all iterations. Update sets are used directly in Flexible ADMM [HLR16a] and variants are used in Hogwild! [Niu+11], Async ADMM [WO13] and ASYMM [Far+19]. To deal with the problem of very infrequent updates by slow workers or even a slow master, we need to introduce the "essentially cyclic update rule". For each iteration t, we require that

$$\bigcup_{i=1}^{T} \mathcal{C}^{t+i} = \{1, \dots, K\} \quad \forall t$$

for a given interval of T.

Assume we have the situation as visualized in Figure 1.4.

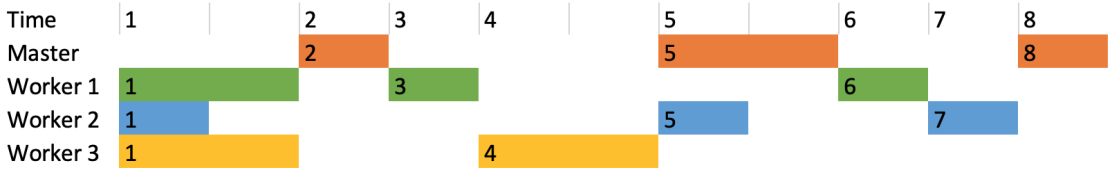


Figure 1.4: Possible updates by an algorithm

By evaluating when each node starts and finishes their current calculation, we can

derive these update sets (Figure 1.5), where each block represents an element of that set and each column symbolizes a set at a given time.

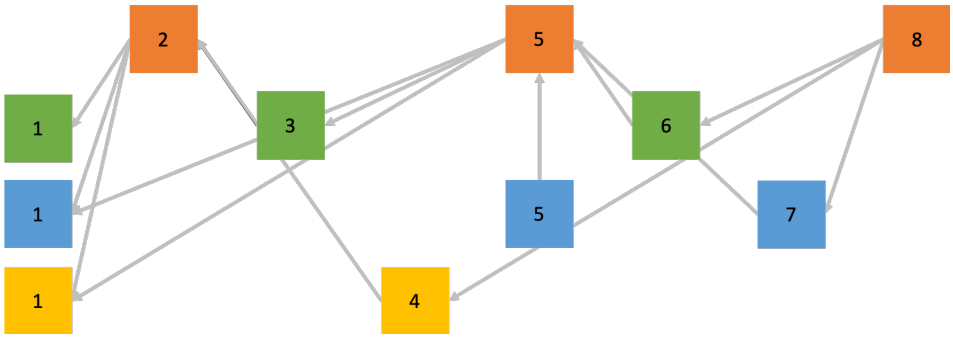


Figure 1.5: Update set perspective of the previously mentioned situation

But, notice that there is still quite a lot of idle time in the example. If we use an algorithm that tries to reduces this idle time, then update sets might not be the right modeling choice as we will see in the next section.

1.2.4 Delays

Another way to simulate asynchronicity is by introducing "delays". Here, we allow each update to use "older" or "delayed" variables from previous iterations. For instance, in iteration t we draw a delay $\tau_t \geq 1$ and perform an update using the variable $x_{t-\tau_t}$ instead of x_{t-1}. Synchronous behaviour is achieved by setting $\tau_t = 1$. In the previously discussed update sets modeling, the master node always refers to the most recent data that is available. However, consider following situation that could occur in Figure 1.6:

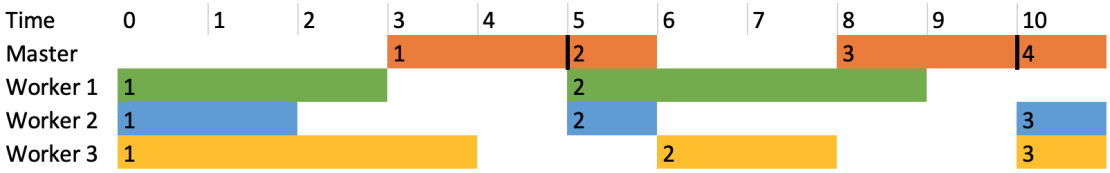


Figure 1.6: Example where update sets cannot model the real world

By taking a closer look at the first master iteration and the first worker 3 iteration, we run into the problem that we cannot capture this situation by just using update sets. If

we put the first iteration of worker 3 into a set *before* the first iteration of the master, then the master would use the result of worker 3. But, if we put it *after* the master iteration, then worker 3 would use the result of the master which does not model the given situation as well. Clearly, we need a different approach. By using delays we generalize the concept of update sets and also solve our problem above. For example, Figure 1.7 visualizes the same situation but using a block representation where each block represents an update step. The gray arrows indicate the dependencies of the updates to previous ones and the two red arrows provide the solution to our initial problem from Figure 1.6.

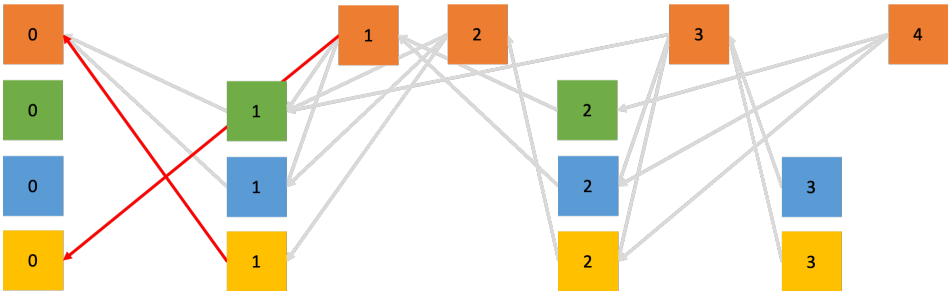


Figure 1.7: Block representation for delays

Delays are used in Asynchronous EXTRA [Wu+18], AsySG-con [Lia+15], AD-PSGD [Lia+17], Async-BCD [LW14] and Async PADMM [Hon18].

1.3 Theoretical Framework vs. Practical Implementation

Notice that we often differentiate between the theoretical algorithm and its practical implementation. For example, in practice, we use one implementation for the master node and one implementation for the worker nodes while the theoretical description unifies both into one algorithm. In fact, this is the case for all algorithms discussed previously and the ones we look into detail in Chapter 2. We split between these two perspectives because a unified description simplifies the theoretical analysis while still staying valid in practice.

In our "simulated delays" experiments in Section 3.1, 3.2.2 and 3.3.6 we actually implement the *theoretic* description because we only simulate possible delays. But, in the "real delays" experiments (Section 3.2.3 and 3.3.7) we use a practical implementation and differentiate between master and workers.

Throughout this thesis, we simulate delays by using outdated variables with a specific

age. Every iteration, we store the updated variables in a queue. When we refer to an (outdated) variable, we sample a random element from that queue. For instance, if we want to simulate the worst-case delay scenario, we always sample the last element from that queue to get the most outdated variable. Or, we sample an element by drawing a random index from a uniform distribution to simulate a noisy but more realistic setting. Further research can look into other sampling strategies for evaluation that better model the real world.

1.4 Overview on Distributed Algorithms

Finally, we summarize all previously mentioned algorithms and give an overview in Table 1.1. Namely, what kind of problems they can solve, how they perform their optimization, on what kind of topology they work on and which asynchronicity they use.

Table 1.1: Short summary of distributed algorithms for various situations

Algorithm	Problem	Aggregation	Topology	Asynchronicity	Additional remarks
DGD and Prox-DGD [NO09][YLY16][ZY18]	nonconvex, smooth (+ proximable)	gradient	decentralized	synchronous	-
EXTRA and PG-EXTRA [Shi+15b][Shi+15a]	convex, smooth (+ proximable)	gradient	decentralized	synchronous	extension to (Prox-)DGD
Async. EXTRA [Wu+18]	convex, smooth (+ proximable)	gradient	decentralized	delays	-
AsySG-con [Lia+15]	nonconvex, smooth	gradient	centralized	delays	see Section 2.2
Hogwild! [Niu+11]	strongly convex, smooth	gradient	decentralized	update sets	shared memory
DC-ASGD [Zhe+16]	nonconvex, smooth	gradient	centralized	delays	extension to AsySG-con
AD-PSGD [Lia+17]	nonconvex, smooth	gradient	decentralized	delays	-
DIGing [NOS17]	strongly convex, smooth	gradient	decentralized, dynamic graph	synchronous	-
DySPGC [HC17]	convex, nonsmooth	gradient	decentralized, dynamic graph	synchronous	-
Async-BCD [LW14] [SHY17]	nonconvex, smooth	gradient	centralized	delays	see Section 2.3
ARock [Pen+15]	nonexpansive operator	gradient/ model	decentralized	delay	-
Decentralized ADMM [Shi+14]	strongly convex, proximable	model	decentralized	synchronous	-
Async. ADMM [WO13]	convex, proximable	model	decentralized	update sets	-
Async. PADMM [Hon18]	nonconvex, smooth + proximable	gradient/ model	centralized	delays	-
ASYMM [Far+19]	nonconvex, smooth	model	decentralized	update sets	-
Async-ADMM [ZK14]	convex, proximable	model	centralized	partial barrier	-
Flexible ADMM [HLR16a]	nonconvex, smooth + proximable	model	centralized	update sets	see Section 2.4

2 Theory of Asynchronous Algorithms

Adding asynchronicity adds complexity to the underlying theory of an algorithm. In this chapter, we are interested in the assumptions of asynchronous algorithms and we present methods on prooving convergence. First, we look at synchronous SGD and ADMM. Then, we present an asynchronous version of SGD, asynchronous Block Coordinate Descent and finally an asynchronous version of ADMM.

2.1 Synchronous Algorithms

Before discussing asynchronous distributed algorithms we give some insights into synchronous ones. First, we look at the convergence of the well understood Stochastic Gradient Descent (SGD) optimizer [RM51]. Then, we summarize the main idea of the ADMM method [Boy+11] and introduce the Consensus Problem 2.1.3 that we will later solve in the practical Section 3.2.

Even though these two algorithms have not been designed for a distributed computing network, the theory behind them still holds if they run in synchronization, i.e. as long as the order of iterations and variables are not violated. From a theoretical perspective, they appear to run sequentially, even though *some* calculations happen in parallel.

2.1.1 Stochastic Gradient Descent (SGD)

SGD [RM51] is one of the most popular optimizers due to its simplicity and great performance for learning tasks [LBH15]. In addition, many introductions to Machine Learning cover SGD, for example, [GBC16]. In this section, we present one important result from [BCN18b] that explains how the learning rate affects the convergence of SGD. To begin with, consider the Empirical Risk Problem 2.1.1.

Problem 2.1.1 (Empirical Risk)**.**

$$\min_{x \in \mathbb{R}^d} \mathbb{E}_{\xi}[F(x; \xi)]$$

where F is smooth and ξ is sampled from a given distribution.

Let T be the number of iterations, n the number of data samples available, γ_k a learning rate and $G(x, \xi)$ an update direction, for example the gradient of $F(x, \xi)$. Then, we can solve Problem 2.1.1 by using the SGD Algorithm 1.

Algorithm 1: Stochastic Gradient Descent

1 Choose initial x_0
2 **for** $t = 0, 1, \ldots, T-1$ **do**
3 | Choose sample data $S_t \subseteq \{1, \ldots, n\}$
4 | Compute stochastic gradient $\Delta_t = \sum_{i \in S_t} G(x_t, \xi_i)$
5 | Choose step size $\gamma_t > 0$
6 | Compute new iterate $x_{t+1} \leftarrow x_t - \alpha_t \Delta_t$
7 **end**

The description does not explicitly tell us how to utilize it in a distributed setting. The idea is to distribute the gradient computation (line 4) onto machines with different samples ξ. Then, the master node collects all gradients G from its workers, sums them into an update direction Δ and computes the new iterate x_{t+1}. After distributing the new variable to all workers, the iteration starts again. Remember that each node needs to finish its calculation first before they can be aggregated by the master. Otherwise, we would violate line 4 and the Theorem 2.1.1 would fail.

Before we show the convergence theorem, we prove a short lemma first.

Lemma 2.1.1. *Let $F\colon \mathbb{R}^d \to \mathbb{R}$ be continuously differentiable and let the gradient $\nabla F\colon \mathbb{R}^d \to \mathbb{R}^d$ be L-Lipschitz continuous, that is*

$$\|\nabla F(v) - \nabla F(w)\|_2 \leq L\|v - w\|_2 \text{ for all } \{v, w\} \subset \mathbb{R}^d$$

Then, the following is true for all $\{v, w\} \subset \mathbb{R}^d$

$$F(v) \leq F(w) + \nabla F(w)^\mathsf{T}(v - w) + \frac{1}{2}L\|v - w\|_2^2$$

Proof.

$$\begin{aligned} F(w) &= F(v) + \int_0^1 \frac{\partial F(v + t(w-v))}{\partial t} \mathrm{d}t \\ &= F(v) + \nabla F(v)^\mathsf{T}(w-v) + \int_0^1 \left(\nabla F(v - t(w-v))\right)^\mathsf{T}(w-v)\mathrm{d}t \\ &\leq F(v) + \nabla F(v)^\mathsf{T}(w-v) + \int_0^1 L\|t(w-v)\|\|w-v\|\mathrm{d}t \\ &= F(v) + \nabla F(v)^\mathsf{T}(w-v) + \frac{1}{2}L\|w-v\|^2 \end{aligned}$$

□

The lemma hints that the convergence is not only depending on the learning rate γ_t but also on the smoothness of the gradients.

Theorem 2.1.1. *Let $F\colon \mathbb{R}^d \to \mathbb{R}$ be continuously differentiable and let the gradient $\nabla F\colon \mathbb{R}^d \to \mathbb{R}^d$ be L-Lipschitz continuous. Then, the iterates of Stochastic Gradient Descent (SGD, 1) satisfy the following inequality for all $k \in \mathbb{N}$:*

$$\mathbb{E}[F(x_{t+1})] - F(x_t) \leq -\gamma_t \nabla F(x_t)^\mathsf{T} \mathbb{E}_{\Xi_t}[\Delta_t] + \frac{1}{2}\gamma_t^2 L \mathbb{E}_{\Xi_t}[\|\Delta_t\|^2]$$

where Ξ_t is the set of all random variables $\{i \in S_t : \xi_i\}$.

Proof. Due to the Lipschitz-continuous gradient assumption and Lemma 2.1.1 the first inequality holds. Then we insert Algorithm 1 and take expectation with respect to Ξ_t.

$$\begin{aligned} F(x_{t+1}) - F(x_t) &\leq \nabla F(x_t)^\mathsf{T}(x_{t+1} - x_t) + \frac{1}{2}L\|x_{t+1} - x_t\|^2 \\ &\leq -\gamma_t \nabla F(x_t)^\mathsf{T}\Delta_t + \frac{1}{2}\gamma_t^2 L\|\Delta_t\|^2 \\ \mathbb{E}[F(x_{t+1})] - F(x_t) &\overset{*}{\leq} -\gamma_t \nabla F(x_t)^\mathsf{T}\mathbb{E}_{\Xi_t}[\Delta_t] + \frac{1}{2}\gamma_t^2 L \mathbb{E}_{\Xi_t}[\|\Delta_t\|^2] \end{aligned}$$

□

Because we are minimizing the objective function F, we want that $\mathbb{E}[F(x_{t+1})] < \mathbb{E}[F(x_t)]$. That implies that the RHS of $(*)$ must be negative, thus $\gamma_t \nabla F(x_t)^\mathsf{T}\mathbb{E}_{\Xi_t}[\Delta_t] > \frac{1}{2}\gamma_t^2 L \mathbb{E}_{\Xi_t}[\|\Delta_t\|^2]$. We see that this can be influenced by the sequence γ_t. Therefore, we minimize the problem when we choose γ appropiatly.

2.1.2 ADMM and the Consensus Problem

Another very popular and well-studied algorithm is the "Alternating Direction Method of Multipliers" method (ADMM) [Boy+11]. In contrast to SGD, ADMM takes a problem, formulates an equivalent primal-dual problem and solves it by optimizing over the primal and dual variables in an alternating way. For the actual and exact convergence analysis, we refer to [Boy+11] and [PB+14] for a more general theory on proximal algorithms. Here, we show the motivation for the update rules in ADMM.

Consider the following problem which is very common in computer vision.

Problem 2.1.2.

$$\min_{u \in \mathbb{R}^n} F(Au) + G(u)$$

where F and G are convex, lower-semicontinous functions and A is a matrix.

The matrix A can make it difficult to optimize the variable u. However, we can derive an equivalent problem by introducing the auxiliary variables $v \in \mathbb{R}^n$ and adding an equality constraint. Then, we transform the constrained problem into an unconstrained one.

$$\begin{aligned}
\min_{u \in \mathbb{R}^n} F(Au) + G(u) &\Leftrightarrow \min_{u,v \in \mathbb{R}^n} F(v) + G(u) \quad \text{s.t. } Au = v \\
&\Leftrightarrow \min_{u,v \in \mathbb{R}^n} F(v) + G(u) + \delta\{Au - v = 0\} \\
&\Leftrightarrow \min_{u,v \in \mathbb{R}^n} \max_{p \in \mathbb{R}^n} F(v) + G(u) + \langle p, Au - v \rangle
\end{aligned}$$

where $\delta\{\cdot\}$ is the indicator function. In the last step, we introduced the dual variable $p \in \mathbb{R}^n$ that enforces the equality constrained to be true. Adding an additional augmentation term does not change the optimization problem but makes ADMM converge in the end. Let $\tau > 0$ and

$$\min_{u,v \in \mathbb{R}^n} \max_{p \in \mathbb{R}^n} \mathcal{L}_\tau(u, v; p) := F(v) + G(u) + \{p, Au - v\} + \frac{\tau}{2}\|Au - v\|^2$$

We call $\mathcal{L}_\tau(u, v; p)$ the augmented Lagrangian. Optimizing over this term solves our original Problem 2.1.2.

Each iteration of ADMM performs an exact, minimization step over u and v and an inexact gradient ascend step over p. By choosing the right τ, we converge to a

solution.

Algorithm 2: Alternating Direction Method of Multipliers

Choose initial $u^0, v^0, p^0 \in \mathbb{R}^n$
for $t = 0, 1, \ldots, T-1$ **do**
  Compute primal variable $u^{t+1} \in \arg\min_u G(u) + \langle p^t, Au \rangle + \frac{\tau}{2}\|Au - v^t\|^2$
  Compute primal variable $v^{t+1} \in \arg\min_v F(v) - \langle p^t, v \rangle + \frac{\tau}{2}\|Au^{t+1} - v\|^2$
  Compute dual variable $p^{t+1} = p^t + \tau(Au^{t+1} - v^{t+1})$
end

The solutions for the variables u, v and p highly depend on the difficulty of the original objective functions F and G. Again, by the description of Algorithm 2 it is not easy to see how one would derive a distributed version. To do this, we first need to ensure our original problem can be reformulated as the Consensus Problem 2.1.3.

Problem 2.1.3 (Consensus Problem)**.**

$$\min_{u \in \mathbb{R}^n} f(x) := F(u) + \sum_{k=1}^{K} G_k(u)$$

Assuming we have K workers, our goal is to separate the problem over the function G such that each worker $1 \leq k \leq K$ can optimize over a specific G_k on its own. Then, each worker returns a solution u_k and the master aggregates all models and distributes the work again. By introducing auxiliary and dual variables we transform the problem similarly to before.

$$\begin{aligned}
&\min_{u \in \mathbb{R}^n} f(x) := F(u) + \sum_{k=1}^{K} G_k(u) \\
\Leftrightarrow &\min_{u_0, \{u_k\} \in \mathbb{R}^n} F(u_0) + \sum_{k=1}^{K} G_k(u_k) \quad \text{s.t. } u_0 = u_k \quad \forall 1 \geq k \geq K \\
\Leftrightarrow &\min_{u_0, \{u_k\} \in \mathbb{R}^n} \max_{\{p_k\} \in \mathbb{R}^n} F(u_0) + \sum_{k=1}^{K} G_k(u_k) + \sum_{k=1}^{K} \langle p_k, u_0 - u_k \rangle \\
\Leftrightarrow &\min_{u_0, \{u_k\} \in \mathbb{R}^n} \max_{\{p_k\} \in \mathbb{R}^n} F(u_0) + \sum_{k=1}^{K} G_k(u_k) + \sum_{k=1}^{K} \langle p_k, u_0 - u_k \rangle + \frac{\tau}{2} \sum_{k=1}^{K} \|u_0 - u_k\|^2 \\
\Leftrightarrow &\min_{u_0, \{u_k\} \in \mathbb{R}^n} \max_{\{p_k\} \in \mathbb{R}^n} \mathcal{L}_\tau(u_0, \{u_k\}; \{p_k\})
\end{aligned}$$

and $\mathcal{L}_\tau$ is the augmented Lagrangian

$$\mathcal{L}_\tau(u_0, \{u_k\}; \{p_k\}) := F(u_0) + \sum_{k=1}^{K} G_k(u_k) + \sum_{k=1}^{K} \langle p_k, u_0 - u_k \rangle + \frac{\tau}{2} \sum_{k=1}^{K} \|u_0 - u_k\|^2$$

The iteration of ADMM changes to

Algorithm 3: ADMM for Consensus Problem

Choose initial $u_0^0, \{u_k^0\}_{k=1}^K, \{p_k^0\}_{k=1}^K$
for $t = 0, 1, \ldots, T-1$ **do**
  $u_0^{t+1} \in \arg\min_u F(u_0) + \sum_{k=1}^K \langle p_k^t, u_0 \rangle + \frac{\tau}{2} \sum_{k=1}^K \|u_0 - u_k^t\|^2$
  for $k = 1, 2, \ldots, K$ **do**
    $u_k^{t+1} \in \arg\min_{u_k} G(u_k) - \langle p_k^t, u_k \rangle + \frac{\tau}{2} \sum_{k=1}^K \|u_0^{t+1} - u_k\|^2$
    $p_k^{t+1} = p_k^t + \tau(u_0^{t+1} - u_k^{t+1})$
  end
end

Now, Algorithm 3 could be distributed by using a star-shaped communication network. While the u_0 "global" update needs the information from *all* u_k and p_k "local" updates, the u_k and p_k updates *only* need their previous values and u_0. That means, we can interpret the u_0 update as a way to aggregate the results of all workers and the communication happens only between the master node and a worker node.

2.2 Asynchronous Stochastic Gradient Descent

To begin with, we present the Asynchronous Stochastic Gradient Descent algorithm "AsySG-con" [Lia+15] which is a distributed extension to SGD [RM51]. We show its convergence theory and also reproduce some of its results from the paper later in Section 3.1. Proving their main Theorem 2.2.1 gives us a first insight on how to work with delayed information.

Like many asynchronous algorithms, the idea behind AsySG-con is to parallelize the work onto different machines or "workers" within a network and collect their results as soon as possible, even if not all workers have finished. In specific, we distribute the calculation of the gradient and let them run in parallel. Then, a master machine collects all gradients, aggregates them and updates the optimization variable x_k to x_{k+1}. After that, it distributes the new variable x_{k+1} to all workers over the network again and a new iteration begins.

We start by recalling Problem 2.1.1. Here, let F be smooth but possibly nonconvex.

$$\min_{x \in \mathbb{R}^n} f(x) := \mathbb{E}_{\xi}[F(x; \xi)]$$

We claim that the following algorithm can solve our problem, i.e. it converges to a solution. Let K be the number of maximum iterations, M be the number of the batch size, $\{\gamma_k\}_{k=1}^K$ be the learning rates for each iteration and $\tau_{k,m}$ be a delay for a given iteration and data point within a batch.

Algorithm 4: Asynchronous Stochastic Gradient (AsySG-con)

1 Choose initial x_0
2 **for** $k = 0, 1, \ldots, K-1$ **do**
3 | Select mini-batch size M training samples $\xi_{k,1}, \ldots, \xi_{k,M}$
4 | Compute $x_{k+1} \leftarrow x_k - \gamma_k \sum_{m=1}^M G(x_{k-\tau_{k,m}}; \xi_{k,m})$
5 **end**

We implement this algorithm for our experiments in Section 3.1 to validate the theory. In Section 1.3 we argued that often algorithms differ between their theoretical description and their actual implementation. Here, we see the first example. Obviously, the description does not mention any worker or master nodes. But, due to the introduced delays $\tau_{k,m}$, it actually models many real-world implementations. It does enforce a star-shaped topology due to the aggregation step in line 4.

Each iteration k performs a gradient descent update on x_k using "old" gradient information based on M training samples. Depending on how we implement the algorithm, we end up with different values for $\tau_{k,m}$.

To prove convergence, the authors in [Lia+15] make several assumptions

Assumption 2.2.1.

- ***(Unbiased Gradient):*** *The stochastic gradients $G(x; \xi) := \nabla F(x; \xi)$ are unbiased, i.e.*

$$\nabla f(x) = \mathbb{E}_{\xi}[G(x; \xi)]$$

- ***(Bounded Variance):*** *The variance of the stochastic gradients $\mathrm{Var}[G(x; \xi)]$ are bounded, i.e.*

$$\mathrm{Var}[G(x; \xi)] = \mathbb{E}_{\xi}[\|G(x; \xi) - \nabla f(x)\|^2] \leq \sigma^2$$

- ***(Lipschitz Continuous Gradients):*** *The gradients $\nabla f(\cdot)$ are Lipschitz continuous, i.e. for all $x, y \in \mathbb{R}^n$*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

- ***(Indepencence):*** *The random variables $\{\xi_{k,m}\}_{k=0,m=1}^{k=K,m=M}$ are independent to each other*
- ***(Bounded Delay):*** *The delay variables $\tau_{k,m}$ are bounded, i.e.*

$$\max_{k,m} \tau_{k,m} \leq T$$

We would like to highlight the last "bounded delay" assumption. It implies that at any point our gradients are based on information not older than T iterations. In our experiments later, we will take this assumption to the extreme by *always* picking the highest delay.

Finally, we introduce the main convergence theorem for Algorithm 4.

Theorem 2.2.1. *Assume that Assumption 2.2.1 is true and that the step size $\{\gamma_k\}_{k=1}^K$ satisfies for all $k = 1, 2, \ldots$*

$$LM\gamma_k + 2L^2M^2T\gamma_k \sum_{t=1}^{T} \gamma_{k+t} \leq 1$$

Then, we have the following convergence,

$$\frac{1}{\sum_{k=1}^K \gamma_k} \sum_{k=1}^{K} \gamma_k \mathbb{E}[\|\nabla f(x_k)\|] \leq \frac{2(f(x_1) - f(x^*)) + \sum_{k=1}^K \left(\gamma_k^2 ML + 2L^2M^2\gamma_k \sum_{j=k-T}^{k-1} \gamma_j^2\right)\sigma^2}{M\sum_{k=1}^K \gamma_k}$$

where x^* denotes the optimal solution for Problem 2.1.1 and $\mathbb{E}(\cdot)$ takes the expectation with respect to all random variables in Algorithm 4.

There are several things to point out first before proving the theorem. First, the learning rates γ_k are depending on the Lipschitz constant L, the batch size M and the maximum delay of T to ensure convergence. In addition, γ_k must not be too large to fulfill the inequality. Second, the convergence rate differs from many other optimization algorithms. Because we are dealing with delayed gradients, we cannot ensure a decrease in the objective after each iteration. Sometimes, we might notice an increase in the objective function due to an old gradient. However, the convergence rate here states that *in average* the gradients of the objective function are smaller than a constant value.

Proof. First, we establish an upper bound for the expected difference between the current function value $f(x_k)$ at iteration k and the function value $f(x_{k+1})$ after applying Algorithm 4 at iteration $k+1$. Then, by using a telescopic sum we upperbound the expected value function difference between iteration $k=1$ and $k=K+1$ and Theorem 2.2.1 follows.

Due to the L-Lipschitz continous gradients in Assumtion 2.2.1 and because of Algorithm 4, following inequality holds.

$$\begin{aligned}
f(x_{k+1}) - f(x_k) &\leq \langle \nabla f(x_k), x_{k+1} - x_k \rangle + \frac{L}{2}\|x_{k+1} - x_k\|^2 \\
&= -\left\langle \nabla f(x_k), \gamma_k \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\rangle + \frac{\gamma_k^2 L}{2} \left\| \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\|^2 \\
&= -M\gamma_k \left\langle \nabla f(x_k), \frac{1}{M} \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\rangle \\
&\quad + \frac{\gamma_k^2 L}{2} \left\| \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\|^2
\end{aligned}$$

By taking the expected value with respect to the random variables $\xi_{k,*} := \{\xi_{k,1}, \ldots, \xi_{k,M}\}$, we have

$$\begin{aligned}
\mathbb{E}_{\xi_{k,*}}[f(x_{k+1}) - f(x_k)] &\leq \mathbb{E}_{\xi_{k,*}}\left[-M\gamma_k \left\langle \nabla f(x_k), \frac{1}{M} \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\rangle \right. \\
&\qquad \left. + \frac{\gamma_k^2 L}{2} \left\| \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\|^2 \right] \\
\mathbb{E}_{\xi_{k,*}}[f(x_{k+1})] - f(x_k) &\leq -M\gamma_k \left\langle \nabla f(x_k), \frac{1}{M} \sum_{m=1}^{M} \nabla f(x_{k-\tau_{k,m}}) \right\rangle \\
&\qquad + \frac{\gamma_k^2 L}{2} \mathbb{E}_{\xi_{k,*}} \left[\left\| \sum_{m=1}^{M} G(x_{k-\tau_{k,m}}; \xi_{k,m}) \right\|^2 \right]
\end{aligned}$$

Because of the fact $\langle a, b\rangle = \frac{1}{2}\left(\|a\|^2 + \|b\|^2 - \|a-b\|^2\right)$, we further derive

$$\begin{aligned}
&\mathbb{E}_{\xi_{k,*}}[f(x_{k+1})] - f(x_k) \\
&\leq -\frac{M\gamma_k}{2}\left(\|\nabla_f(x_k)\|^2 + \left\|\frac{1}{M}\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2 - \underbrace{\left\|\nabla_f(x_k) - \frac{1}{M}\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2}_{T_1}\right) \\
&\quad + \frac{\gamma_k^2 L}{2}\underbrace{\mathbb{E}_{\xi_{k,*}}\left[\left\|\sum_{m=1}^{M} G(x_{k-\tau_{k,m}};\xi_{k,m})\right\|^2\right]}_{T_2}
\end{aligned}$$

Now we upper bound T_2. Notice that the following terms vanish due to our unbiased gradients assumption.

$$\begin{aligned}
&\mathbb{E}_{\xi_{k,*}}\left[\left\langle\sum_{m=1}^{M}\left(G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}})\right), \sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\rangle\right] \\
&= \left\langle\sum_{m=1}^{M}\left(\underbrace{\mathbb{E}_{\xi_{k,*}}\left[G(x_{k-\tau_{k,m}};\xi_{k,m})\right]}_{\nabla f(x_{k-\tau_{k,m}})} - \nabla f(x_{k-\tau_{k,m}})\right), \sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\rangle \\
&= 0
\end{aligned}$$

and

$$\begin{aligned}
&\mathbb{E}_{\xi_{k,*}}\sum_{1\leq m<m'\leq M}\left\langle G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}}), G(x_{k-\tau_{k,m'}};\xi_{k,m'}) - \nabla f(x_{k-\tau_{k,m'}})\right\rangle \\
&= \mathbb{E}_{\xi_{k,*}}\sum_{1\leq m<m'\leq M}\left\langle\underbrace{\mathbb{E}_{\xi_{k,m}}\left[G(x_{k-\tau_{k,m}};\xi_{k,m})\right]}_{\nabla f(x_{k-\tau_{k,m}})} - \nabla f(x_{k-\tau_{k,m}}), G(x_{k-\tau_{k,m'}};\xi_{k,m'}) - \nabla f(x_{k-\tau_{k,m'}})\right\rangle \\
&= 0
\end{aligned}$$

That gives us for T_2:

$$
\begin{aligned}
T_2 &= \mathbb{E}_{\xi_{k,*}}\left[\left\|\sum_{m=1}^{M} G(x_{k-\tau_{k,m}};\xi_{k,m})\right\|^2\right] \\
&= \mathbb{E}_{\xi_{k,*}}\left[\left\|\sum_{m=1}^{M}\Big(G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}})\Big) + \sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2\right] \\
&= \mathbb{E}_{\xi_{k,*}}\left[\left\|\sum_{m=1}^{M}\Big(G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}})\Big)\right\|^2\right] + \left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2 \\
&\quad + \underbrace{2\,\mathbb{E}_{\xi_{k,*}}\left[\left\langle\sum_{m=1}^{M}\Big(G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}})\Big), \sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\rangle\right]}_{=0} \\
&= \mathbb{E}_{\xi_{k,*}}\left[\sum_{m=1}^{M}\left\|G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}})\right\|^2\right] + \left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2 \\
&\quad + \underbrace{2\,\mathbb{E}_{\xi_{k,*}}\sum_{1\le m<m'\le M}\Big\langle G(x_{k-\tau_{k,m}};\xi_{k,m}) - \nabla f(x_{k-\tau_{k,m}}), G(x_{k-\tau_{k,m'}};\xi_{k,m'}) - \nabla f(x_{k-\tau_{k,m'}})\Big\rangle}_{=0} \\
&\le M\sigma^2 + \left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2
\end{aligned}
$$

Now, we bound T_1:

$$
\begin{aligned}
T_1 &= \left\|\nabla f(x_k) - \frac{1}{M}\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2 = \frac{1}{M^2}\left\|\sum_{m=1}^{M}\Big(\nabla f(x_k) - \nabla f(x_{k-\tau_{k,m}})\Big)\right\|^2 \\
&\le \frac{1}{M}\sum_{m=1}^{M}\underbrace{\left\|\nabla f(x_k) - \nabla f(x_{k-\tau_{k,m}})\right\|^2}_{\le L^2\|x_k - x_{k-\tau_{k,m}}\|^2} \le L^2\frac{1}{M}\sum_{m=1}^{M}\|x_k - x_{k-\tau_{k,m}}\|^2 \\
&= L^2\max_{1\le k\le M}\|x_k - x_{k-\tau_{k,m}}\|^2 \\
&= L^2\|x_k - x_{k-\tau_{k,\mu}}\|^2
\end{aligned}
$$

where we set $\mu \coloneqq \arg\max_{1\le k\le M} \|x_k - x_{k-\tau_{k,\mu}}\|^2$. Next,

$$
\begin{aligned}
T_1 &\le L^2 \|x_k - x_{k-\tau_{k,\mu}}\|^2 \\
&= L^2 \left\| \sum_{j=k-\tau_{k,\mu}}^{k-1} (x_{j+1} - x_j) \right\|^2 \\
&= L^2 \left\| \sum_{j=k-\tau_{k,\mu}}^{k-1} \gamma_j \sum_{m=1}^{M} G(x_{j-\tau_{j,m}}; \xi_{j,m}) \right\|^2 \\
&= L^2 \left\| \sum_{j=k-\tau_{k,\mu}}^{k-1} \gamma_j \sum_{m=1}^{M} \Big(G(x_{j-\tau_{j,m}}; \xi_{j,m}) - \nabla f(x_{j-\tau_{j,m}}) \Big) + \sum_{j=k-\tau_{k,\mu}}^{k-1} \gamma_j \sum_{m=1}^{M} \nabla f(x_{j-\tau_{j,m}}) \right\|^2 \\
&\le 2L^2 \left(\underbrace{\left\| \sum_{j=k-\tau_{k,\mu}}^{k-1} \gamma_j \sum_{m=1}^{M} \Big(G(x_{j-\tau_{j,m}}; \xi_{j,m}) - \nabla f(x_{j-\tau_{j,m}}) \Big) \right\|^2}_{T_3} + \underbrace{\left\| \sum_{j=k-\tau_{k,\mu}}^{k-1} \gamma_j \sum_{m=1}^{M} \nabla f(x_{j-\tau_{j,m}}) \right\|^2}_{T_4} \right)
\end{aligned}
$$

Take expectation for T_3 with respect to $\hat{\xi} \coloneqq \{\xi_{j,M} \text{ for } j \in \{k-\tau_{k,\mu}, \dots, K-1\}\}$. We also

introduce the function $S(j)$ to ease notation.

$$\begin{aligned}
\mathbb{E}_{\hat{\xi}}[T_3] &= \mathbb{E}_{\hat{\xi}}\left[\left\|\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j\underbrace{\sum_{m=1}^{M}\Big(G(x_{j-\tau_{j,m}};\xi_{j,m})-\nabla f(x_{j-\tau_{j,m}})\Big)}_{S(j)}\right\|^2\right] \\
&= \mathbb{E}_{\hat{\xi}}\left[\sum_{j=k-\tau_{k,\mu}}^{k-1}\|S(j)\|^2\right] + 2\mathbb{E}_{\hat{\xi}}\left[\sum_{k-1\geq j''>j'\geq k-\tau_{k,\mu}}\underbrace{\langle S(j''),S(j')\rangle}_{=0}\right] \\
&= \mathbb{E}_{\hat{\xi}}\left[\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\left\|\sum_{m=1}^{M}\Big(G(x_{j-\tau_{j,m}};\xi_{j,m})-\nabla f(x_{j-\tau_{j,m}})\Big)\right\|\right] \\
&= \mathbb{E}_{\hat{\xi}}\left[\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\sum_{m=1}^{M}\underbrace{\left\|G(x_{j-\tau_{j,m}};\xi_{j,m})-\nabla f(x_{j-\tau_{j,m}})\right\|^2}_{\leq\sigma^2}\right] \\
&\leq M\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\sigma^2 \leq M\sum_{j=k-T}^{k-1}\gamma_j^2\sigma^2
\end{aligned}$$

Also, take expectation for T_4 with respect to $\hat{\xi}$.

$$\mathbb{E}_{\hat{\xi}}[T_4] = \mathbb{E}_{\hat{\xi}}\left[\left\|\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right] \leq T\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\mathbb{E}_{\hat{\xi}}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right]$$

Taking the full expectation for T_1 by using $\mathbb{E}_{\hat{\xi}}[T_3]$ and $\mathbb{E}_{\hat{\xi}}[T_4]$ gives us a useful term for the following step.

$$\mathbb{E}[T_1] \leq 2L^2\left(M\sum_{j=k-T}^{k-1}\gamma_j^2\sigma^2 + T\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right]\right)$$

For the second last step, we derive an upper bound for the difference between $f(x_k)$ and the total expected value of $f(x_{k+1})$. Then, we use this upper bound for deriving

the upperbound for $\mathbb{E}[f(x_{K+1})] - f(x_1)$.

$$\begin{aligned}\mathbb{E}[f(x_{k+1})] - f(x_k) \leq & -\frac{M\gamma_k}{2}\left(\mathbb{E}[\|\nabla f(x_k)\|^2]) + \frac{1}{M^2}\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_k,m})\right\|^2\right]\right) \\ & - L^2M\gamma_k\left(M\sum_{j=k-T}^{k-1}\gamma_j^2\sigma^2 + T\sum_{j=k-\tau_{k,\mu}}^{k-1}\gamma_j^2\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right]\right) \\ & + \frac{\gamma_k^2 L}{2}\left(M\sigma^2 + \mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2\right]\right)\end{aligned}$$

Use the upper bounds for $\mathbb{E}[T_1]$ and $\mathbb{E}[T_2]$ for the second and third sum respectively.

$$\begin{aligned}\mathbb{E}[f(x_{k+1})] - f(x_k) \leq & -\frac{M\gamma_k}{2}\mathbb{E}[\|\nabla f(x_k)\|^2] + \left(\frac{\gamma_k^2 L}{2} - \frac{\gamma_k}{2M}\right)\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2\right] \\ & + \left(\frac{\gamma_k^2 ML}{2} + L^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2 \\ & + L^2MT\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right]\end{aligned}$$

Next,

$$\begin{aligned}
\mathbb{E}[f(x_{K+1})] - f(x_1) &= \mathbb{E}[(f(x_2) - f(x_1)) + (f(x_3) - f(x_2)) + \cdots + (f(x_K - f(x_{K+1}))] \\
&\leq -\frac{M}{2}\sum_{k=1}^{K}\gamma_k \mathbb{E}[\|\nabla f(x_k)\|^2] \\
&\quad + \sum_{k=1}^{K}\left(\frac{\gamma_k^2 L}{2} - \frac{\gamma_k}{2M}\right)\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau k,m})\right\|^2\right] \\
&\quad + \sum_{k=1}^{K}\left(\frac{\gamma_k^2 ML}{2} + L^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2 \\
&\quad + L^2MT\sum_{k=1}^{K}\left(\gamma_k\sum_{j=K-T}^{k-1}\gamma_j^2\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{j-\tau_{j,m}})\right\|^2\right]\right) \\
&= -\frac{M}{2}\sum_{k=1}^{K}\gamma_k\mathbb{E}[\|\nabla f(x_k)\|^2] + \sum_{k=1}^{K}\left(\frac{\gamma^2 ML}{2} + L^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2 \\
&\quad + \sum_{k=1}^{K}\left(\gamma_k^2\left(\frac{L}{2} + L^2MT\sum_{\kappa=1}^{T}\gamma_{k+\kappa}\right) - \frac{\gamma_k}{2M}\right)\mathbb{E}\left[\left\|\sum_{m=1}^{M}\nabla f(x_{k-\tau_{k,m}})\right\|^2\right]
\end{aligned}$$

The last term is negative because

$$\begin{aligned}
LM\gamma_k + 2L^2M^2T\gamma_k\sum_{\kappa=1}^{T}\gamma_{k+\kappa} \leq 1 &\Leftrightarrow 2M\left(\frac{L\gamma_k^2}{2} + L^2MT\gamma_k^2\sum_{\kappa=1}^{T}\gamma_{k+\kappa}\right) \leq \gamma_k \\
&\Leftrightarrow \gamma_k^2\left(\frac{L}{2} + L^2MT\sum_{k=1}^{T}\gamma_{k+\kappa}\right) \leq \frac{\gamma_k}{2M}
\end{aligned}$$

Finally, rearranging this term concludes the proof.

$$\begin{aligned}
&\mathbb{E}[f(x_{K+1})] - f(x_1) \\
&\leq -\frac{M}{2}\sum_{k=1}^{K}\gamma_k\mathbb{E}[\|\nabla f(x_k)\|^2] + \sum_{k=1}^{K}\left(\frac{\gamma_k^2 ML}{2} + L^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_j^2\right)\sigma^2 \\
\Leftrightarrow\quad &\frac{1}{\sum_{k=1}^{K}\gamma_k}\sum_{k=1}^{K}\gamma_k\mathbb{E}[\|\nabla f(x_k)\|^2] \\
&\leq \frac{2(f(x_1) - \mathbb{E}f(x_{K+1})) + \sum_{k=1}^{K}\left(\gamma_k^2 ML + 2L^2M^2\gamma_k\sum_{j=k-T}^{k-1}\gamma_k^2\right)\sigma^2}{M\sum_{k=1}^{K}\gamma_k}
\end{aligned}$$

□

2.3 Asynchronous Block Coordinate Descent

A related algorithm to gradient descent is Block Coordinate Descent (BCD). Basically, each iteration updates only a subset of components of the optimizer $x \in \mathbb{R}^N$ instead of all. Convergence of (synchronous) BCD has been established in [Tse01] and [Liu+14] introduced an asynchronous version of it. Here, we present the analysis and convergence proof given in [SHY17] which could give us an important tool for the convergence theory in Section 2.4.3.

To start with, consider this very general problem, where f is possibly nonconvex but differentiable and its gradients are L-Lipschitz continuous.

Problem 2.3.1.

$$\min_{x \in \mathbb{R}^N} f(x) = f(x_1, \dots, x_N)$$

To solve Problem 2.3.1, for each iteration k the algorithm picks a subset $I_k \subseteq \{1, \dots, N\}$ and performs an update for each $i_k \in I_k$.

Algorithm 5: Asynchronous Block Coordinate Descent

```
for k = 0, 1, ... do
    Pick I_k ⊆ {1, ..., N}
    for i_k ∈ I_k do
        Update x_{i_k}^{k+1} = x_{i_k}^k − (γ_k / L) ∇_{i_k} f(x̂^k)
    end
    for i_k ∉ I_k do
        x_{i_k}^{k+1} = x_{i_k}^k
    end
end
```

where γ_k is the current learning rate and $\hat{x}^k$ is a possibly delayed optimizer. Formally, $\hat{x}^k$ is defined as

$$\hat{x}^k = \left(x_1^{k-j(k,1)}, \dots, x_N^{k-j(k,N)} \right)$$

and $j(k, n)$ denotes a bounded delay for a given iteration and dimension. we have seen a very similar algorithm in the previous Section 2.2 that also uses delays as a framework. Also, we denote the maximum delay across all dimensions for an iteration k as

$$j(k) := \max_{1 \leq i \leq N} \{ j(k, i) \}$$

Before we start with the main theorem and its proof, we also introduce Δ^k that notates the difference between before and after of an Async-BCD iteration.

$$\Delta^k := x^{k+1} - x^k = -\frac{\gamma_k}{L} \nabla_{i_k} f(\hat{x}^k)$$

Next, we introduce the concept of a Lyapunov function. The main idea of the proof is then to show a decrease within the Lyapunov function as long as one picks a sufficient learning rate. Then, we show that this implies convergence.

$$\xi_k := f(x^k) + \frac{L}{2\varepsilon} \sum_{i=k-\tau}^{k-1} (i - (k-\tau) + 1) \|\Delta^i\|_2^2$$

Last but not least, we need to ensure that each block is updated frequently enough. To do that, we assume the following "essentially cyclic" update rule: There exists an $N' \geq N$ such that each block $i \in \{1, \ldots, N\}$ is updated at least once within N' iterations, i.e. for each $t \in \mathbb{N}_{\geq 0}$ there exists an integer $K(i,t) \in \{tN', tN'+1, \ldots, (1+t)N'-1\}$ such that $i_{K(i,t)} = i$. Notice that this update rule generalizes many existing pick-strategies, like block picks in a sequential or random order.

Let's start the analysis by proving a decrease within the Lyapunov function when one picks the right learning rate γ.

Lemma 2.3.1. *Let f be a possibly nonconvex function with L-Lipschitz continuous gradients and be lower bounded. Then, if the step size is*

$$\gamma_k \equiv \gamma := \frac{2c}{2\tau + 1}$$

for arbitrarty fixed $0 < c < 1$ and the sequence $(x^k)_{k \geq 0}$ is generated by the async-BCD Algorithm 5 with bounded delay τ, we derive that

$$\xi_k - \xi_{k+1} \geq \frac{1}{2} \left(\frac{1}{\gamma} - \frac{1}{2} - \tau \right) L \|\Delta^k\|_2^2$$

which implies,

$$\lim_k \|\Delta^k\|_2 = 0$$

Proof. If we insert γ into the term $\frac{1}{\gamma} - \frac{1}{2} - \tau$ then we see that the term is strictly positive. That means, that that for each iteration k, we see a decrease in the Lyapunov function, i.e. $\xi_{k+1} < \xi_k$.

$$\frac{1}{\gamma} - \frac{1}{2} - \tau = \frac{2\tau + 1}{2c} - \frac{1c}{2c} - \frac{2\tau c}{2c} = \frac{1}{2}(2\tau(1-c) + (1-c)) > 0$$

Also, in each iteration we only update one coordinate, therefore we have

$$\Delta^k := x^{k+1} - x^k = \begin{pmatrix} 0 & \cdots & \Delta^k_{i_k} & \cdots & 0 \end{pmatrix}^\mathsf{T}$$

Thus, we derive the next equality by using the update rule in Algorithm 5 line 4 in addition.

$$-\langle \Delta^k, \nabla f(\hat{x}^k) \rangle = -\langle \Delta^k_{i_k}, \nabla_{i_k} f(\hat{x}^k) = -\langle \Delta^k_{i_k}, -\frac{L}{\gamma_k} \Delta^k_{i_k} \rangle = \frac{L}{\gamma} \|\Delta^k_{i_k}\|^2$$

Because ∇f is L-Lipschitz continuous, we get the next inequalities.

$$\begin{aligned}
f(x^{k+1}) - f(x^k) &\leq \langle \nabla f(x^k), \Delta^k \rangle + \frac{L}{2} \|\Delta^k\|^2 \\
&= \langle \nabla f(x^k), \Delta^k \rangle + \frac{L}{2} \|\Delta^k\|^2 - \langle \Delta^k, \nabla f(\hat{x}^k) \rangle - \frac{L}{\gamma} \|\Delta^k\|^2 \\
&= \langle \nabla f(x^k) - \nabla f(\hat{x}^k), \Delta^k \rangle + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&\overset{CS}{\leq} \|\nabla f(x^k) - \nabla f(\hat{x}^k)\| \|\Delta^k\| + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&\leq L \|x^k - \hat{x}^k\| \|\Delta^k\| + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&\overset{TS}{\leq} L \sum_{i=k-\tau}^{k-1} \|\Delta^i\| \|\Delta^k\| + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&\overset{PP}{\leq} L \sum_{i=k-\tau}^{k-1} \left(\frac{1}{2\epsilon} \|\Delta^i\|^2 + \frac{\epsilon}{2} \|\Delta^k\|^2 \right) + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&= \frac{L}{2\epsilon} \sum_{i=k-\tau}^{k-1} \|\Delta^i\|^2 + \tau \frac{\epsilon}{2} \|\Delta^k\|^2 + \left(\frac{L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2 \\
&= \frac{L}{2\epsilon} \sum_{i=k-\tau}^{k-1} \|\Delta^i\|^2 + \left(\frac{(\tau\epsilon + 1)L}{2} - \frac{L}{\gamma} \right) \|\Delta^k\|^2
\end{aligned}$$

In (CS) we used the well known Cauchy-Schwarz-inequality and in (PP) the "Peter-Paul" fact, namely

$$\langle x, y \rangle \leq \frac{1}{2\epsilon} \|x\|^2 + \frac{\epsilon}{2} \|y\|^2$$

which can be seen by rearranging $0 \leq \|x' - y'\|^2$ and setting $x := \frac{x'}{\sqrt{\epsilon}}$ and $y := \sqrt{\epsilon}y$. Last but not least, in (TS) we used the telescopic sum

$$\begin{aligned}\|x^k - \hat{x}^k\| &= \|x^k - x^{k-1} + x^{k+1} - x^{k-1} + \cdots + x^{k-\tau+1} - x^{k-\tau}\| \\ &\leq \sum_{i=k-\tau}^{k-1} \|x^{i+1} - x^i\| = \sum_{i=k-\tau}^{k-1} \|\Delta^i\|\end{aligned}$$

We are now in a position where we can upper bound the difference within the Lyapunov-functions between two iterations k and $k+1$.

$$\begin{aligned}\xi_k - \xi_{k+1} &= f(x^k) - f(x^{k+1}) + \frac{L}{2\epsilon} \sum_{i=k-\tau}^{k-1} (i - (k-\tau) + 1)\|\Delta^i\|^2 \\ &\quad - \frac{L}{2\epsilon} \sum_{i=k+1-\tau}^{k-1} (i - (k-\tau))\|\Delta^i\|^2 - \frac{L}{2\epsilon}\tau\|\Delta^k\|^2 \\ &= f(x^k) - f(x^{k+1}) + \frac{L}{2\epsilon} \sum_{k-\tau}^{k-1} \|\Delta^i\|^2 - \frac{L}{2\epsilon}\tau\|\Delta^k\|^2 \\ &\geq -\frac{L}{2\epsilon} \sum_{i=k-\tau}^{k-1} \|\Delta^i\|^2 - \left(\frac{(\tau\epsilon+1)L}{2} - \frac{L}{\gamma}\right)\|\Delta^k\|^2 \\ &\quad + \frac{L}{2\epsilon} \sum_{i=k-\tau}^{k-1} \|\Delta^i\|^2 - \frac{L}{2\epsilon}\tau\|\Delta^k\|^2 \\ &= \frac{L}{\gamma}\|\Delta^k\|^2 - \frac{(\tau\epsilon+1)L}{2}\|\Delta^k\|^2 - \frac{L}{2\epsilon}\tau\|\Delta^k\|^2 \\ &= L\|\Delta^k\|^2 \left(\frac{1}{\gamma} - \frac{\tau\epsilon+1}{2} - \frac{\tau}{2\epsilon}\right)\end{aligned}$$

For the last step, we choose an $\epsilon > 0$ such that

$$\epsilon + \frac{1}{\epsilon} = 1 + \frac{1}{\tau} \left(\frac{1}{\gamma} = \frac{1}{2}\right)$$

e.g. by solving a quadratic system. Rearranging the last inequality and replacing ϵ with our maximum delay τ and learning rate γ gives $\frac{1}{2}\left(\frac{1}{\gamma} - \frac{1}{2} - \tau\right) L\|\Delta^k\|^2$. Because the factor infront of $\|\Delta^k\|^2$ is constant and greater than 0 it follows that $\|\Delta^k\|^2$ must decrease and approach 0.

□

By applying Lemma 2.3.1, we now show that the gradient of $f(x^k)$ approaches 0 when $k \to \infty$.

Theorem 2.3.1. *Assume the conditions in Lemma 2.3.1 for the function f. Then, for the essentially cyclic update block rule, we have*

$$\lim_k \|\nabla f(x^k)\|_2 = 0$$

Proof.

$$\begin{aligned}
\|\nabla_i f(x^k)\| &= \|\nabla_i f(x^k) + \nabla_i f(\hat{x}^{K(i,t)}) - \nabla_i f(\hat{x}^{K(i,t)}) + \nabla_i f(\hat{x}^k) - \nabla_i f(\hat{x}^k)\| \\
&\leq \|\nabla_i f(\hat{x}^{K(i,t)})\| + \|\nabla_i f(x^k) - \nabla_i f(\hat{x}^k)\| + \|\nabla_i f(\hat{x}^k) - \nabla_i f(\hat{x}^{K(i,t)})\| \\
&\leq \|\nabla_i f(\hat{x}^{K(i,t)})\| + L\|x^k - \hat{x}^k\| + L \sum_{j=K(i,t)}^{k-1} \|\hat{x}^{j+1} - \hat{x}^j\|
\end{aligned}$$

To conclude the proof, we need to upper bound each summand. We start with $\|x^k - \hat{x}^k\|$, apply (TS) again and take its limit with respect to k.

$$\begin{aligned}
\|x^k - \hat{x}^k\| &\leq \sum_{i=k-\tau}^{k-1} \|\Delta^i\| \\
\lim_{k\to\infty} \|x^k - \hat{x}^k\| &\leq \lim_{k\to\infty} \sum_{i=k-\tau}^{k-1} \|\Delta^i\|
\end{aligned}$$

Using Lemma 2.3.1 implies that $\lim_{k\to\infty} \|x^k - \hat{x}^k\| = 0$. Next, $\|\hat{x}^{j+1} - \hat{x}^j\|$ can be upper bounded by the triangle inequality:

$$\begin{aligned}
\|\hat{x}^{j+1} - \hat{x}^j\| &= \| - (x^{k+1} - \hat{x}^{k+1}) + x^{k+1} - x^k + x^k - \hat{x}^k\| \\
&\leq \| - (x^{k+1} - \hat{x}^{k+1})\| + \|x^{k+1} - x^k\| + \|x^k - \hat{x}^k\|
\end{aligned}$$

Due to the previous results, each term converge to 0 when we take their limit. For the last term $\|\nabla_i f(\hat{x}^{K(i,t)})\|$ we recall the "Essentially cyclic update rule" and notice that when $k \to \infty$ then also $K(i,t) \to \infty$. Since $i = K(i,t)$,

$$\|\nabla_i f(\hat{x}^{K(i,t)})\| = \|\nabla_{K(i,t)} f(\hat{x}^{K(i,t)})\| = \| - \frac{L}{\gamma} \Delta^{K(i,t)}\| = \frac{L}{\gamma} \|\Delta^K(i,t)\|$$

Taking the limit and applying Lemma 2.3.1 concludes the proof. □

Because we use delays in our algorithm, we cannot prove that *every* iteration of async-BCD provides a decrease within the objective function $f(x)$. This is highly dependent on how we pick the blocks and delays. But, due to the "essentially cyclic" update assumption and the bounded delays, it is at least possible to show that we approach the optimality condition of f.

2.4 Flexible ADMM

In contrast to the previously described algorithms that mainly aggregate gradients, we categorize ADMM under the "model aggregation" type algorithms. Instead of distributing the task of calculating gradients and collecting them, each worker solves a specific subproblem on its own and a master aggregates all solutions of the workers. Then, it defines new subproblems and distributes them to the workers again.

2.4.1 Convergence Analysis

The synchronous ADMM Algorithm 2 requires us to synchronize all workers with the master. To solve this limitation, Hong's Flexible ADMM [HLR16b] removes this constraint. Here, the asynchronicity only refers to "update set", see Section 1.2.3: Each iteration, the master does not necessarily need the results of *all* workers but only of a "flexible" subset of it. [HLR16b] was able to show convergence for nonconvex functions under certain assumptions and the analysis might be able to be extended to support delays in the optimization updates. We give some insights on this extension in Section 2.4.3. For now, we will look more closely into Hong's convergence theory as it provides us yet another proofing technique on how to deal with asynchronous updates.

Consider the regularized consensus problem

Problem 2.4.1.

$$\min \sum_{k=1}^{K} g_k(x_k) + h(x_0)$$
$$\text{s.t. } x_k = x_0 \; \forall k = 1, \dots, K; \; x_0 \in X$$

where g_k is differentiable but not necessarily convex for all k, h is convex but not necessarily differentiable and X is closed.

And, its augmented Lagrangian $L(\{x_k\}, x_0; y)$ with parameter ρ_k affecting the strong-

convexity of the objective functions:

$$L(\{x_k\}, x_0; y) := \sum_{k=1}^{K} g_k(x_k) + h(x_0) + \sum_{k=1}^{K} \langle y_k, x_k - x_0 \rangle + \sum_{k=1}^{K} \frac{\rho_k}{2} \|x_k - x_0\|^2$$

We will show that "Flexible ADMM" can be used for Problem 2.4.1.

Algorithm 6: Flexible ADMM

Initialize x_0^0, $\{x_k^t\}_{k=1}^K$ and $\{y_k^t\}_{k=1}^K$
Choose appropriate $\{\rho_k\}_{k=1}^K$
for $t = 0, 1, \ldots$ **do**
  if $t = 0$ **then**
    $\mathcal{C}^{t+1} = \{0, \ldots, K\}$
  else
    $\mathcal{C}^{t+1} \subseteq \{0, \ldots, K\}$
  end
  if $0 \in \mathcal{C}^{t+1}$ **then**
    $x_0^{t+1} = \arg\min_{x \in X} L(\{x_k^t\}, x_0; y^t)$
  else
    $x_0^{t+1} = x_0^t$
  end
  if $k \neq 0$ *and* $k \in \mathcal{C}^{t+1}$ **then**
    Distribute to worker k
    $x_k^{t+1} = \arg\min_{x_k} g_k(x_k) + \langle y_k^t, x_k - x_0^{t+1} \rangle + \sum_{k=1}^{K} \frac{\rho_k}{2} \|x_k - x_0^{t+1}\|^2$
    $y_k^{t+1} = y_k^t + \rho_k \left(x_k^{t+1} - x_0^{t+1} \right)$
  else
    $x_k^{t+1} = x_k^t$
    $y_k^{t+1} = y_k^t$
  end
end

Notice the update sets $\mathcal{C}$ in the algorithm which affects the type and order of updates. Also, [HLR16b] provided several assumptions to ensure convergence.

Assumption 2.4.1.

- ***(Lipschitz Continuous Gradients):*** *For all k, the gradients $\nabla g_k(\cdot)$ are L_k-Lipschitz continuous, i.e. for all x_k, x_k'*

$$\|\nabla_k g_k(x_k) - \nabla g_k(x_k')\| \leq L_k \|x_k - x_k'\|$$

- ***(Large Enough Penalty Paramter):*** *The parameter ρ_k is chosen large enough such that for all k, the subproblem in line 16 is strongly convex with modulus $\gamma_k(\rho_k)$ and $\rho_k\gamma_k(\rho_k) > 2L_k^2$ and $\rho_k \leq L_k$.*
- ***(Bounded from below):*** *$f(x)$ is lower bounded over X, i.e.*

$$\underline{f} := \min_{x \in X} f(x) > -\infty$$

The most important assumption here is to choose a large enough penalty parameter ρ_k. It allows us to "strongly convexify" the subproblems of the workers. However, this comes with a cost of slower convergence. Again, one must choose a balance between a well-behaved convex problem and fast learning. In practice, it is not always clear how to estimate the right parameter for a given function, so some experimentation might be required.

Similar to asynchronous block coordinate descent, we need to ensure that all local variables contribute to the master *at some point*. We do that by assuming essentially cyclic updates. Namely,

$$\bigcup_{i=1}^{T} \mathcal{C}^{t+i} = \{0, \dots, K\} \quad \forall t$$

Now, we are going to prove three important lemmas that build the foundation for the main Theorem 2.4.1. The first Lemma 2.4.1 upper bounds the difference between the dual variables when we apply Flexible ADMM.

Lemma 2.4.1. *Suppose Assumption 2.4.1 holds. Then, for Algorithm 6 with the essentially cyclic update rule, we have*

$$L_k^2 \|x_k^{t+1} - x_k^t\|^2 \geq \|y_k^{t+1} - y_k^t\|^2 \; \forall k = 1, \dots, K$$

Proof. We write down the optimality condition for the update x_k^{t+1} and combine it with the dual update step $\rho_k(x_k^{t+1} - x_0^{t+1}) = y_k^{t+1} - y_k$:

$$\begin{aligned} 0 &= \nabla g_k(x_k^{t+1}) + y_k^t + \rho_k(x_k^{t+1} - x_0^{t+1}) \\ \Leftrightarrow \nabla g_k(x_k^{t+1}) &= -y_k^{t+1} \end{aligned}$$

Now, we combine it with the first Assumption 2.4.1 which concludes the proof.

$$\|y_k^{t+1} - y_k^t\| = \|\nabla g_k(x_k^t) - \nabla g_k(x_k^{t+1})\| = \|\nabla g_k(x_k^{t+1}) - \nabla g_k(x_k^t)\| \leq L_k \|x_k^{t+1} - x_k^t\|$$

□

The next lemma shows a decrease in the augmented Lagrangian after each iteration t as long as we pick sufficiently large ρ_ks.

Lemma 2.4.2. *For Algorithm 6 with the essentially cyclic update rule, we have*

$$\begin{aligned} &L(\{x_k^{t+1}, x_0^{t+1}; y^{t+1}\}) - L(\{x_k^t, x_0^t; y^t\}) \\ \leq &\sum_{k \in \mathcal{C}^{t+1} \setminus \{0\}} \left(\frac{L_k^2}{\rho_k} - \frac{\gamma_k(\rho_k)}{2} \right) \left\| x_k^{t+1} - x_k^t \right\|^2 - \frac{\gamma}{2} \left\| x_0^{t+1} - x_0^t \right\|^2 \end{aligned}$$

Proof. We start by splitting the difference into three summands, each representing an update within one iteration. Then we upper bound each term.

$$\begin{aligned} L(\{x_k^{t+1}, x_0^{t+1}; y^{t+1}\}) - L(\{x_k^t, x_0^t; y^t\}) = &L(\{x_k^{t+1}, x_0^{t+1}; y^{t+1}\}) - L(\{x_k^{t+1}, x_0^{t+1}; y^t\}) \\ &+ L(\{x_k^{t+1}, x_0^{t+1}; y^t\}) - L(\{x_k^t, x_0^{t+1}; y^t\}) \\ &+ L(\{x_k^t, x_0^{t+1}; y^t\}) - L(\{x_k^t, x_0^t; y^t\}) \end{aligned}$$

We upper bound the first term by inserting the dual variable update rule.

$$\begin{aligned} &L(\{x_k^{t+1}, x_0^{t+1}; y^{t+1}\}) - L(\{x_k^{t+1}, x_0^{t+1}; y^t\}) \\ = &\sum_{k=1}^K \left\langle y_k^{t+1}, x_k^{t+1} - x_0^{t+1} \right\rangle - \sum_{k=1}^K \left\langle y_k^t, x_k^{t+1} - x_0^{t+1} \right\rangle \\ = &\sum_{k=1}^K \left\langle y_k^{t+1} - y_k^t, x_k^{t+1} - x_0^{t+1} \right\rangle \\ = &\sum_{k=1}^K \left\langle y_k^{t+1} - y_k^t, \frac{1}{\rho_k} \left(y_k^{t+1} - y_k^t \right) \right\rangle \\ = &\sum_{k=1}^K \frac{1}{\rho_k} \left\| y_k^{t+1} - y_k^t \right\|^2 \end{aligned}$$

For the second term, we introduce an auxiliary function l_k that is strongly convex with respect to x_k as long as ρ_k has been chosen large enough.

$$l_k(x_k, x_0; y) \coloneqq g_k(x_k) + \langle y_k, x_k - x_0 \rangle + \frac{\rho_k}{2} \|x_k - x_0\|^2$$

Due to the strong convexity with modulus $\gamma_k(\rho_k)$ we get the next inequality.

$$l_k(x_k^{t+1}, x_0; y) - l_k(x_k^t, x_0; y) \leq \left\langle \nabla l_k(x_k^{t+1}, x_0; y), x_k^{t+1} - x_k^t \right\rangle - \frac{\gamma_k(\rho_k)}{2} \left\| x_k^{t+1} - x_k^t \right\|^2$$

We insert l_k into the second term that we want to upper bound. Then we recall that we solve the l_k subproblem exactly which implies that ∇l_k is zero.

$$\begin{aligned}
&L(\{x_k^{t+1}, x_0^{t+1}; y^t\}) - L(\{x_k^t, x_0^{t+1}; y^t\}) \\
&= \sum_{k=1}^{K} \left(l_k(x_k^{t+1}) - l_k(x_k^t) \right) \\
&\leq \sum_{k=1}^{K} \left(\left\langle \underbrace{\nabla l_k(x_k^{t+1}, x_0^{t+1}; y^t)}_{=0}, x_k^{t+1} - x_k^t \right\rangle - \frac{\gamma_k(\rho_k)}{2} \left\| x_k^{t+1} - x_k^t \right\|^2 \right)
\end{aligned}$$

For the last difference, we see that by the assumptions 2.4.1, $L(\{x_k\}, x_0; y)$ is strongly convex with respect to x_0 with modulus γ. Because we solve the x_0 subproblem exactly, we let $0 = \zeta_{x_0}^{t+1} \in \partial L(\{x_k^t\}, x_0^{t+1}; y^t)$. Then, we bound the term.

$$L(\{x_k^t\}, x_0^{t+1}; y^t) - L(\{x_k^t\}, x_0^t; y^t) \leq \langle \underbrace{\zeta_{x_0}^{t+1}}_{=0}, x_0^{t+1} - x_0^t \rangle - \frac{\gamma}{2} \|x_0^{t+1} - x_0^t\|^2$$

Finally, we combine each term

$$\begin{aligned}
&L(\{x_k^{t+1}\}, x_0^{t+1}; y^{t+1}) - L(\{x_k^t\}, x_0^t; y^t) \\
&\leq -\sum_{k=1}^{K} -\frac{\gamma_k(\rho_k)}{2} \|x_k^{t+1} - x_k^t\|^2 + \frac{1}{\rho_k} \|y_k^{t+1} - y_k^t\| - \frac{\gamma}{2} \|x_0^{t+1} - x_0^t\|^2 \\
&\leq \sum_{k=1}^{K} \left(\frac{L_k^2}{\rho_k} - \frac{\gamma_k(\rho_k)}{2} \right) \|x_k^{t+1} - x_k^t\|^2 - \frac{\gamma}{2} \|x_0^{t+1} - x_0^t\|^2
\end{aligned}$$

One can confirm that both summands are strictly negative. This implies that after each iteration, the augmented Lagrangian L decreases. □

The last lemma lower bounds the augmented Lagrangian.

Lemma 2.4.3. *Suppose Assumption 2.4.1 holds. Let $\{\{x_k^t\}, x_0^t, y^t\}$ be generated by Algorithm 6 with the essentially cyclic update rule. Then, the following limit exists and is lower bounded by $\underline{f}$ as defined in Assumption 2.4.1.*

$$\lim_{t \to \infty} L(\{x_k^t, x_0^t; y^t\}) \geq \underline{f}$$

Proof.

$$
\begin{aligned}
& L(\{x_k^{t+1}\}, x_0^{t+1}; y^{t+1}) \\
&= h(x_0^{t+1}) + \sum_{k=1}^{K} \left(g_k(x_k^{t+1}) + \langle y_k^{t+1}, x_k^{t+1} - x_0^{t+1} \rangle + \frac{\rho_k}{2} \|x_k^{t+1} - x_0^{t+1}\|^2 \right) \\
&\overset{1}{=} h(x_0^{t+1}) + \sum_{k=1}^{K} \left(g_k(x_k^{t+1}) + \langle \nabla g_k(x_k^{t+1}), x_0^{t+1} - x_k^{t+1} \rangle + \frac{\rho_k}{2} \|x_k^{t+1} - x_0^{t+1}\|^2 \right) \\
&\geq h(x_0^{t+1}) + \sum_{k=1}^{K} \left(g_k(x_k^{t+1}) + \langle \nabla g_k(x_k^{t+1}), x_0^{t+1} - x_k^{t+1} \rangle + \frac{L_k}{2} \|x_k^{t+1} - x_0^{t+1}\|^2 \right) \\
&\geq h(x_0^{t+1}) + \sum_{k=1}^{K} g_k(x_0^{t+1}) \\
&= f(x_0^{t+1}) \geq \underline{f}
\end{aligned}
$$

In (1) we used Lemma 2.4.1, namely $\langle y_k^{t+1}, x_k^{t+1} - x_0^{t+1} \rangle = \nabla g_k(x_0^{t+1} - x_k^{t+1})$. □

Finally, we state the convergence theorem.

Theorem 2.4.1. *Assume Assupmtion 2.4.1. Then,*

$$\lim_{t \to \infty} \|x_k^{t+1} - x_0^{t+1}\| = 0$$

Proof. Due to Lemma 2.4.2, we have

$$L(\{x_k^{t+1}\}, x_0^{t+1}; y^{t+1}) - L(\{x_k^t\}, x_0^t; y^t) \leq \sum_{k=1}^{K} \left(\frac{L_k^2}{\rho_k} - \frac{\gamma_k(\rho_k)}{2} \right) \|x_k^{t+1} - x_k^t\|^2 - \frac{\gamma}{2} \|x_0^{t+1} - x_0\|^2$$

By Lemma 2.4.3, the limits of $\|x_k^{t+1} - x_k^t\|$ and $\|x_0^{t+1} - x_0^t\|$ must converge go to 0 since the factors in front of them are independent of t. And to conclude the proof, we apply Lemma 2.4.1 and use the dual variable update rule.

$$L_k^2 \underbrace{\|x_k^{t+1} - x_k^t\|}_{\to 0} \geq \underbrace{\|y_k^{t+1} - y_k^t\|}_{\Rightarrow \to 0} = \underbrace{\|\rho_k(x_k^{t+1} - x_0^{t+1})\|}_{\Rightarrow \to 0}$$

□

Once Theorem 2.4.1 is established, the full convergence proof can be derived. For this, we refer to the original paper [HLR16b] since the rest of the proof does not provide any further insights on how to deal with asynchronicity.

2.4.2 Order of Updates

It turns out that the Flexible ADMM Algorithm 6 still converges if we perform its update steps in an arbitrary order, as long as every update happens *at some point in time*. Recall the essentially cyclic update rule:

$$\bigcup_{i=1}^{T} \mathcal{C}^{t+i} = \{0, \dots, K\} \quad \forall t$$

The condition only requires from us that a given update $k \in \{0, \dots, K\}$ happens within an interval T. For example, instead of performing a global aggregation step for node 0 first, it would also be fine to do it in the middle or at the very end of an interval. We only require that the local primal and dual updates always happen together.

2.4.3 Delayed ADMM

Given the previously mentioned algorithms "Asynchronous SGD" and "Asynchronous BCD", it is very natural to question whether we can derive a similar analysis for ADMM as well. Consider the following modification to Flexible ADMM.

Algorithm 7: Delayed ADMM

Initialize x_0^0, $\{x_k^t\}_{k=1}^K$ and $\{y_k^t\}_{k=1}^K$
Choose appropiate $\{\rho_k\}_{k=1}^K$
for $t = 0, 1, \dots$ **do**
  $x_0^{t+1} = \arg\min_{x_0 \in X} L(\{\hat{x}_k^t\}, x_0; \hat{y}^t)$
  for $k = 1, \dots, K$ **do**
    Distribute to worker k
    $x_k^{t+1} = \arg\min_{x_k} g_k(x_k) + \langle y_k^t, x_k - \hat{x}_0^{t+1} \rangle + \sum_{k=1}^K \frac{\rho_k}{2} \|x_k - \hat{x}_0^{t+1}\|^2$
    $y_k^{t+1} = y_k^t + \rho_k \left(x_k^{t+1} - \hat{x}_0^{t+1} \right)$
  end
end

where $\hat{x}^t$ is a delayed variable, not older than τ.

$$\hat{x}^t \in \{x^{t-\tau}, \dots, x^t\}$$

This algorithm describes the "most general" asynchronous case and generalizes Flexible ADMM. However, at this point in time we could not derive a full theoretical analysis for

this case yet. The original idea was to concatenate the global and local *primal* variable and to connect the analysis given in the asynchronous BCD method with the one provided in Flexible ADMM. One might also look into [Hon18] as it shows an analysis for delays but with inexact update rules. Nevertheless, we provide experimental results in Section 3.2 for this algorithm. We believe that is it is worth it to analyze the modification further.

3 Examples and Experiments

To support the provided theory, we perform several experiments on asynchronous SGD and asynchronous ADMM. We vary their hyperparameters affecting their convergence rate and we also look at the effects of delayed information. In specific, we derive a distributed version of an image classification problem and an image segmentation task and we also give details on how we implemented a distributed optimizer using a GPU cluster.

3.1 SGD Experiments

The main goal of this first experiment is to implement the asynchronous distributed SGD algorithm and to simulate different delays and their effect on the convergence rate by using different learning rates. We implement the algorithm that we describe in detail in 2.2 or from [Lia+15].

We consider the MNIST logistic regression problem:

Problem 3.1.1 (MNIST Logistic Regression)**.**

$$\min_{x\in\mathbb{R}^n} \mathbb{E}_{\xi}[F(x;\xi)] = \min_{x\in\mathbb{R}^n} \frac{1}{m}\sum_{i=1}^{m} l(h(\text{image}_i; x), \text{label}_i)$$

where l is the cross-entropy loss function and h is a logistic regression score function parameterized by x. ξ is sampled from a distribution consisting of the 60000 data pairs available in MNIST.

We could choose a different model for h like a neural network. However, we decided to keep the experiment sufficiently simple. To model delays, we store the new variable in a queue for every iteration. In all experiments, we set the batch size to 64. Each batch uses the *same*, constant delay. That way, we can simulate a worst-case scenario where we simulate a huge failure in receiving the data. In addition, it also makes interpreting the graphs easier since there is not any noise in the sampling itself. Each graph shows the cross-entropy loss given by the different learning rates γ for a fixed

constant delay τ. Due to the discussed theory, we expect that decreasing the learning rate γ results in an expected decrease in the loss. Figure 3.1 shows various plots with delays of $1, 10, 100, 1000$ and 10000.

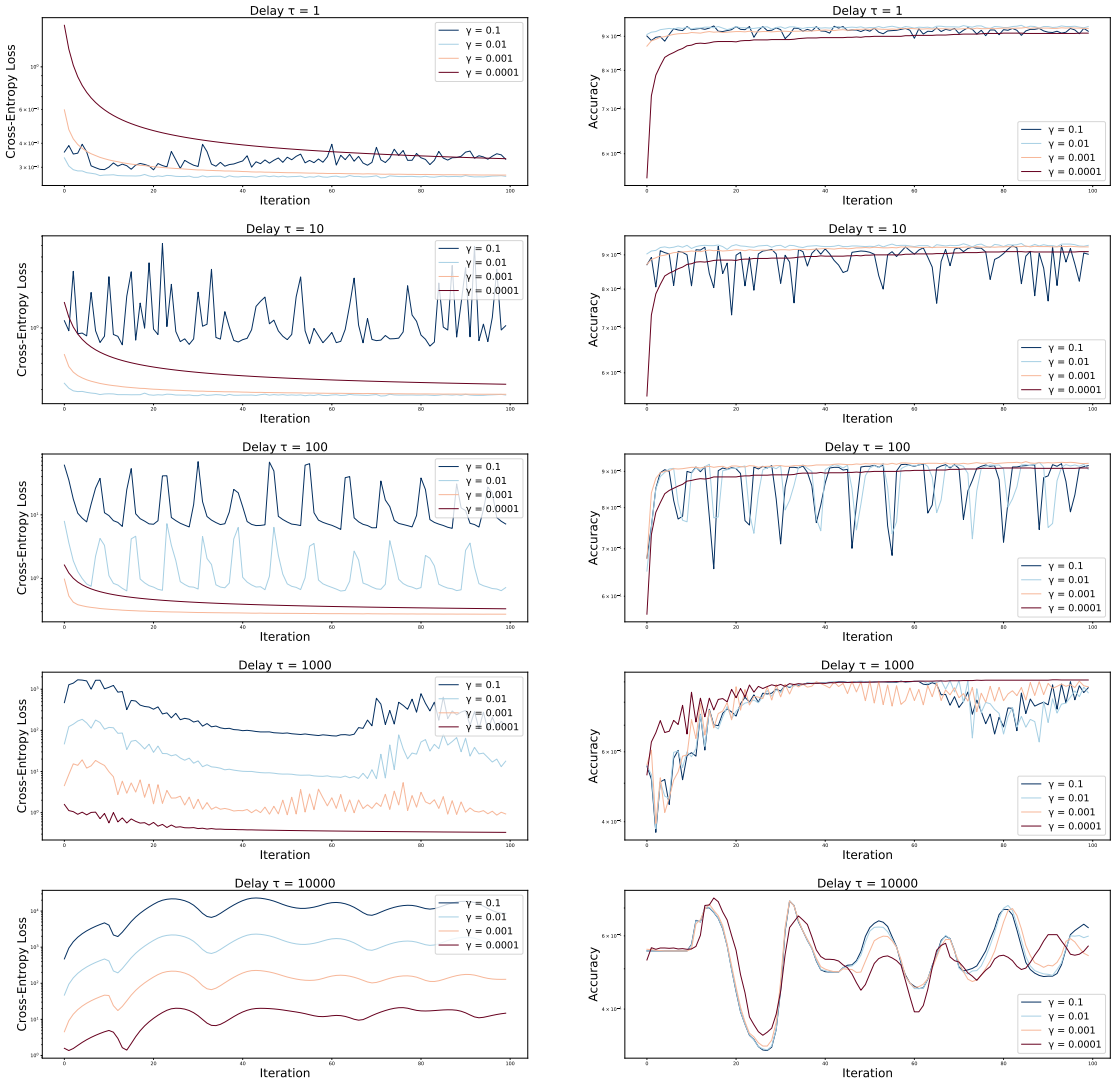


Figure 3.1: Cross-entropy loss and accuracy with different delays and learning rates

In all experiments, a delay of 1 means no delay at all, because it describes the size of the queue. On the left-hand side, we plot the cross-entropy loss and on the right-hand side, we plot the accuracy measured on a validation set with each iteration. The main observation we make coincides with the provided theory: The higher we set the delays,

the lower we need to set the learning rate. At the same time, we do not want to set the learning rate too low as it slows down learning. Consider the graph with a delay of 10 (second row). Clearly, the learning rate of 0.1 is too high because the loss function is diverging. However, a learning rate of 0.0001 is also not optimal because it cannot minimize the loss as fast as 0.001.

In the last row, we jeopardize the delay completely by setting a meaningless high delay. In fact, in most iterations, we always refer to the initial random guess. Clearly, this prevents any learning.

3.2 Asynchronous ADMM Logistic Regression

So far, we have seen the theoretic analysis of Flexible ADMM (Section 2.4) and an extension to it which could deal with delays in Section 2.4.3. Here, we want to evaluate this extension and see how delays affect the results but also how running ADMM asynchronously could speed up the whole computation in some situations. To see this, we are going to solve two problems, namely a logistic regression image classification task using the MNIST dataset again and an image segmentation problem using convex optimization methods.

Each problem will be solved using an implementation with simulated delays and with real delays. By simulated delays, we mean that we do not run multiple workers but our iteration will use delayed information, similar to the SGD experiments in Section 3.1. In contrast to that, we also perform experiments that use real delays, i.e. we start multiple workers on a real GPU cluster and let them communicate synchronously and asynchronously and we measure their performance. To let workers communicate over a network, we use the PyTorch `distributed` Package[1] to which we will also give a short tutorial on how to use the package in Section 3.4.

3.2.1 Derivation

Recall the MNIST Logistic Regression Problem 3.1.1.

$$\min_{x \in \mathbb{R}^n} f(x) = \mathbb{E}_{\xi}[F(x;\xi)]$$

We notice that f is separable and we can split it by distributing the MNIST dataset onto K workers. Then, each worker optimizes only over a fixed subset. By doing so we

[1] see official PyTorch documentation here: `https://pytorch.org/docs/stable/distributed.html`

end up with the consensus Problem 2.1.3. Now, Each new function $G_i(x)$ consists of a subset of the original MNIST data.

$$\min_{x \in R^n} f(x) \Leftrightarrow \min_{x_0, \{x_k\} \in \mathbb{R}^n} \max_{\{y_k\} \in \mathbb{R}^n} \mathcal{L}_\rho(x_0, \{x_k\}; \{y_k\})$$
$$\min_{x_0, \{x_k\} \in \mathbb{R}^n} \max_{\{y_k\} \in \mathbb{R}^n} \sum_{k=1}^{K} G_k(x_k) + \sum_{k=1}^{K} \langle y_k, x_0 - x_k \rangle + \sum_{k=1}^{K} \frac{\rho_k}{2} \|x_0 - x_k\|^2$$

In practice, it depends on the problem and the available resources on how one would split the data in a meaningful way. If the data set is small, every worker could have access to the *full* data set. If some workers have better hardware than others, they could receive a larger subset than the ones with average hardware. And, if the data set is very large, it makes sense that each worker only has access to a subset of it.

Because we only need to achieve consensus and no other optimization over x_0, it can be interpreted as an averaging over all model variables x_k. To see this, we state the optimality condition and solve for x_0^*.

$$\begin{aligned} x_0^* &= \arg\min_{x_0 \in \mathbb{R}^n} \mathcal{L}_{\{\rho_k\}}(x_0, \{x_k^t\}; \{y_k^t\}) \\ &\quad \arg\min_{x_0 \in \mathbb{R}^n} \sum_{k=1}^{K} G_k(x_k) + \sum_{k=1}^{K} \langle y_k, x_0 \rangle + \sum_{k=1}^{K} \frac{\rho_k}{2} \|x_0 - x_k\|^2 \\ 0 &= \sum_{k=1}^{K} y_k + \sum_{k=1}^{K} \rho_k (x_0^* - x_k) \\ x_0^* &= \frac{\sum_{k=1}^{K} \rho_k x_k - \sum_{k=1}^{K} y_k}{\sum_{k=1}^{K} \rho_k} \end{aligned}$$

As for the local subproblems, we approximate a solution by applying gradient descent with a learning rate γ_k for it. This implies, that we do not solve the problem exactly. To mitigate this issue, we could run multiple gradient descent epochs E to better approximate the *true* minimum. This comes with the cost of additional calculation time. In practice, it turned out that one epoch $E = 1$ of gradient descent is a good tradeoff between the approximation and calculation time in this experiment.

This results in the final Algorithm 8.

Algorithm 8: Delayed ADMM for logistic regression with Lagrangian multipliers

Initialize x_0^0, $\{x_k^0\}_{k=1}^K$ and $\{y_k^0\}_{k=1}^K$
Choose appropiate $\{\rho_k\}_{k=1}^K$
for $t = 0, 1, \ldots$ **do**
  $x_0^{t+1} = \frac{\sum_{k=1}^K \rho_k \hat{x}_k - \sum_{k=1}^K \hat{y}_k}{\sum_{k=1}^K \rho_k}$
  for $k = 1, \ldots, K$ **do**
    Distribute to worker k
    $x_k^{t+1} = x_k^t$
    for $e = 1, \ldots, E$ **do**
      $x_k^{t+1} = x_k^{t+1} - \gamma_k \left(\nabla g_k(x_k^{t+1}) + y_k^t + \rho_k (x_k^{t+1} - \hat{x}_0^{t+1}) \right)$
    end
    $y_k^{t+1} = y_k^t + \rho_k \left(x_k^{t+1} - \hat{x}_0^{t+1} \right)$
  end
end

If the data comes from the same distribution, we can neglect the dual variables y_k because the primal variables x_k will achieve consensus naturally. The authors in [LWC19] give further insights into it. Removing the dual variables simplifies our global and local update rules. The augmented Lagrangian is now:

$$\mathcal{L}_{\{\rho_k\}}(x_0, \{x_k\}) := \sum_{k=1}^K g_k(x_k) + \sum_{k=1}^K \frac{\rho_k}{2} \|x_0 - x_k\|^2$$

Solving the optimality condition and rearranging for x_0^{t+1} gives an averaging over all worker variables again. And, for the optimization over the local worker updates, we

use SGD as well, resulting in Algorithm 9.

Algorithm 9: Delayed ADMM for logistic regression without Lagrangian multipliers

Initialize x_0^0, $\{x_k^0\}_{k=1}^K$ and $\{y_k^0\}_{k=1}^K$
Choose appropiate $\{\rho_k\}_{k=1}^K$
for $t = 0, 1, \ldots$ **do**
    $x_0^{t+1} = \frac{\sum_{k=1}^K \rho_k x_k}{\sum_{k=1}^K \rho_k}$
    for $k = 1, \ldots, K$ **do**
        Distribute to worker k
        $x_k^{t+1} = x_k^t$
        for $e = 1, \ldots, E$ **do**
            $x_k^{t+1} = x_k^{t+1} - \gamma_k \left(\nabla g_k(x_k^{t+1}) + \rho_k (x_k^{t+1} - \hat{x}_0^{t+1}) \right)$
        end
    end
end

3.2.2 Simulated Delays

For the following experiments, we simulate asynchronicity by using outdated/delayed variables for the global "master" update and the local "worker" updates. There are different possibilities to simulate delays. Here, we first simulate a worst-case scenario by setting a constant delay of τ for all variables. That means, *every* update uses the variables from τ iterations ago. Obviously, this scenario might not be very realistic but it is not unreasonable to assume that it models the worst-case. To provide a more realistic modeling, we sample delays from a uniform distribution in a second experiment. We are especially interested in whether and when our cross-entropy loss function converges and how these two different delay sampling strategies compare.

We distinguish between these four categories and measure the cross-entropy loss and accuracy:

- Lagrangian multipliers with data splitting
- Lagrangian multipliers without data splitting
- No Lagrangian multipliers with data splitting
- No Lagrangian multipliers without data splitting

When we use data splitting, the MNIST dataset is partitioned into nonoverlapping, equal-sized subsets and each worker gets access to one subset only. In the case without splitting, each worker has access to the full training data. This also implies that one epoch of a worker without splitting goes through more data than one worker with splitting. Each local worker performs one epoch and we stop the experiment after 100 iterations.

For the first experiment, we use a constant delay of $\tau = 5$ and vary the $\rho_k \equiv \rho \in \{1.0, 5.0, 10.0, 50.0\}$ parameter that affects the strong convexity of every local objective function. The learning rate γ for the workers is set to 0.001 and we simulate $K = 5$ worker nodes. We also compare the results with a run without delays ($\tau = 1$) and a small $\rho = 1.0$, visualized by a dotted line in Figure 3.2.

In all the graphs we see: The higher the penalty factor ρ, the slower the learning. However, for some small ρ we do not have convergence in the cross-entropy loss at all. Also, due to the constant delay, we see "step-functions" in the graph. This happens because of the lag in the variables. In the beginning, we always refer to the same initial guess and we cannot progress until the iteration number becomes higher than the delay.

In specific, if we do have Lagrangian multipliers, then $\rho = 1.0$ and $\rho = 5.0$ are set too low. But, once we set $\rho = 10.0$, we converge towards the baseline. Interestingly, if we remove the Lagrangian multipliers from the experiment, it performs much more well-behaved and even low ρ values make the loss function converge. Additionally, the accuracy is much closer to the baseline than in the experiments with multipliers. Last but not least, data splitting did not result in a much different outcome.

Figure 3.3 shows the same experiment but with delays sampled from a uniform distribution $\tau \sim U(1, 5)$ where 1 represents no delay at all.

Now, because we sample smaller delays much more often compared to the constant distribution experiment, the graphs do not look like a step function anymore. But, we still observe the effect of ρ on the convergence and speed.

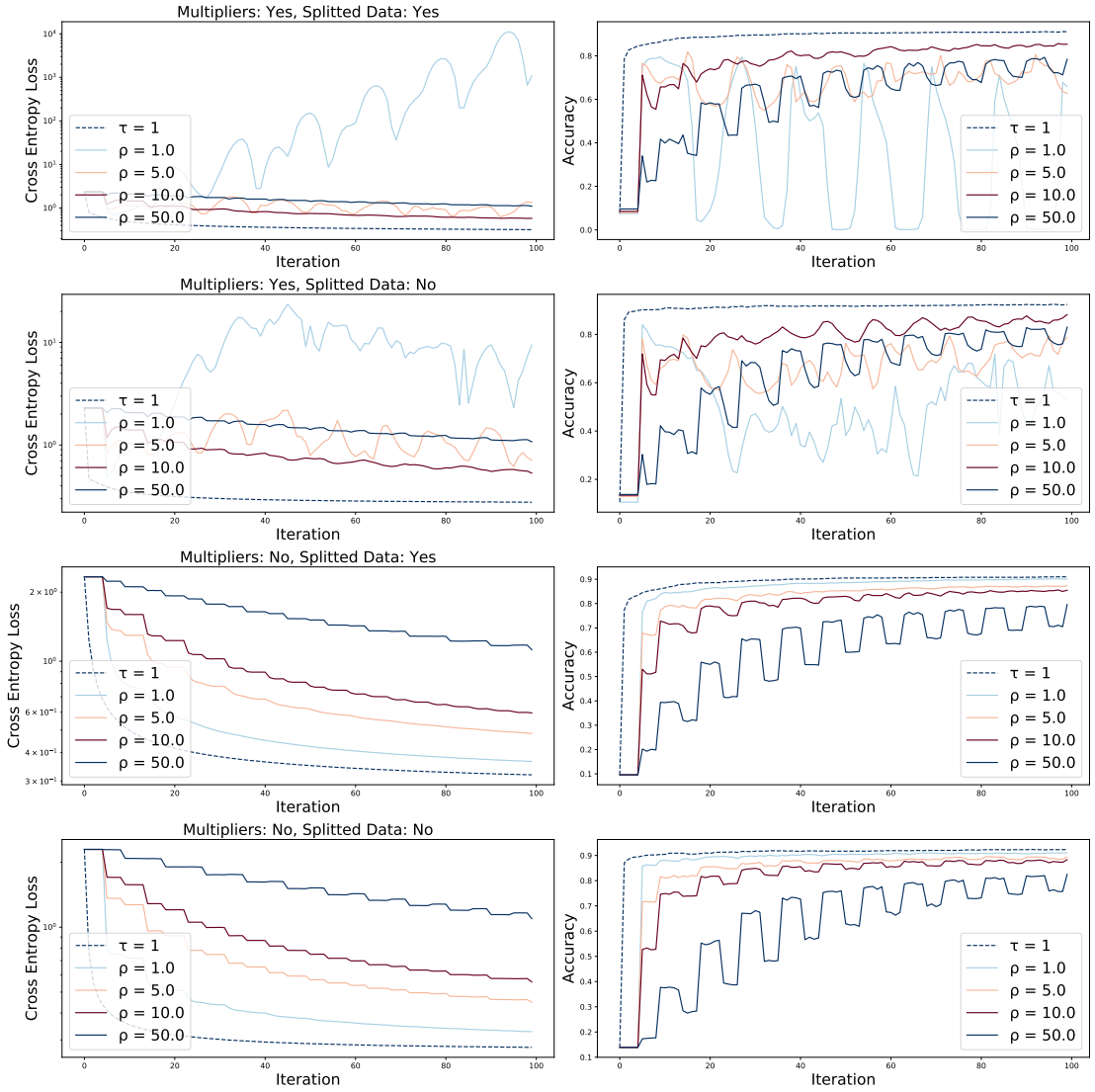


Figure 3.2: Constant delay

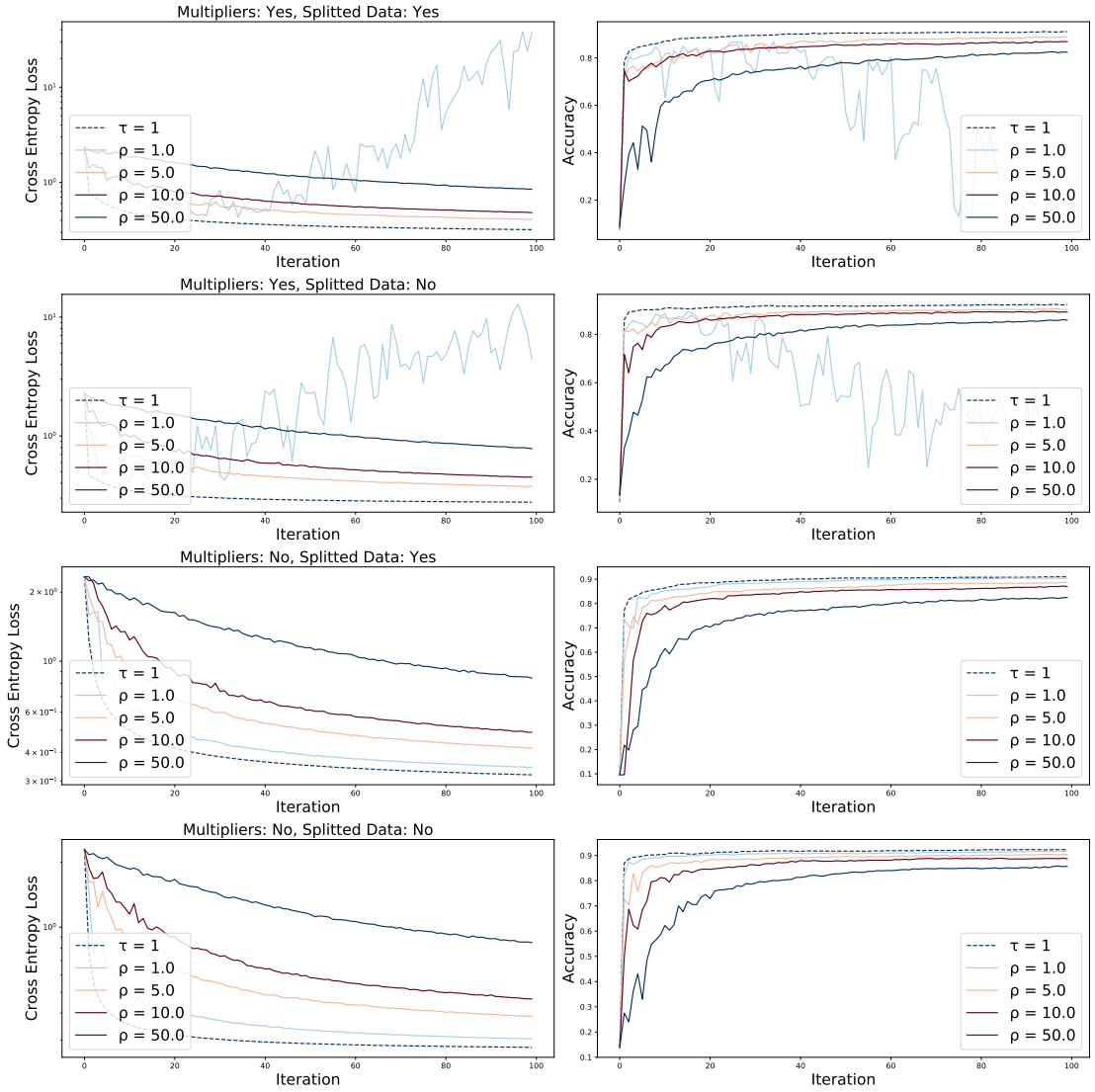


Figure 3.3: Uniform delay

Next, we vary the maximum delay, fix $\rho = 10.0$ and also use the same local learning rate $\gamma = 0.001$ and same number of simulated nodes $K = 5$.

Due to the constant delay, in the beginning, we observe a lag in all graphs from Figure 3.4 depending on the delay size. We also see step-functions again where the step size depends on the current delay τ. In addition, the graphs confirm that a higher delay slows down learning. This happens because we only approximately progress every τ iterations.

And, notice that the experiments without Lagrangian multipliers even show convergence with higher delays. When we consider Lagrangian multipliers, the delay must not exceed $\tau = 6$. However, if we remove the multipliers, even the highest delay of $\tau = 10$ results in convergence.

Finally, we change the sampling strategy again to a uniform distribution and plot the results in Figure 3.5.

We observe that all graphs converge regardless of the delay. This seems reasonable since we sample a delay of $\tau = 5.5$ on average and in the previous experiment this delay was not too high. Regarding the accuracies, they are closer to each other and range between 80% to 90%.

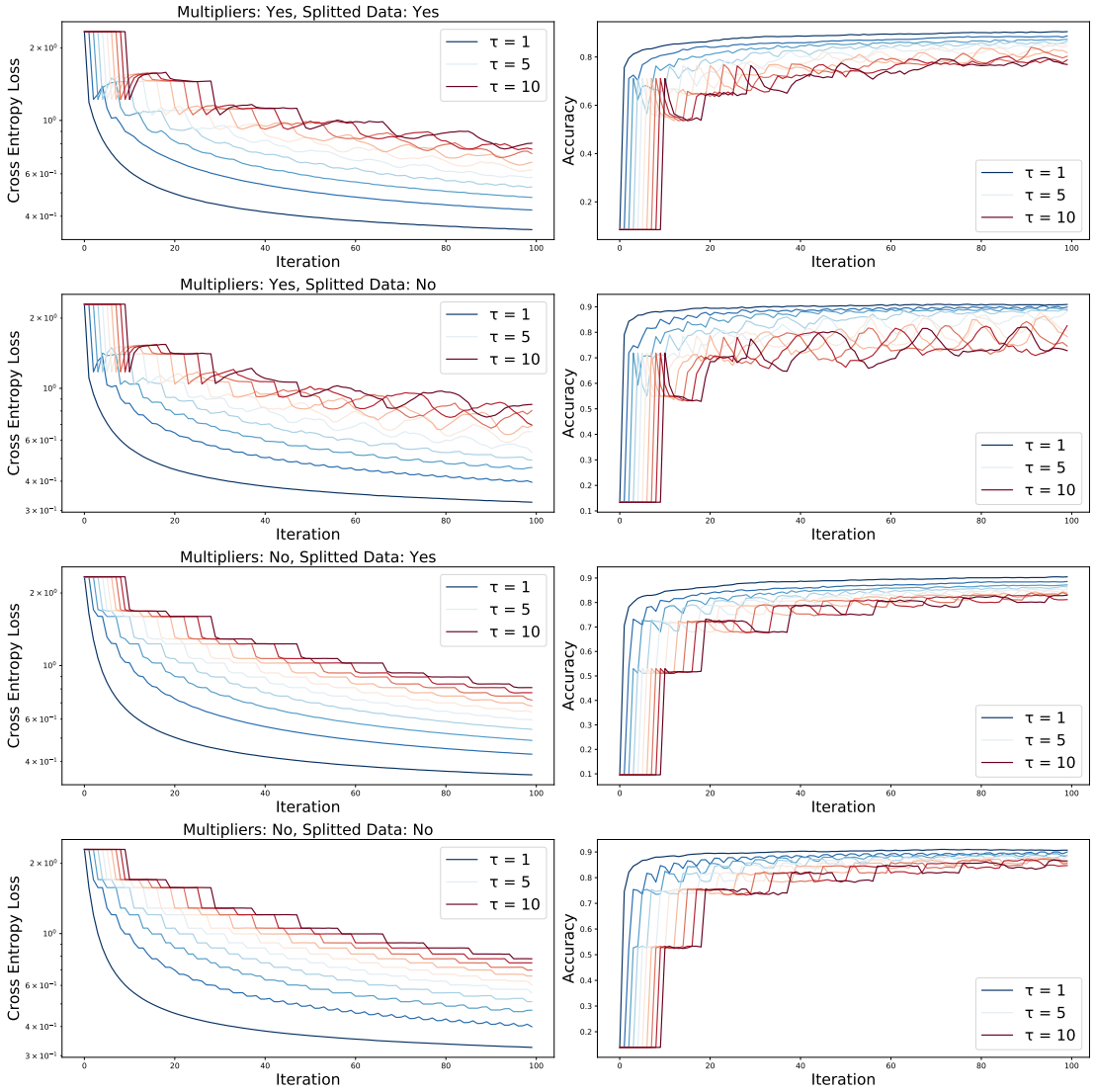


Figure 3.4: Constant delay

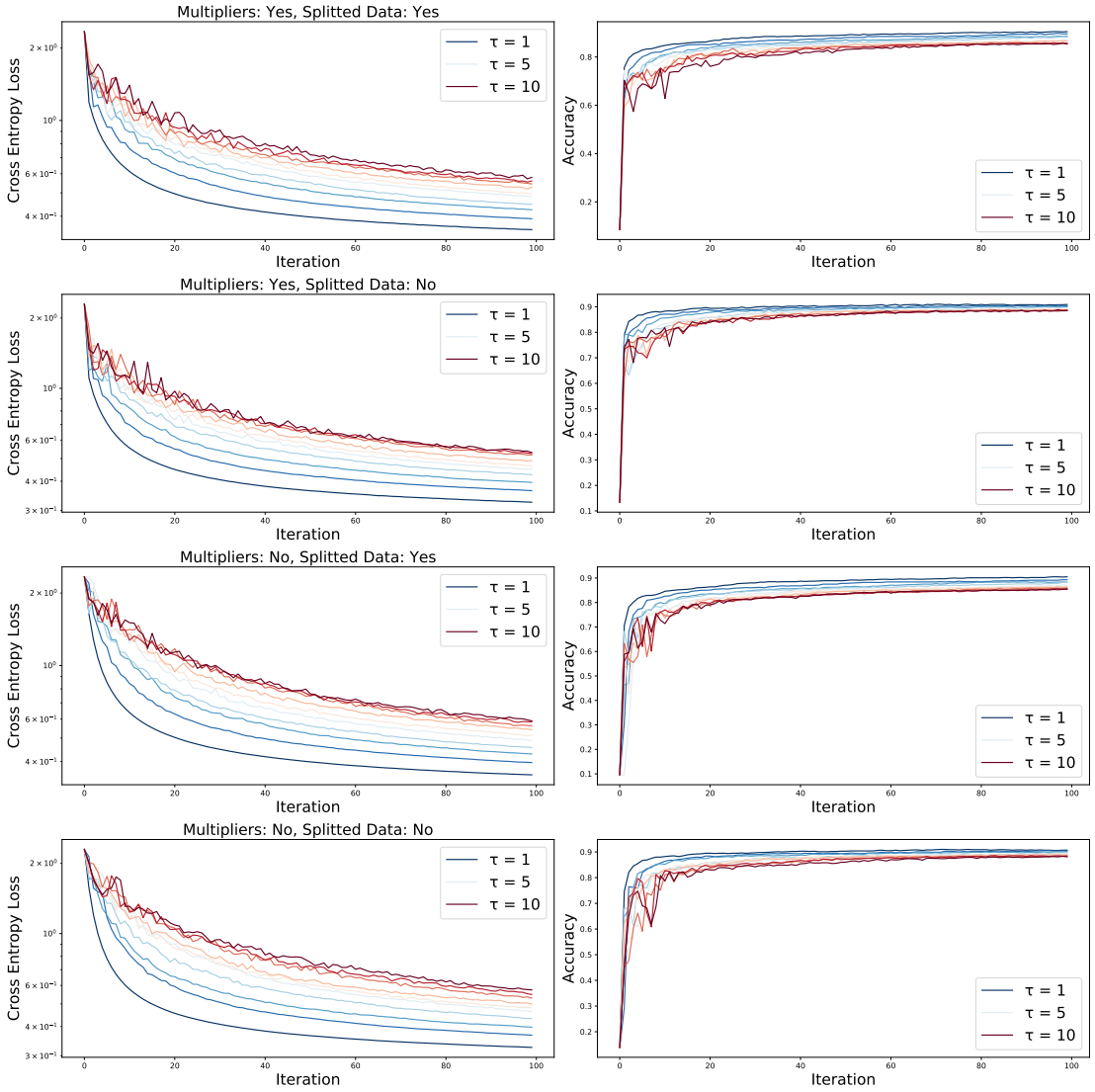


Figure 3.5: Uniform delay

3.2.3 Real Delays

For the next logistic regression experiment, we implement a real distributed ADMM version using the partial barrier strategy and let it run on a real GPU cluster. The communication between workers and master is implemented via the Pytorch `distributed` package (see Section 3.4). Here, we are interested in how two slow workers affect the performance of three fast ones in a synchronous and an asynchronous setting. Again, we measure the cross-entropy loss on a validation dataset and its accuracy for the previously mentioned four different categories.

To implement this experiment successfully, we need to change the algorithm from the general theoretical perspective to a specific, practical one. We decided to use a partial barrier due to the simplicity in its implementation. Consider this modification of Algorithm 8 for the master node:

Algorithm 10: Master algorithm for logistic regression

Initialize x_0^0, $\{x_k\}_{k=1}^K$, $\{y_k\}_{k=1}^K$ and partial barrier B
$U = \emptyset$
for $k = 1, \ldots, K$ **do**
    Start receiving x_k and y_k from worker k
end
for $t = 0, 1, \ldots$ **do**
    for $k \notin U$ **do**
        If x_k and y_k have been received, add k to U
    end
    if $|U| \geq B$ **then**
        $x_0^{t+1} = \frac{\sum_{k=1}^K \rho_k x_k - \sum_{k=1}^K y_k}{\sum_{k=1}^K \rho_k}$
        for $k \in U$ **do**
            Start sending x_0^{t+1} to worker k
            Remove k from U
        end
    end
end

And, for the worker nodes:

Algorithm 11: Worker k algorithm for logistic regression

Initialize x_k^0, y_k^0, x_0
while *True* **do**
  Distribute x_k^0 (and y_k^0)
  $x_0 \leftarrow$ Receive x_0
  $x_k^{t+1} = x_k^t - \gamma_k \left(\nabla g_k(x_k) + y_k^t + \rho_k (x_k^t - x_0) \right)$
  $y_k^{t+1} = y_k^t + \rho_k \left(x_k^{t+1} - x_0^{t+1} \right)$
end

The algorithm without Lagrangian multipliers can be derived accordingly. In addition, because averaging the models by the master is a very easy task, we do not model delays *in both directions*. That means the worker nodes stay idle until they receive a new job which resulted in idle time between 1s and 3s on our hardware.

In total, we run 5 worker nodes and 1 master node. We set $\rho = 10$ and the local learning rate $\gamma = 0.01$. We quickly noticed that the calculation time of each worker is roughly the same and that communication within a GPU cluster is very stable. This resulted in an (almost) synchronous behavior without even enforcing it. To artificially counteract this, we purposely slowed down the calculation time by adding 60 seconds to each iteration of two workers to observe, how this affects the final results. First, we set the partial barrier $B = 5$ to enforce synchronicity. Then, we lower it to $B = 1$ to allow asynchronicity.

In all of our four experiments in Figure 3.6 we plot the results with respect to the physical time. It turns out that the asynchronous mode minimizes the cross-entropy faster than the synchronous mode.

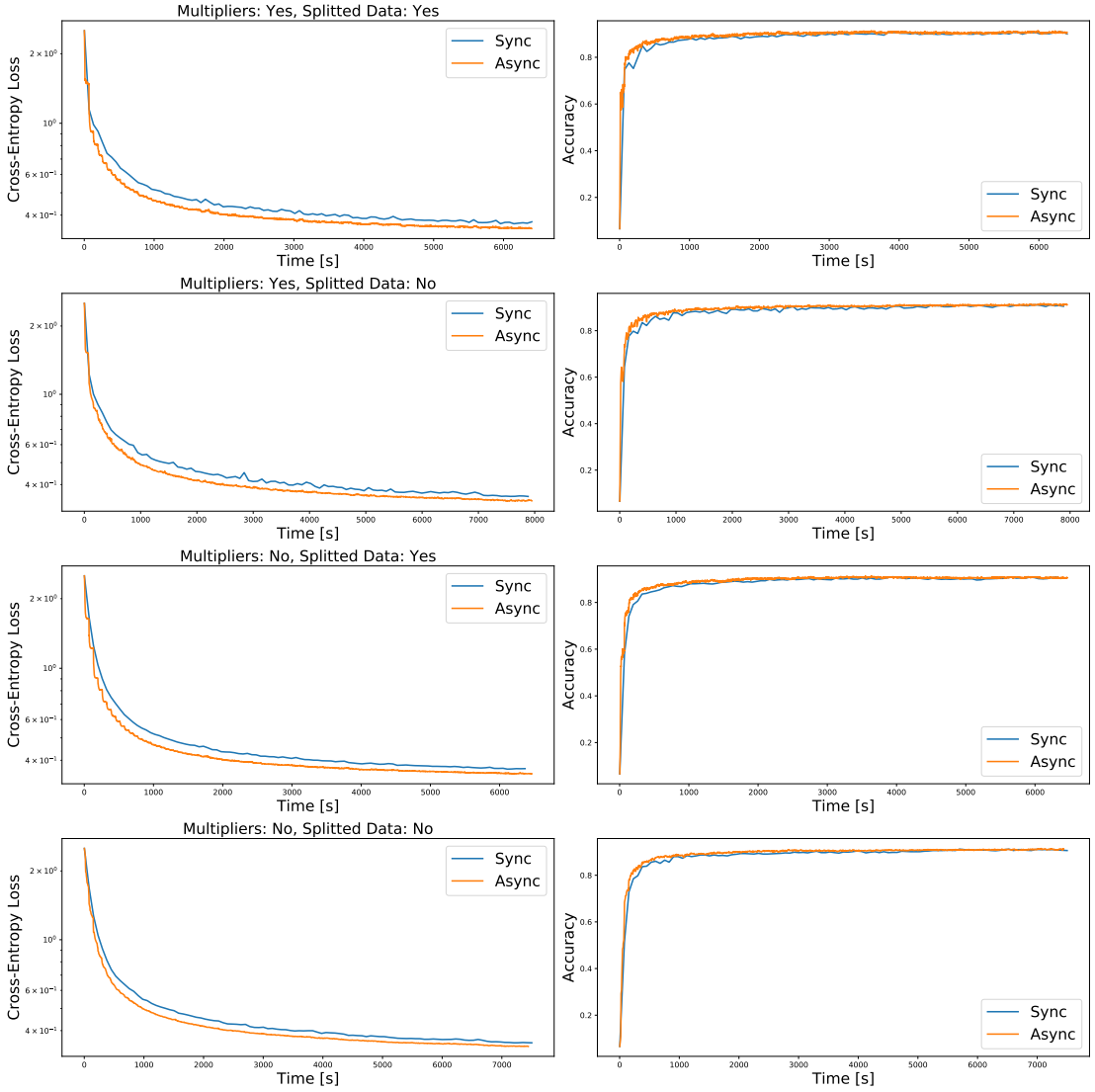


Figure 3.6: Cross-entropy and accuracies for synchronous and asynchronous implementation depending on the category

3.3 Asynchronous ADMM Image Segmentation

For the next task, we are given an image and depending on the colors we cluster the image in L segments. Figure 3.7 shows an example input and its segmentation where $L = 4$.

(a) original input image

(b) segmented output image

Figure 3.7: Image segmentation example using 4 segments

The term image segmentation is ambiguous since there exist other segmentation tasks as well that do not create segments based on the color of the image but based on the context of the objects within it. Usually, these kinds of methods need to make usage of heavy deep learning methods and require a lot of data. We refer to [BKC17] as one famous example. Here, we focus on the segmentation task described at the beginning which works without training and uses convex optimization methods only.

3.3.1 Derivation of the Master Update

We start by stating the image segmentation Problem 3.3.1.

Problem 3.3.1 (Image Segmentation)**.** *Consider the following optimization problem*

$$\min_{u \in \mathbb{R}^{n \times L}} \sum_{j=1}^{n} \Big(\delta\{u_j \in \Delta^{L-1}\} + \langle u_j, f_j \rangle \Big) + \alpha \sum_{l=1}^{L} \|\nabla u^l\|_{1,\delta}$$

where u is our optimization variable, n the number of pixel in the image, $\delta\{\cdot\}$ is the indicator function, Δ^{L-1} is the probability simplex and $\|\cdot\|_{1,\delta} \colon \mathbb{R}^n \to \mathbb{R}$ is the Huber function parameterized by δ and defined as

$$\|u\|_{1,\delta} := \sum_{i=1}^{n} |u_i|_{1,\delta} = \sum_{i=1}^{n} \begin{cases} \frac{u_i^2}{2\delta} & \text{if } |u_i| \leq \delta \\ |u_i| - \frac{\delta}{2} & \text{otherwise} \end{cases}$$

Our goal is to split the objective function such that multiple workers can optimize over a part of the image independently. One option to achieve this is by splitting the Huber term. Instead of applying the discrete gradient operator ∇ on the whole image, each worker could only apply it on a smaller section, for example like Figure 3.8

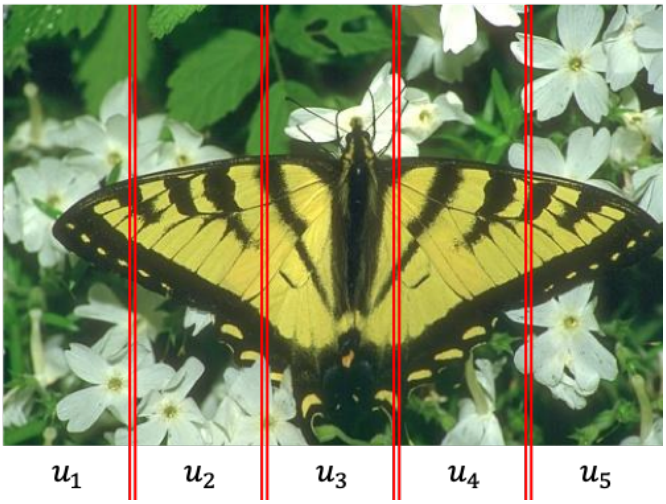


Figure 3.8: One possible approach to split an image

Notice that there are two splitting lines visualized in Figure 3.8. Because we calculate a discrete gradient, every pixel needs access to the pixels from their neighborhood above and to the left. Therefore, the splits must overlap or we would end up with visible lines between sections.

On the one hand, we introduce k new optimization variables u_k. On the other hand, this splitting shrinks the dimensionality from $\mathbb{R}^{n\times L}$ to $\mathbb{R}^{n_k\times L}$ where n_k is the number of pixel in section k. Due to this shrinkage, we also need to introduce selection matrices A_ks that select the right section of the whole image. For the master node "aggregation" update, the projection onto the probability simplex Δ^{L-1} gets applied to the whole image. Therefore, the variable $u_0 \in \mathbb{R}^{n\times L}$ keeps its dimensions. Let's formalize these ideas.

Problem 3.3.2 (Splitted Image Segmentation)**.**

$$\min_{u_0,\{u_k\}} \sum_{i=1}^{n} \Big(\delta\{(u_0)_i \in \Delta^{L-1}\} + \langle (u_0)_i, f_i \rangle \Big) + \sum_{k=1}^{K} \alpha \sum_{l=1}^{L} \|\nabla {u_k}^l\|_{1,\delta}$$

such that for all k, $A_k u_0 = u_k$.

As before, we introduce dual variables and state the augmented Lagrangian.

$$\mathcal{L}(u_0, \{u_k\}; \{p_k\}) := \sum_{i=1}^{n} \left(\delta\{(u_0)_i \in \Delta^{L-1}\} + \langle (u_0)_i, f_i \rangle \right) + \sum_{k=1}^{K} \alpha \sum_{l=1}^{L} \|\nabla {u_k}^l\|_{1,\delta} + \sum_{k=1}^{K} \langle p_k, A_k u_0 - u_k \rangle_F + \sum_{k=1}^{K} \frac{\rho_k}{2} \|A_k u_0 - u_k\|_F^2$$

Now, we derive the master u_0 update and the worker primal u_k and dual p_k updates, as defined by ADMM. For clarity, we do not add delays to the variables in our derivation yet. However, in the experiments, we do simulate delays again and also present a real implementation using the partial barrier method.

$$\begin{aligned}
u_0^{t+1} &\in \operatorname*{arg\,min}_{u_0 \in \mathbb{R}^{n \times L}} \mathcal{L}(u_0, \{u_k^t\}; \{p_k^t\}) \\
&\in \operatorname*{arg\,min}_{u_0 \in \mathbb{R}^{n \times L}} \sum_{i=1}^{n} \left(\delta\{(u_0)_i \in \Delta^{L-1}\} + \langle (u_0)_i, f_i \rangle \right) + \sum_{k=1}^{K} \langle p_k^t, A_k u_0 \rangle_F \\
&\quad + \sum_{k=1}^{K} \frac{\rho_k}{2} \|A_k u_0 - u_k^t\|_F^2 \\
&\in \operatorname*{arg\,min}_{u_0 \in \mathbb{R}^{n \times L}} \sum_{i=1}^{n} \delta\{(u_0)_i \in \Delta^{L-1}\} + \langle u_0, f \rangle + \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} p_k^t, u_0 \right\rangle_F \\
&\quad + \frac{\rho_k}{2} \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} A_k u_0, u_0 \right\rangle_F - \rho_k \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} u_k^t, u_0 \right\rangle_F \\
&\in \operatorname*{arg\,min}_{u_0 \in \mathbb{R}^{n \times L}} \sum_{i=1}^{n} \delta\{(u_0)_i \in \Delta^{L-1}\} - \rho_k \left\langle u_0, -\frac{1}{\rho_k} f \right\rangle - \rho_k \left\langle -\frac{1}{\rho_k} \sum_{k=1}^{K} A_k^\mathsf{T} p_k^t, u_0 \right\rangle_F \\
&\quad + \frac{\rho_k}{2} \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} A_k u_0, u_0 \right\rangle_F - \rho_k \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} u_k^t, u_0 \right\rangle_F \\
&\in \operatorname*{arg\,min}_{u_0 \in \mathbb{R}^{n \times L}} \sum_{i=1}^{n} \delta\{(u_0)_i \in \Delta^{L-1}\} + \frac{\rho_k}{2} \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} A_k u_0, u_0 \right\rangle_F \\
&\quad - \rho_k \left\langle \sum_{k=1}^{K} A_k^\mathsf{T} u_k^t - \frac{1}{\rho_k} \left(f + \sum_{k=1}^{K} A_k^\mathsf{T} p_k^t \right), u_0 \right\rangle_F
\end{aligned}$$

We see that we can minimize the whole function by minimizing each component of u_0

separately.

$$
\begin{aligned}
(u_0^{t+1})_i \in & \underset{(u_0)_i \in \mathbb{R}^L}{\arg\min}\, \delta\{(u_0)_i \in \Delta^{L-1}\} + \frac{\rho_k}{2} \left\langle \left(\sum_{k=1}^{K} A_k^\mathsf{T} A_k \right)_i (u_0)_i, (u_0)_i \right\rangle \\
& - \rho_k \left\langle \left(\sum_{k=1}^{K} A_k^\mathsf{T} u_k^t - \frac{1}{\rho_k} \left(f + \sum_{k=1}^{K} A_k^\mathsf{T} p_k^t \right) \right)_i, (u_0)_i \right\rangle \\
\in & \underset{(u_0)_i \in \mathbb{R}^L}{\arg\min}\, \delta\{(u_0)_i \in \Delta^{L-1}\} + \frac{\rho_k \left(\sum_{k=1}^{K} A_k^\mathsf{T} A_k \right)_i}{2} \langle (u_0)_i, (u_0)_i \rangle \\
& - \rho_k \left(\sum_{k=1}^{K} A_k^\mathsf{T} A_k \right)_i \Bigg\langle \underbrace{\frac{1}{\left(\sum_{k=1}^{K} A_k^\mathsf{T} A_k \right)_i} \left(\sum_{k=1}^{K} A_k^\mathsf{T} u_k^t - \frac{1}{\rho_k} \left(f + \sum_{k=1}^{K} A_k^\mathsf{T} p_k^t \right) \right)_i}_{(v)_i}, (u_0)_i \Bigg\rangle \\
\in & \underset{(u_0)_i \in \mathbb{R}^L}{\arg\min}\, \delta\{(u_0)_i \in \Delta^{L-1}\} + \frac{\rho_k \left(\sum_{k=1}^{K} A_k^\mathsf{T} A_k \right)_i}{2} \| (u_0)_i - (v)_i \|^2
\end{aligned}
$$

The last problem can be efficiently solved by the "(Scaled) Projection onto the Probability Simplex" algorithm.

3.3.2 Scaled Projection onto Probability Simplex

One subproblem that arises during the master update is the following:

Problem 3.3.3 (Scaled Projection onto Probability Simplex)**.** *Consider the minimization problem*

$$
\begin{aligned}
w^* &= \underset{w \in \mathbb{R}^n}{\arg\min}\, \delta_{\Delta^{L-1}}(w) + \frac{\tau}{2} \|w - v\|_{H'}^2 \\
&= \underset{w \in \mathbb{R}^n}{\arg\min}\, \delta_{\Delta^{L-1}}(w) + \frac{1}{2} (w - v)^\mathsf{T} H (w - v)
\end{aligned}
$$

where Δ^{L-1} is the probability simplex and $H := \tau H'$ is a diagonal matrix with strictly positive entries only and $\tau > 0$.

The scaling matrix H allows us to define different weights in case one would like to favor a certain segment in the image over another. In our original segmentation task 3.3.1 we can simply set H to be the identity matrix but it might be interesting to tune H to get a more favorable outcome. Therefore, in this section, we provide a modified algorithm based on [Duc+08] that can also deal with scaling. The authors in [WC13]

give a simplified proof based on the KKT conditions of Problem 3.3.3. We take their proof and modify it to include a diagonal scaling matrix H.

(a) reference image (b) scaling example 1 (c) scaling example 2

Figure 3.9: Scaling examples with high weights on a single color

Figure 3.9 visualizes the effect of two different weight matrices and compares it with an image that has not been scaled.

We claim that Algorithm 12 solves Problem 3.3.3.

Algorithm 12: Scaled Projection onto Probability Simplex

1 Rearrange entries in H and v based on sorting of Hv in descending order

2 $\rho = \max \left\{ 1 \leq j \leq L : H_{(jj)} v_{(j)} + \left(\sum_{i=1}^{j} \frac{1}{H_{(ii)}} \right)^{-1} \left(1 - \sum_{i=1}^{j} v_{(i)} \right) > 0 \right\}$

3 $\lambda^* = \left(\sum_{i=1}^{\rho} \frac{1}{H_{(ii)}} \right)^{-1} \left(1 - \sum_{i=1}^{\rho} v_{(i)} \right)$

4 **return** $w_i^* = \max \left\{ v_i + \frac{\lambda^*}{H_{ii}}, 0 \right\}$

Observe its fast run time of $\mathcal{O}(n \log n)$ due to the sorting operation at the very beginning and that it can be efficiently implemented for both CPUs and GPUs.[2]

In order to prove our claim, we start by stating the Lagrangian of Problem 3.3.3 with its Lagrange multipliers $\lambda \in \mathbb{R}$ and $\beta \in \mathbb{R}^n_{\geq 0}$. Obviously, the probability simplex enforces the values of w to be non-negative and to sum up to one.

$$\begin{aligned} \mathcal{L}(w, \lambda, \beta) &:= \frac{1}{2}(w - v)^\mathsf{T} H (w - v) - \lambda \left(\sum_{i=1}^{n} w_i - 1 \right) - \sum_{i=1}^{n} \beta_i w_i \\ &= \frac{1}{2}(w^\mathsf{T} H w - v^\mathsf{T} H w - v^\mathsf{T} H^\mathsf{T} w + v^\mathsf{T} v) - \lambda \left(\sum_{i=1}^{n} w_i - 1 \right) - \sum_{i=1}^{n} \beta_i w_i \end{aligned}$$

[2] An efficient implementation using PyTorch can be found here: `https://gist.github.com/filipre/f90c2b49d55ebd89329218a0f64dcf5a`

We take the gradient with respect to w. Remember that H is diagonal and its values are strictly positive.

$$\begin{aligned}\nabla_w \mathcal{L}(w, \lambda, \beta) &= \frac{1}{2}(H^\mathsf{T} + H)w - \frac{1}{2}H^\mathsf{T}v - \frac{1}{2}Hv - \lambda - \beta \\ &= H(w - v) - \lambda - \beta \\ \Rightarrow \frac{\partial \mathcal{L}(w, \lambda, \beta)}{\partial w_i} &= H(w_i - v_i) - \lambda - \beta_i\end{aligned}$$

If w is optimal, these KKT-conditions hold:

$$\begin{aligned} H_{ii}w_i^* - H_{ii}v_i - \lambda^* - \beta_i^* &= 0 && \forall i \\ H_{ii}w_i^* &\geq 0 && \forall i \\ \beta_i^* &\geq 0 && \forall i \\ w_i^*\beta_i^* &= 0 && \forall i \\ \sum_{i=1}^{n} w_i^* &= 1 \end{aligned}$$

On the one hand, if $w_i^* > 0$ then $\beta_i^* = 0$ due to the "complementary slackness" condition and thus, $H_{ii}w_i^* = H_{ii}v_i + \lambda^* > 0$. On the other hand, if $w_i^* = 0$ then $0 = H_{ii}v_i + \lambda^* + \beta_i^* = 0$ and $H_{ii}v_i + \lambda^* = -\beta_i^* \leq 0$.

Now, if we sort the entries in the vector Hv in descending order, we see that the sequence $H_{(ii)}v_{(i)} + \lambda^*$ is also descending and that there exists an index $(\rho + 1)$ where the sequence starts to be non-positive. We also confirm that at least one value in $H_{(ii)}v_{(i)} + \lambda^*$ is strictly positive. Otherwise, we could not fulfill the condition that all elements in w^* sum up to one.

$$\begin{gathered} H_{(11)}v_{(1)} + \lambda^* \cdots \geq H_{(\rho\rho)}v_{(\rho)} + \lambda^* > 0 \geq H_{(\rho+1\rho+1)}v_{(\rho+1)} + \lambda^* \geq \cdots \geq H_{(LL)}v_{(L)} + \lambda^* \\ H_{(11)}w_{(1)}^* \cdots \geq H_{(\rho\rho)}w_{(\rho)}^* > 0 = H_{(\rho+1\rho+1)}w_{(\rho+1)}^* = \cdots \geq H_{(LL)}w_{(L)}^* \end{gathered}$$

Using the condition $\sum_{i=1}^{n} w_i^* = 1$ gives us a handy formula for λ^*.

$$\begin{aligned} 1 = \sum_{i=1}^{n} w_i^* = \sum_{i=1}^{\rho} w_{(i)}^* &= \sum_{i=1}^{\rho} \left(v_{(i)} + \frac{\lambda^*}{H_{(ii)}} \right) = \sum_{i=1}^{\rho} v_{(i)} + \lambda^* \sum_{i=1}^{\rho} \frac{1}{H_{(ii)}} \\ \lambda^* &= \left(\sum_{i=1}^{\rho} \frac{1}{H_{(ii)}} \right)^{-1} \left(1 - \sum_{i=1}^{\rho} v_{(i)} \right) \end{aligned}$$

What's left is a way to find the right $\rho \in \{1, \dots, L\}$. Then we can calculate λ^* and finally w^* via

$$w_i^* = \max\left\{v_i - \frac{\lambda^*}{H_{ii}}, 0\right\}$$

One simple approach would be to try out every possible ρ and check if it fulfills the KKT-conditions. This works fine in our segmentation task because the run time of the algorithm is dominated by the sorting step and L is small. But, we can also perform a smarter test for ρ which will be faster for large L. We utilize the following lemma.

Lemma 3.3.1. *The optimal ρ can be obtained by increasing ρ until the test below becomes non-positive.*

$$\rho = \max\left\{1 \leq j \leq L : H_{(jj)}v_{(j)} + \left(\sum_{i=1}^{j} \frac{1}{H_{(ii)}}\right)^{-1}\left(1 - \sum_{i=1}^{j} v_{(i)}\right) > 0\right\}$$

Proof. We perform a case splitting and show that the test stays positive for $j \leq \rho$ and becomes non-positive for $j > \rho$.

Case I: $j = \rho$

$$H_{(\rho\rho)}v_{(\rho)} + \left(\sum_{i=1}^{\rho} \frac{1}{H_{(ii)}}\right)^{-1}\left(1 - \sum_{i=1}^{\rho} v_{(i)}\right) = H_{(\rho\rho)}v_{(\rho)} + \lambda^* > 0 \text{ by definition of } \rho$$

Case II: $j < \rho$

$$
\begin{aligned}
& H_{(jj)}v_{(j)} + \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(1-\sum_{i=1}^{j}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + 1 - \sum_{i=1}^{\rho}v_{(i)} + \sum_{i=j+1}^{\rho}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + \sum_{i=1}^{\rho}\frac{1}{H_{(ii)}}\lambda^* + \sum_{i=j+1}^{\rho}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + \sum_{i=1}^{j}\frac{1}{H_{(ii)}}\lambda^* + \sum_{i=j+1}^{\rho}\frac{1}{H_{(ii)}}\lambda^* + \sum_{i=j+1}^{\rho}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\underbrace{(H_{(jj)}v_{(j)}+\lambda^*)}_{>0} + \sum_{i=j+1}^{\rho}\underbrace{v_{(i)}+\frac{\lambda^*}{H_{(ii)}}}_{>0}\right) > 0
\end{aligned}
$$

Case III: $j > \rho$

$$
\begin{aligned}
& H_{(jj)}v_{(j)} + \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(1-\sum_{i=1}^{j}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + 1 - \sum_{i=1}^{j}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{\rho}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + \sum_{i=\rho+1}^{j}\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} + 1 - \sum_{i=1}^{\rho}v_{(i)} - \sum_{i=\rho+1}^{j}v_{(i)}\right) \\
&= \left(\sum_{i=1}^{j}\frac{1}{H_{(ii)}}\right)^{-1}\left(\sum_{i=1}^{\rho}\frac{1}{H_{(ii)}}\underbrace{H_{(jj)}v_{(j)}+\lambda^*}_{\leq 0} + \sum_{i=\rho+1}^{j}\underbrace{\left(\frac{1}{H_{(ii)}}H_{(jj)}v_{(j)} - v_{(i)}\right)}_{\leq 0}\right) \leq 0
\end{aligned}
$$

For the last inequality, notice that for all $i \in [\rho+1, \ldots, j]$ the entries $H_{(ii)}v_{(i)}$ are sorted in a descending order. Rearranging $H_{(jj)}v_{(j)} \leq H_{(ii)}v_{(i)}$ concludes the proof. □

3.3.3 Derivation of the Worker Updates

Next, we derive the u_k update for a worker k.

$$\begin{aligned}
u_k^{t+1} &\in \operatorname*{arg\,min}_{u_k \in \mathbb{R}^{n\times L}} \mathcal{L}({u_0}^{t+1}, \{u_1^{t+1}, \dots, u_{k-1}^{t+1}, u_k, u_{k+1}^t, \dots, u_K^t\}; \{p_k^t\}) \\
&\in \operatorname*{arg\,min}_{u_k \in \mathbb{R}^{n\times L}} \mathcal{L}({u_0}^{t+1}, u_k; p_k^t) \\
&\in \operatorname*{arg\,min}_{u_k \in \mathbb{R}^{n\times L}} \alpha \sum_{l=1}^{L} \|\nabla {u_k}^l\|_{1,\delta} - \langle p_k^t, u_k\rangle_F + \frac{\rho_k}{2}\|A_k u_0^{t+1} - u_k\|_F^2 \\
&\in \operatorname*{arg\,min}_{u_k \in \mathbb{R}^{n\times L}} \alpha \sum_{l=1}^{L} \|\nabla {u_k}^l\|_{1,\delta} - \rho_k \left\langle \frac{1}{\rho_k} p_k^t, u_k \right\rangle_F - \rho_k \langle A_k u_0^{t+1}, u_k\rangle_F + \frac{\rho_k}{2}\langle u_k, u_k\rangle_F \\
&\in \operatorname*{arg\,min}_{u_k \in \mathbb{R}^{n\times L}} \alpha \sum_{l=1}^{L} \|\nabla {u_k}^l\|_{1,\delta} + \frac{\rho_k}{2}\left\| u_k - \left(A_k u_0^{t+1} + \frac{1}{\rho_k} p_k^t\right)\right\|_F^2 \\
\Leftrightarrow (u_k^{t+1})^l &\in \operatorname*{arg\,min}_{(u_k)^l \in \mathbb{R}^n} \alpha \|\nabla (u_k)^l\|_{1,\delta} + \frac{\rho_k}{2}\left\| (u_k)^l - \left(A_k u_0^{t+1} + \frac{1}{\rho_k} p_k^t\right)^l\right\|^2
\end{aligned}$$

The last minimization problem is the "Huber-ROF" problem which is a modification of the original ROF problem [ROF92].

3.3.4 Solving the Huber-ROF Problem

Huber-ROF is defined as follows:

Problem 3.3.4 (Huber-ROF). *Let $\nabla \in \mathbb{R}^{m\times n}$, $v \in \mathbb{R}^n$ and $\alpha, \rho > 0$. Then*

$$\begin{aligned}
u^* &= \operatorname*{arg\,min}_{u\in\mathbb{R}^n} \alpha\|\nabla u\|_{1,\delta} + \frac{\rho}{2}\|u - v\|^2 \\
&= \operatorname*{arg\,min}_{u\in\mathbb{R}^n} \alpha \sum_{i=1}^{m} |\nabla_i u|_{1,\delta} + \frac{\rho}{2}\|u - v\|^2
\end{aligned}$$

is the Huber-ROF problem where $|\cdot|_{1,\delta}\colon \mathbb{R} \to \mathbb{R}$ is the Huber function given by

$$|u|_{1,\delta} = \begin{cases} \frac{1}{2}e^2 & |u| \le \delta \\ \delta|u| - \frac{1}{2}\delta^2 & |u| > \delta \end{cases}$$

There are different ways on how to solve this problem. For our segmentation task, ∇ can become very large but also very sparse. Depending on the implementation and available software we need to choose the right method. In this section, we show two different approaches.

Half-Quadratic Minimization

The "Half-Quadratic Minimization" or "Gradient Linearization Iteration" method [NC07] provides a fast method where the speed depends on how fast one can solve a linear system defined by ∇. For example, if a sparse linear solver is available, then even with a large but sparse matrix, Half-Quadratic Minimization can be attractive to use. During our application to the segmentation task, it turned out very successful when letting the calculation run on a CPU due to the existence of fast sparse solvers provided by the SciPy project [Vir+20]. However, for GPUs, there are no sparse linear solvers available in PyTorch and we had to utilize the method described in the next part.

Now, we solve Problem 3.3.4 and point out the main idea of the optimization procedure. To obtain the solution, we construct the matrix $L(u)$ and z and run following "relaxed fixed-point iteration":

$$L(u^t)u^{t+1} = z$$

We only show how L and z are constructed but refer to [NC07] why this construction works. There, the authors also solve the problem more generally, explain when Half-Quadratic Minimization is applicable and show the equivalence between Half-Quadratic Minimization and Gradient Linearization Iteration. Starting off, we take the derivative of the Huber function denoted as h'.

$$\frac{\mathrm{d}}{\mathrm{d}x}|x|_{1,\delta} = h'_\delta(x) = \begin{cases} \frac{1}{\delta} & \text{if } x \leq \delta \\ \frac{\operatorname{sign} x}{x} & \text{otherwise} \end{cases}$$

We use h' in matrix L. z is a constant based on the original problem.

$$\begin{aligned} L(u) &= \rho + \alpha\nabla^\mathsf{T} \operatorname{diag}([h'_\delta(u_i)]_{i=1}^n)\nabla \\ z &= \rho v \end{aligned}$$

Each iteration, we solve the linear system and update L with the new obtained variable u.

Solving Huber-ROF via a First-Order Primal-Dual Algorithm

Another approach to solving the Huber-ROF problem is by a primal-dual algorithm. In specific, we used the algorithm given in [CP11] that showed good results on many

other computer vision tasks as well. In its most general form, their algorithm is very related to PDHG [ZC08] and given by

Algorithm 13: First-Order Primal-Dual Algorithm

Choose initial $\gamma_{\text{primal}}, \gamma_{\text{dual}} > 0, \theta \in [0,1], u^0, p^0$ and set $\bar{u}^0 = u^0$
for $t = 0, 1, \ldots, T-1$ **do**
  $p^{t+1} = (I + \gamma_{\text{dual}} \partial F^*)^{-1}(p^t + \gamma_{\text{dual}} \nabla \bar{u}^t)$
  $x^{t+1} = (I + \gamma_{\text{primal}} \partial G)^{-1}(u^t - \gamma_{\text{primal}} \nabla^\mathsf{T} p^{t+1})$
  $\bar{u}^{t+1} = u^{t+1} + \theta(u^{t+1} - u^t)$
end

for the primal-dual problem:

$$\min_u \max_p \langle \nabla u, p \rangle + G(u) - F^*(p)$$

Here, we set $F(\nabla u) = \alpha \|\nabla u\|_{1,\delta}$ and $G(u) = \frac{\rho}{2}\|u - v\|^2$ and then solve the resolvent operators.

The convex conjugate of the Huber function is

$$F^*(p) = \delta_P(p) - \frac{\delta}{2}\|p\|^2$$

where $P = \{p \colon \|p\|_\infty \leq 1\}$. But, we also scale the Huber function by α in the objective. Thus, by the scaling properties of the convex conjugate, we get

$$F_\alpha^*(p) = \alpha F^*\left(\frac{p}{\alpha}\right) = \alpha \delta_P\left(\frac{p}{\alpha}\right) - \frac{\alpha\delta}{2}\left\|\frac{p}{\alpha}\right\|^2 = \alpha \delta_{P_\alpha}(p) - \frac{\alpha\delta}{2\alpha^2}\|p\|^2 = \delta_{P_\alpha}(p) - \frac{\delta}{2\alpha}\|p\|^2$$

with the new $P_\alpha = \{p \colon \|p\|_\infty \leq \alpha\}$. What's left to do is to evaluate the resolvent operator $(I + \sigma \partial F^*)^{-1}(p + \gamma_{\text{dual}} \nabla \bar{u})$. For notation, we set $\tilde{p} = p + \gamma_{\text{dual}} \nabla \bar{u}$. We get

$$p_i = \frac{\tilde{p}_i}{1 + \frac{\gamma_{\text{dual}}\delta}{\alpha}} \Bigg/ \max\left(\alpha, \frac{\tilde{p}_i}{1 + \frac{\gamma_{\text{dual}}\delta}{\alpha}}\right)$$

The resolvent of $\frac{\rho}{2}\|u - v\|^2$ can be easily computed. Again, we set $\tilde{u} = u - \gamma_{\text{primal}} \nabla^\mathsf{T} p$, derive the term and solve for u_i.

$$u_i = \frac{\tilde{u}_i + \gamma_{\text{primal}} \rho v_i}{1 + \gamma_{\text{primal}} \rho}$$

By choosing appropriate γ_{primal} and γ_{dual} we minimize the Huber-ROF.

3.3.5 Final Algorithm

As before, the dual updates p_ks are simple gradient ascends optimizations.

$$p_k^{t+1} = p_k^t + \rho_k(A_k u_0^{t+1} - u_k^{t+1})$$

Finally, the algorithm for the image segmentation problem is given by

Algorithm 14: Delayed ADMM for image segmentation

Initialize global u_0^0, local primal $\{u_k^0\}_{k=1}^K$ and local dual $\{p_k^0\}_{k=1}^K$
Choose appropiate $\{\rho_k\}_{k=1}^K$
for $t = 0, 1, \ldots$ **do**
  $\forall i\colon (v)_i = \frac{1}{\left(\sum_{k=1}^K A_k^\mathsf{T} A_k\right)_i} \left(\sum_{k=1}^K A_k^\mathsf{T} \hat{u}_k^t - \frac{1}{\rho_k}\left(f + \sum_{k=1}^K A_k^\mathsf{T} \hat{p}_k^t\right)\right)_i$
  $\forall i\colon (u_0^{t+1})_i \in \arg\min_{(u_0)_i \in \mathbb{R}^L} \delta\{(u_0)_i \in \Delta^{L-1}\} + \frac{\rho_k\left(\sum_{k=1}^K A_k^\mathsf{T} A_k\right)_i}{2} \|(u_0)_i - (v)_i\|^2$
  for $k = 1, \ldots, K$ **do**
    Distribute to worker k
    $\forall l\colon (u_k^{t+1})^l \in$
    $\arg\min_{(u_k)^l \in \mathbb{R}^n} \alpha \|\nabla (u_k)^l\|_{1,\delta} + \frac{\rho_k}{2} \left\| (u_k)^l - \left(A_k \hat{u}_0^{t+1} + \frac{1}{\rho_k} p_k^t\right)^l \right\|^2$
    $p_k^{t+1} = p_k^t + \rho_k(A_k \hat{u}_0^{t+1} - u_k^{t+1})$
  end
end

where $\hat{u}_0$, $\{\hat{u}_k\}$ and $\{\hat{p}_k\}$ are possibly delayed variables. We are now in a position where we can implement each update and perform experiments with delayed variables.

3.3.6 Simulated Delays

We set the "segmentation factor" $\alpha = 0.1$, the parameter for the Huber function $\delta = 0.01$ and $\rho = 10.0$. These parameters were determined experimentally to get an output similar to Figure 3.8 at the very begining of this section. Like before, we sample the delay $1 \leq \tau \leq 10$ from a constant and uniform distribution and report the output image and its scores of the augmented Lagrangian.

By taking a look at the first image results in Figure 3.10, we see that a delay of 3 already affects the image quality noticeably. The higher the delay, the more artifacts can be seen in some areas and especially on the boundary regions. This makes sense because it is more difficult to decide for a color (segment) on these regions and the delays introduce additional noise.

In Figure 3.11 we observe similar behavior in the uniform delay case but because delays are less present, the image quality is better. Especially the first row (delay 1 to 5) returns well-segmented regions even though delays can spike up to 5 in the worst case. This implies that as long as delays are not constantly present, we can "recover" from even higher delays.

Let's see how delays affect the augmented Lagrangian in Figure 3.12.

Similar to the simulated delays in the logistic regression problem, we observe a step function for the constant distribution. We also see that the delays strongly affect the scores of the augmented Lagrangian because even the lowest delay ($\tau = 2$, blue line) cannot reach the baseline when no delays are present ($\tau = 1$, dark blue line).

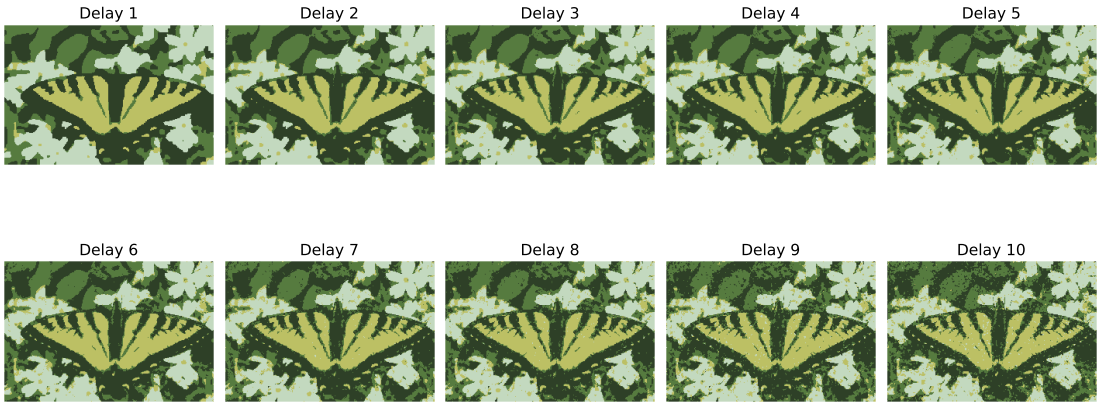


Figure 3.10: Constant delay

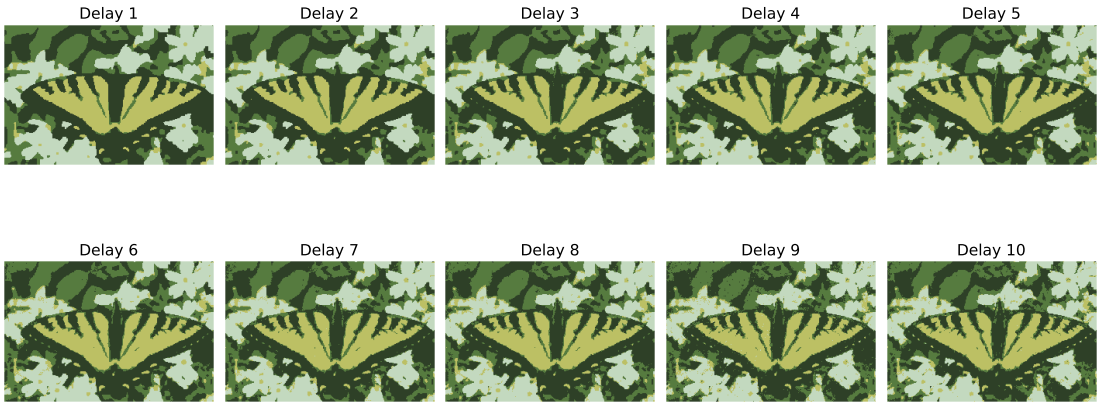


Figure 3.11: Uniform delay

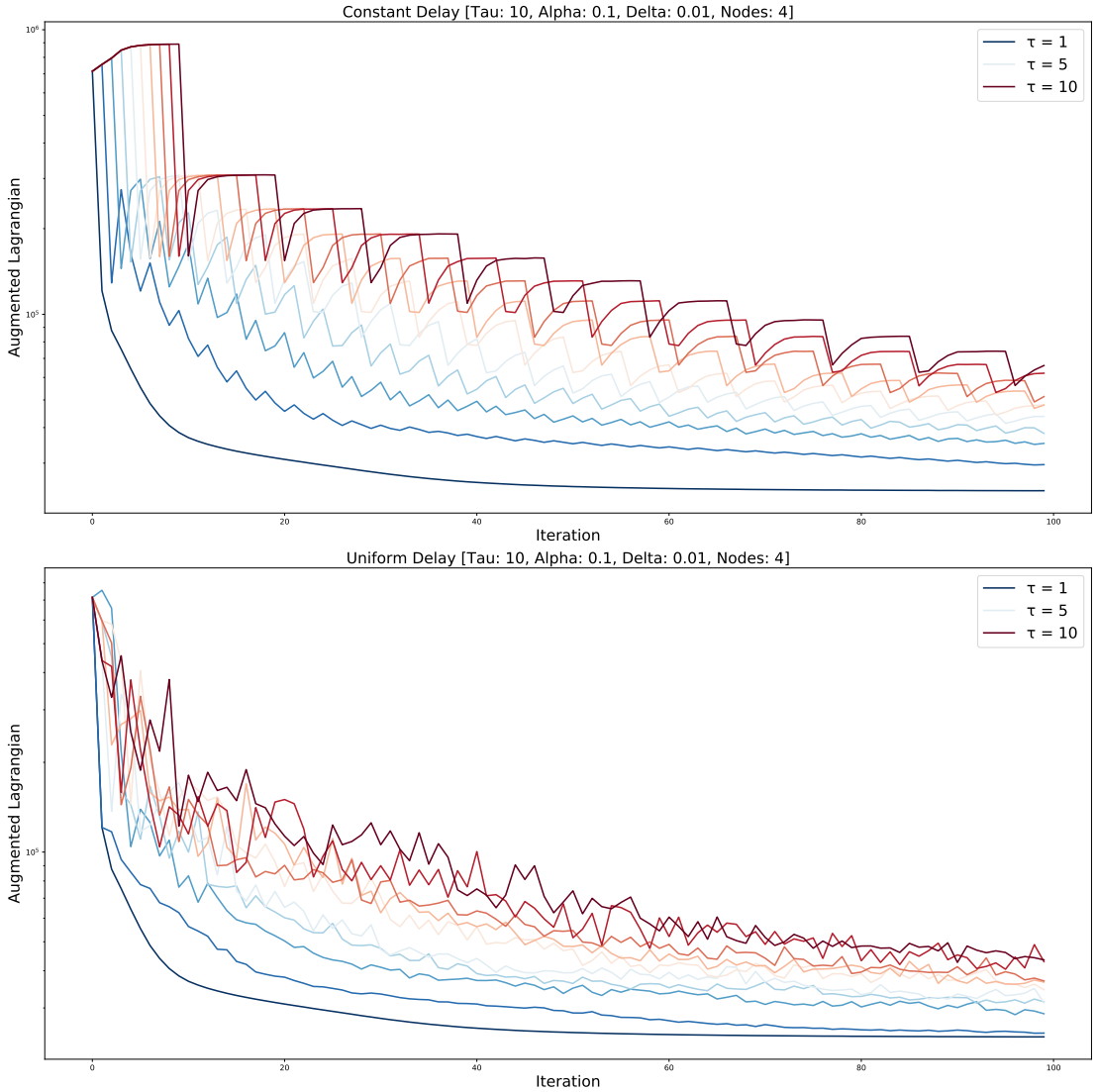


Figure 3.12: Augmented Lagrangian with different delays

3.3.7 Real Delays

Finally, we present a real asynchronous implementation for the image segmentation problem. Again, we are interested in the effect of two slow worker on the image quality and objective function. We leave the hyperparameters the same, i.e. $\rho_k \equiv \rho = 10.0$, $\alpha = 0.1$ and $\delta = 0.01$. We use a very similar partial barrier implementation as in Section 3.2.3, Algorithm 10 and 11. The full Python code can be found here[3].

First, we report that splitting the problem onto different machines improves our global runtime. In the first run, we only use one worker solving the Huber-ROF problem for the whole image. Then, we utilize 4 workers that only operate on a subset of the whole image. All runs perform 100 master iterations.

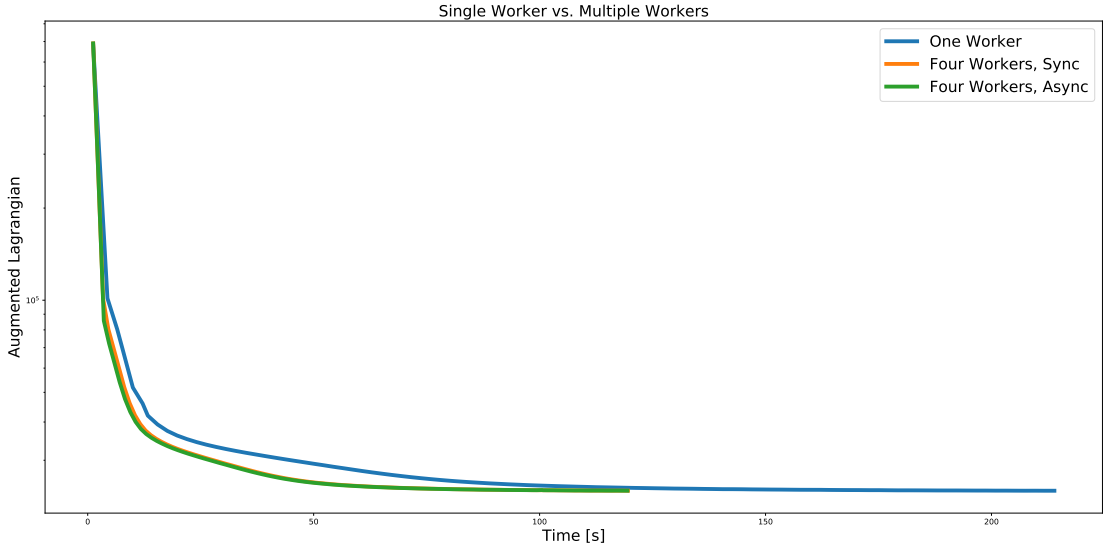


Figure 3.13: Comparison of the augmented Lagrangian between a single worker and four workers

We see that both the orange and green plot (four workers) stays below the blue plot (one worker), i.e. they minimize the objective function faster. In addition, four workers only need half of the time than a single worker to finish the experiment.

Next, to get a better understanding of the asynchronicity, we visualize the update sets in the master node that occur due to the partial barrier in Figure 3.14. We use a heatmap where each row represents a master iteration and each column represents a worker. For example, if one iteration of the master node uses the information from worker

[3] `https://github.com/filipre/master-experiments`

2 and 5, we color the cell black. We set 5 workers and let them run synchronously, asynchronously, asynchronously with a random sleep after each worker iteration and asynchronously where two workers are slowed down artificially.

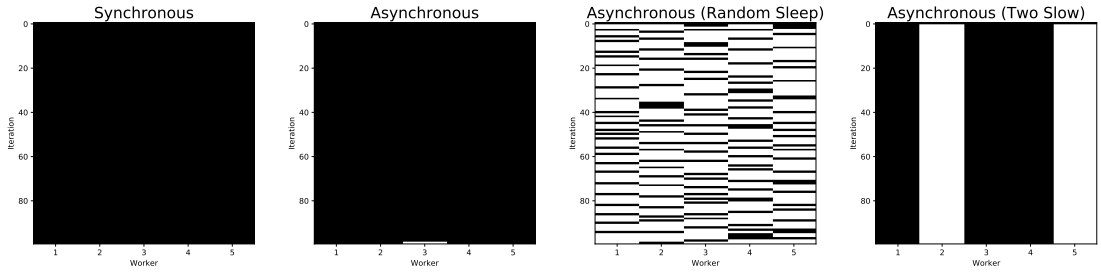


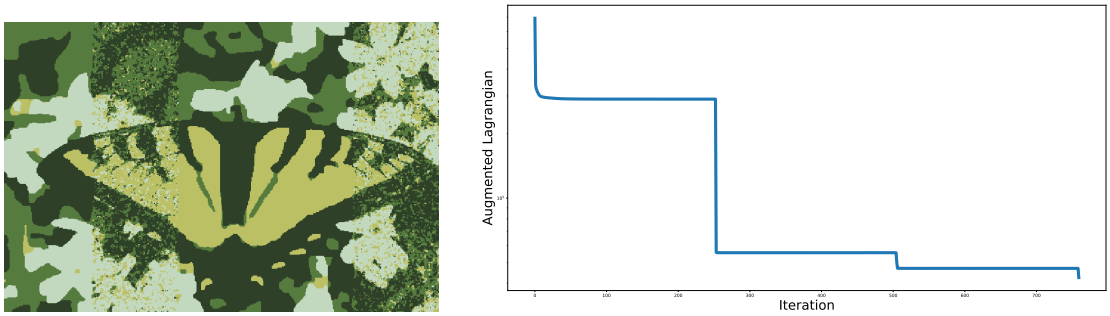
Figure 3.14: Update sets visualization

Obviously, a synchronous implementation results in a fully black heat map because each iteration is waiting for all nodes. However, we also see in the second figure that our workers are all roughly working equally fast. Even though we do not enforce synchronicity there, we end up with a result very similar to the synchronous one. By introducing a random sleep of at most 10 seconds, we observe how the workers communicate asynchronously. The last visualization shows how the update sets look like when we have two very slow workers as we can see by the large gaps in columns 2 and 5.

Especially in the first few iterations of a run, two slow workers (here workers 2 and 5) affect the image quality noticeably. Notice in Figure 3.15a how the fast workers (1, 3 and 4) already finished their total variation minimization while workers 2 and 5 still need time. If we would run the algorithm in synchronization, then the whole image quality would suffer because the fast workers would be slowed down by the slow workers.

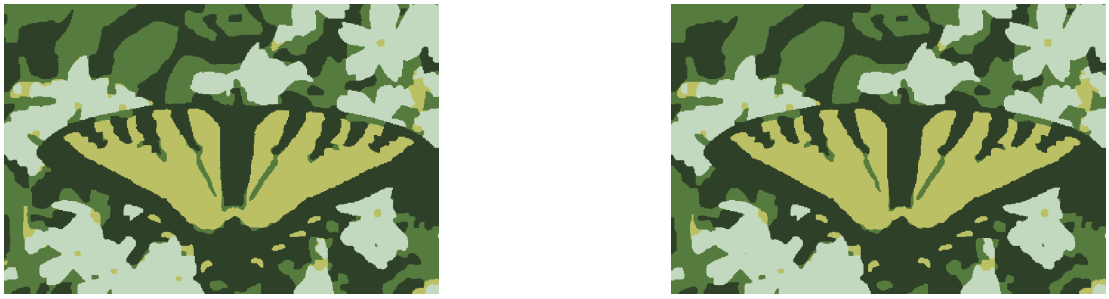
If we take a look at the augmented Lagrangian in Figure 3.15b, we observe a step function. This occurs because the fast workers reached their optimum while the slow workers still need to optimize towards it.

Nevertheless, at the end of the experiment, we still get a well-segmented image as shown in Figure 3.16 regardless of whether we run it synchronously or asynchronously.



(a) output image (b) related augmented Lagrangian

Figure 3.15: Results after only a few iterations of the 2 slow workers



(a) synchronous (b) asynchronous

Figure 3.16: Final image after at least 100 worker iterations of each (slow) worker

Interestingly enough, we also converge faster to a near optimum in the asynchronous case than in the synchronous one. In Figure 3.17 we report the augmented Lagrangian with respect to the real-world time. Even though the synchronous and asynchronous scores reach equal scores later in the experiment, at the beginning we minimize the objective function in asynchronicity faster than in synchronicity. This coincides with our observations from the “real" logistic regression experiments in Section 3.2.3.

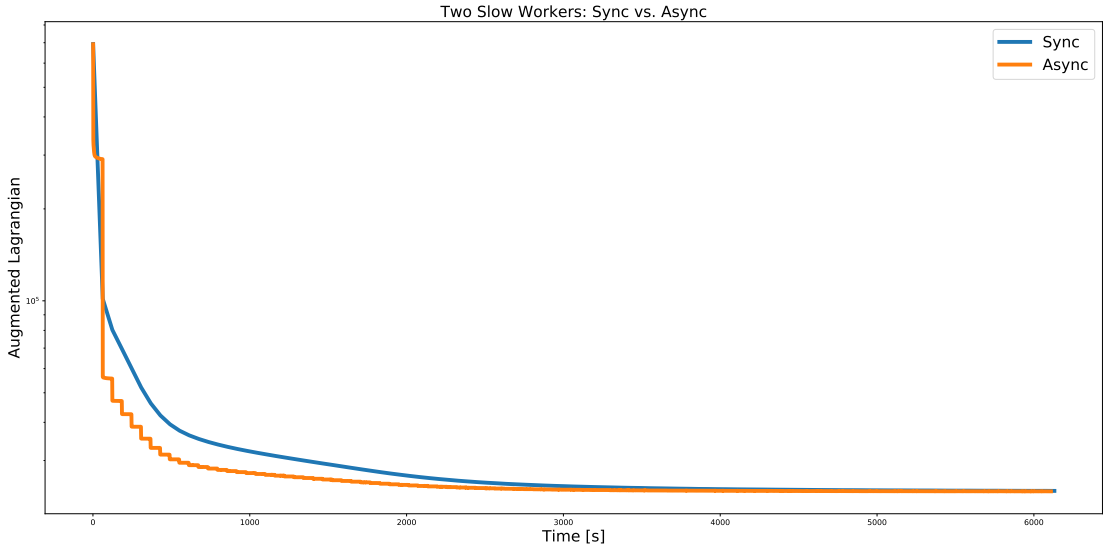


Figure 3.17: Comparison between synchronous and asynchronous mode

3.4 PyTorch's `distributed` Package

During development, it turned out that PyTorch [Pas+19] in combination with its `distributed` package is a good candidate to implement a distributed optimization algorithm. First, it uses Python as an interface language providing a simple and fast-to-learn language for researchers. Second, PyTorch already implements many useful methods for CPUs *and* GPUs which accelerated development during the thesis. And third, it provides useful methods for writing a distributed application via the `distributed` package. In this section we are going to highlight important methods from this package and explain how we used it in our logistic regression 3.2 and segmentation 3.3.7 applications.

In general, there exist two phases that need to be implemented. In the beginning, all nodes within a network must start up and connect to a "master" node. This master node can deviate from the master node defined in a distributed algorithm since communication can also happen between any two nodes and is not necessarily limited between master and workers only. Once all nodes have started and all connections have been established, the actual communication can happen. PyTorch provides methods for general point-to-point communication between two nodes or very specific "broadcast", "reduce" and "gather" operations between workers and one master. Before going into details, we would like to point out to the documentation[4] and one great tutorial[5] which can be read supplementary to this section here. We start with the firstly mentioned "start-up" phase.

Establishing communication between nodes is implemented very easily by setting 4 environment variables and calling one method only:

```
assert "WORLD_SIZE" in os.environ, "WORLD_SIZE not set"
assert "RANK" in os.environ, "RANK not set"
assert "MASTER_ADDR" in os.environ, "MASTER_ADDR not set"
assert "MASTER_PORT" in os.environ, "MASTER_PORT not set"
torch.distributed.init_process_group(backend='gloo')
```

`WORLD_SIZE` refers to the number of nodes within the network including the master node. For example, if there are 4 workers and one master node, then the value should be `5`. Next, `RANK` is the node identifier. By definition, the rank of the master node must be `0` while the rank of the other nodes should increment for each existing node in the network. The last environment variables `MASTER_ADDR` and `MASTER_PORT` define the network address of the machine that runs the master node to make a connection

[4] `https://pytorch.org/docs/stable/distributed.html`
[5] `https://pytorch.org/tutorials/intermediate/dist_tuto.html`

between one node and their master. For the master node, this can refer to `localhost` and any port. Finally, calling `torch.distributed.init_process_group(backend='gloo')` blocks until all other nodes have called this method as well and all initialization processes return successfully. PyTorch relies on a "backend" that implements the actual network communication and there exist different options depending on whether one would like to utilize CPUs or GPUs and what kind of operations (transfer of tensors vs. reduce operations) are used. In general, "NCCL" should be used for GPUs while "Gloo" is suitable for CPUs.

In our application, we only used `send` and `recv` methods for sending and receiving tensors synchronously and `isend` and `irecv` for doing the same asynchronously. Because these calls are only available for CPUs, we run the actual calculation on GPUs and transfer the tensors back to the CPU to send them. Similarly, we receive tensors on the CPU and send them to the GPUs for calculations.

```
torch.distributed.send(tensor=tensor, dst=dst, tag=tag)
torch.distributed.recv(tensor=tensor, src=src, tag=tag)
```

It is only possible to send and receive PyTorch `tensors`. In the logistic regression application, it was necessary to send and receive full PyTorch models though. Thus, we had to serialize a model first and call `send/recv` multiple times. To do this, we used the optional `tag` parameter that matches a `send` to the right `recv` command and vice versa. Both calls block until the transfer is successful or until an error occurs.

In the asynchronous case, `isend` and `irecv` behave very similar but do not block the main thread. Instead, they return a `req` request object and the user can either `req.wait()` until the transfer finishes or issue the status by calling the boolean method `req.is_completed()`.

```
# method 1
req =torch.distributed.isend(tensor=tensor, dst=dst, tag=tag)
req.wait() # blocks thread until sending/receiving finishes

# method 2
req =torch.distributed.irecv(tensor=tensor, src=src)
while not req.is_completed():
    time.sleep(1) # check every second for status of transfer.
    # do something else
```

In theory, using the second method makes it possible to perform other tasks until the new tensor is available. In practice however, due to a bug within the `req.is_completed()` method, one has to utilize a slightly different approach:

Instead of calling `req.is_completed()` from the main thread, one must call `req.wait()` in a *different* thread and wait until it terminates.

```
def daemon_thread(req):
    req.wait()

req =dist.irecv(tensor=tensor, src=src)
t =threading.Thread(target=daemon_thread, args=(req,), daemon=True)
t.start()
while t.is_alive():
    time.sleep(1) # check every second for status of transfer.
    # do something else
```

We raised this issue to the authors of PyTorch here[6]. It also mentions our temporal fix with accompanying code examples. The exact implementation used by our experiments can be found at the full code repository[7].

[6] https://github.com/pytorch/pytorch/issues/30723
[7] https://github.com/filipre/master-experiments

4 Conclusion

In this thesis, we discussed various methods to model asynchronicity in real life, for example by using a partial barrier or using variable delays within the iteration. We covered the theory of asynchronous SGD, asynchronous BCD, and an asynchronous ADMM algorithm and we have seen different strategies to show the convergence of these algorithms under the right assumptions and the right hyperparameters. In general, the convergence was depending on the Lipschitz continuity constant and the step size.

In the practical section, we introduced an ADMM algorithm with delays and we used it to solve two problems: First, a logistic regression classification problem and second, an image segmentation problem. For the latter problem, we used a (scaled) projection onto the probability simplex and a Huber-ROF solver. In a real-world implementation, we ran the algorithm on a computer cluster and observed how asynchronous coordination speeds up the computation when some nodes work slower than others. However, once delays increase too much, it does affect the quality of the results. Last but not least, we have shown how we utilized PyTorch in a distributed setting and we also provided the code that has been used throughout our experiments for replication purposes.

4.1 Further Research

Nevertheless, we recommend further research on the following topics:

1. First, it would be interesting to implement more difficult tasks that explicitly *require* distributed computing. All our experiments were simple and could be solved using a single machine only. Additionally, all distributed tasks were approximately similar difficult and we had to enforce different calculation times. Obviously, there are many problems that offer greater variety.
2. Second, given that GPUs became essential in machine learning and in deep learning specifically, shifting the focus towards shared memory architectures seems promising to cover more real-world applications.

3. Third, it is not unlikely that there exists a greater, unified theory to explain all asynchronous algorithms that we have covered in this thesis.
4. Fourth, in our thesis we only discussed *centralized* algorithms but sometimes, we can only perform our calculations *decentralized*. Even though we mentioned some decentralized algorithms, we recommend investing further research on this topic.
5. Last but not least, we see an opportunity in writing a survey on the topic of asynchronous algorithms due to the increase of interest and new methods. We also believe that many methods are very related and proving their relationships with each other appears to be very valuable to the distributed optimization community.

List of Figures

1.1 Examples of common communication topologies 3
1.2 Example calculation time of master and worker nodes 5
1.3 Partial barrier and bounded delay example 6
1.4 Possible updates by an algorithm 6
1.5 Update set perspective of the previously mentioned situation 7
1.6 Example where update sets cannot model the real world 7
1.7 Block representation for delays 8

3.1 Cross-entropy loss and accuracy with different delays and learning rates 40
3.2 Constant delay 46
3.3 Uniform delay 47
3.4 Constant delay 49
3.5 Uniform delay 50
3.6 Cross-entropy and accuracies for synchronous and asynchronous implementation depending on the category 53
3.7 Image segmentation example using 4 segments 54
3.8 One possible approach to split an image 55
3.9 Scaling examples with high weights on a single color 58
3.10 Constant delay 67
3.11 Uniform delay 67
3.12 Augmented Lagrangian with different delays 68
3.13 Comparison of the augmented Lagrangian between a single worker and four workers 69
3.14 Update sets visualization 70
3.15 Results after only a few iterations of the 2 slow workers 71
3.16 Final image after at least 100 worker iterations of each (slow) worker . . 71
3.17 Comparison between synchronous and asynchronous mode 72

List of Tables

1.1 Short summary of distributed algorithms for various situations 10

List of Algorithms

1 Stochastic Gradient Descent 12
2 Alternating Direction Method of Multipliers 15
3 ADMM for Consensus Problem 16
4 Asynchronous Stochastic Gradient (AsySG-con) 17
5 Asynchronous Block Coordinate Descent 26
6 Flexible ADMM 32
7 Delayed ADMM 37

8 Delayed ADMM for logistic regression with Lagrangian multipliers . . . 43
9 Delayed ADMM for logistic regression without Lagrangian multipliers . 44
10 Master algorithm for logistic regression 51
11 Worker k algorithm for logistic regression 52
12 Scaled Projection onto Probability Simplex 58
13 First-Order Primal-Dual Algorithm 64
14 Delayed ADMM for image segmentation 65

Bibliography

[BCN18a] L. Bottou, F. Curtis, and J. Nocedal. "Optimization Methods for Large-Scale Machine Learning." In: *SIAM Review* 60.2 (2018), pp. 223–311. DOI: 10.1137/16M1080173. eprint: https://doi.org/10.1137/16M1080173.

[BCN18b] L. Bottou, F. E. Curtis, and J. Nocedal. "Optimization methods for large-scale machine learning." In: *Siam Review* 60.2 (2018), pp. 223–311.

[BDX03] S. Boyd, P. Diaconis, and L. Xiao. "Fastest Mixing Markov Chain on A Graph." In: *SIAM REVIEW* 46 (2003), pp. 667–689.

[Ber15] D. P. Bertsekas. "Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey." In: *arXiv e-prints*, arXiv:1507.01030 (July 2015), arXiv:1507.01030. arXiv: 1507.01030 [cs.SY].

[BKC17] V. Badrinarayanan, A. Kendall, and R. Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation." In: *IEEE transactions on pattern analysis and machine intelligence* 39.12 (2017), pp. 2481–2495.

[Boy+11] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers." In: *Foundations and Trends® in Machine learning* 3.1 (2011), pp. 1–122.

[CP11] A. Chambolle and T. Pock. "A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging." In: *Journal of Mathematical Imaging and Vision* 40.1 (2011), pp. 120–145. DOI: 10.1007/s10851-010-0251-1.

[Duc+08] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. "Efficient projections onto the l 1-ball for learning in high dimensions." In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 272–279.

[Far+19] F. Farina, A. Garulli, A. Giannitrapani, and G. Notarstefano. "A distributed asynchronous method of multipliers for constrained nonconvex optimization." In: *Automatica* 103 (2019), pp. 243–253. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2019.02.003.

[FSS15] F. Facchinei, G. Scutari, and S. Sagratella. "Parallel Selective Algorithms for Nonconvex Big Data Optimization." In: *IEEE Transactions on Signal Processing* 63.7 (Apr. 2015), pp. 1874–1889. ISSN: 1053-587X. DOI: 10.1109/TSP.2015.2399858.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[HC17] M. Hong and T. Chang. "Stochastic Proximal Gradient Consensus Over Random Networks." In: *IEEE Transactions on Signal Processing* 65.11 (June 2017), pp. 2933–2948. ISSN: 1053-587X. DOI: 10.1109/TSP.2017.2673815.

[HLR16a] M. Hong, Z. Luo, and M. Razaviyayn. "Convergence Analysis of Alternating Direction Method of Multipliers for a Family of Nonconvex Problems." In: *SIAM Journal on Optimization* 26.1 (2016), pp. 337–364. DOI: 10.1137/140990309. eprint: https://doi.org/10.1137/140990309.

[HLR16b] M. Hong, Z.-Q. Luo, and M. Razaviyayn. "Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems." In: *SIAM Journal on Optimization* 26.1 (2016), pp. 337–364.

[Hon18] M. Hong. "A Distributed, Asynchronous, and Incremental Algorithm for Nonconvex Optimization: An ADMM Approach." In: *IEEE Transactions on Control of Network Systems* 5.3 (Sept. 2018), pp. 935–945. ISSN: 2325-5870. DOI: 10.1109/TCNS.2017.2657460.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning." In: *Nature* 521 (May 2015), 436 EP -.

[Lia+15] X. Lian, Y. Huang, Y. Li, and J. Liu. "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization." In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 2737–2745.

[Lia+17] X. Lian, W. Zhang, C. Zhang, and J. Liu. "Asynchronous Decentralized Parallel Stochastic Gradient Descent." In: *arXiv e-prints*, arXiv:1710.06952 (Oct. 2017), arXiv:1710.06952. arXiv: 1710.06952 [math.OC].

[Liu+14] J. Liu, S. Wright, C. Ré, V. Bittorf, and S. Sridhar. "An asynchronous parallel stochastic coordinate descent algorithm." In: *International Conference on Machine Learning*. 2014, pp. 469–477.

[LW14] J. Liu and S. J. Wright. "Asynchronous Stochastic Coordinate Descent: Parallelism and Convergence Properties." In: *arXiv e-prints*, arXiv:1403.3862 (Mar. 2014), arXiv:1403.3862. arXiv: 1403.3862 [math.OC].

[LWC19] E. Laude, T. Wu, and D. Cremers. "Optimization of Inf-Convolution Regularized Nonconvex Composite Problems." In: *Proceedings of Machine Learning Research*. Ed. by K. Chaudhuri and M. Sugiyama. Vol. 89. Proceedings of Machine Learning Research. PMLR, 16–18 Apr 2019, pp. 547–556.

[Mob02] R. K. Mobley. *An introduction to predictive maintenance*. Elsevier, 2002.

[NC07] M. Nikolova and R. H. Chan. "The equivalence of half-quadratic minimization and the gradient linearization iteration." In: *IEEE Transactions on Image Processing* 16.6 (2007), pp. 1623–1627.

[Niu+11] F. Niu, B. Recht, C. Ré, and S. J. Wright. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." In: *NIPS* 24 (June 2011).

[NO09] A. Nedic and A. Ozdaglar. "Distributed Subgradient Methods for Multi-Agent Optimization." In: *IEEE Transactions on Automatic Control* 54.1 (Jan. 2009), pp. 48–61. ISSN: 0018-9286. DOI: 10.1109/TAC.2008.2009515.

[NOS17] A. Nedich, A. Olshevsky, and A. Shi. "Achieving geometric convergence for distributed optimization over time-varying graphs." English (US). In: *SIAM Journal on Optimization* 27.4 (Jan. 2017), pp. 2597–2633. ISSN: 1052-6234. DOI: 10.1137/16M1084316.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[PB+14] N. Parikh, S. Boyd, et al. "Proximal algorithms." In: *Foundations and Trends® in Optimization* 1.3 (2014), pp. 127–239.

[Pen+15] Z. Peng, Y. Xu, M. Yan, and W. Yin. "ARock: an Algorithmic Framework for Asynchronous Parallel Coordinate Updates." In: *arXiv e-prints*, arXiv:1506.02396 (June 2015), arXiv:1506.02396. arXiv: 1506.02396 [math.OC].

[RM51] H. Robbins and S. Monro. "A stochastic approximation method." In: *The annals of mathematical statistics* (1951), pp. 400–407.

[ROF92] L. I. Rudin, S. Osher, and E. Fatemi. "Nonlinear total variation based noise removal algorithms." In: *Physica D: nonlinear phenomena* 60.1-4 (1992), pp. 259–268.

[Ros58] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[Shi+14] W. Shi, Q. Ling, K. Yuan, G. Wu, and W. Yin. "On the Linear Convergence of the ADMM in Decentralized Consensus Optimization." In: *IEEE Transactions on Signal Processing* 62.7 (Apr. 2014), pp. 1750–1761. ISSN: 1053-587X. DOI: 10.1109/TSP.2014.2304432.

[Shi+15a] W. Shi, Q. Ling, G. Wu, and W. Yin. "A Proximal Gradient Algorithm for Decentralized Composite Optimization." In: *IEEE Transactions on Signal Processing* 63.22 (Nov. 2015), pp. 6013–6023. ISSN: 1053-587X. DOI: 10.1109/TSP.2015.2461520.

[Shi+15b] W. Shi, Q. Ling, G. Wu, and W. Yin. "EXTRA: An Exact First-Order Algorithm for Decentralized Consensus Optimization." In: *SIAM Journal on Optimization* 25.2 (2015), pp. 944–966. DOI: 10.1137/14096668X. eprint: https://doi.org/10.1137/14096668X.

[SHY17] T. Sun, R. Hannah, and W. Yin. "Asynchronous Coordinate Descent under More Realistic Assumptions." In: *arXiv e-prints*, arXiv:1705.08494 (May 2017), arXiv:1705.08494. arXiv: 1705.08494 [math.OC].

[Tse01] P. Tseng. "Convergence of a block coordinate descent method for nondifferentiable minimization." In: *Journal of optimization theory and applications* 109.3 (2001), pp. 475–494.

[Vir+20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. 1. 0. Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* (2020). DOI: https://doi.org/10.1038/s41592-019-0686-2.

[WC13] W. Wang and M. A. Carreira-Perpinán. "Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application." In: *arXiv preprint arXiv:1309.1541* (2013).

[WO13] E. Wei and A. Ozdaglar. "On the O(1/k) Convergence of Asynchronous Distributed Alternating Direction Method of Multipliers." In: *arXiv e-prints*, arXiv:1307.8254 (July 2013), arXiv:1307.8254. arXiv: 1307.8254 [math.OC].

[Wu+18] T. Wu, K. Yuan, Q. Ling, W. Yin, and A. H. Sayed. "Decentralized Consensus Optimization With Asynchrony and Delays." In: *IEEE Transactions on Signal and Information Processing over Networks* 4.2 (June 2018), pp. 293–307. ISSN: 2373-776X. DOI: 10.1109/TSIPN.2017.2695121.

[XB04] L. Xiao and S. Boyd. "Fast linear iterations for distributed averaging." In: *Systems & Control Letters* 53.1 (2004), pp. 65–78. ISSN: 0167-6911. DOI: https://doi.org/10.1016/j.sysconle.2004.02.022.

[Yan+19] T. Yang, X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin, and K. H. Johansson. "A survey of distributed optimization." In: *Annual Reviews in Control* 47 (2019), pp. 278–305. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2019.05.006.

[YLY16] K. Yuan, Q. Ling, and W. Yin. "On the Convergence of Decentralized Gradient Descent." In: *SIAM Journal on Optimization* 26.3 (2016), pp. 1835–1854. DOI: 10.1137/130943170. eprint: https://doi.org/10.1137/130943170.

[ZC08] M. Zhu and T. Chan. "An efficient primal-dual hybrid gradient algorithm for total variation image restoration." In: *UCLA CAM Report* 34 (2008).

[Zhe+16] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z. Ma, and T. Liu. "Asynchronous Stochastic Gradient Descent with Delay Compensation for Distributed Deep Learning." In: *CoRR* abs/1609.08326 (2016). arXiv: 1609.08326.

[ZK14] R. Zhang and J. T. Kwok. "Asynchronous Distributed ADMM for Consensus Optimization." In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. II-1701–II-1709.

[ZY18] J. Zeng and W. Yin. "On Nonconvex Decentralized Gradient Descent." In: *IEEE Transactions on Signal Processing* 66.11 (June 2018), pp. 2834–2848. ISSN: 1053-587X. DOI: 10.1109/TSP.2018.2818081.