Black-Box Parallelization for Machine Learning

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt
von
Michael Kamp
aus
Bonn

Bonn, 2018

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Stefan Wrobel
2. Gutachter: Prof. Dr. Thomas Gärtner
Tag der Promotion: 13.09.2019
Erscheinungsjahr: 2019

Michael Kamp

University of Bonn
Department of Computer Science III

and

Fraunhofer Institute for Intelligent Analysis
and Information Systems IAIS

Declaration

Acknowledgments

I would like to thank my supervisors Stefan Wrobel and Mario Boley for their thoughtful advice, the freedom and trust they granted me, and for the opportunity to work in such an inspiring environment with such remarkable colleagues, both at the department Knowledge Discovery at Fraunhofer IAIS and at the University of Bonn. It was a real pleasure working with Mario and I am sincerely grateful for his guidance, his patience, and his scientific rigor.

I was lucky enough to be surrounded by a group of great people, first and foremost the research group CAML. I want to thank Thomas Gärtner for creating this exciting, fruitful, and fun environment and for his endless advice. Daniel Paurat for our discussions that really shaped my intuitive understanding of machine learning—not to mention all the coffee. Linara Adilova for her great support with the theoretical and practical aspects of deep learning. Roman Garnett, Nikos Giatrakos, Tamas Horvath, Daniel Keren, Kristian Kersting, Michael May, Michael Mock, Henning Petzka, Stefan Rüping, and Izchak Sharfman for all the quality discussions. The other PhD students Babak Ahmadi, Sven Giesselbach, Fabian Hadiji, Dirk Hecker, Olana Missura, Marion Neumann, Dino Oglic, Nathalie Paul, Anja Pilz, Jil Sander, Till Schulz, Florian Seiffhart, Daniel Trabold, Katrin Ullrich, Mirwaes Wahabzada, and Pascal Welke, Dorina Weichert and some really bright and inspiring students I met over the last couple of years. Thanks to you, given the chance, I would do it all over again.

Special thanks go to Linara Adilova, Thomas Gärtner, Karl-Heinz Kamp, Stephanie Kamp, Nathalie Paul, Daniel Paurat, Jil Sander, Daniel Trabold, and Pascal Welke for reading various drafts of this thesis and giving me valuable feedback.

I would like to thank my father Karl-Heinz Kamp and my sister Stephanie Kamp for their trust and support. Last but not least, I want to thank my partner Vanessa Corzelius for her strength, patience, constant support, and motivation—without her, I would not have been able to complete this thesis—as well as our daughter Sophia and her unborn sister for being the final motivation for completing this thesis.

Abstract

The landscape of machine learning applications is changing rapidly: large centralized datasets are replaced by high volume, high velocity data streams generated by a vast number of geographically distributed, loosely connected devices, such as mobile phones, smart sensors, autonomous vehicles or industrial machines. Current learning approaches centralize the data and process it in parallel in a cluster or computing center. This has three major disadvantages: (i) it does not scale well with the number of data-generating devices since their growth exceeds that of computing centers, (ii) the communication costs for centralizing the data are prohibitive in many applications, and (iii) it requires sharing potentially privacy-sensitive data. Pushing computation towards the data-generating devices alleviates these problems and allows to employ their otherwise unused computing power. However, current parallel learning approaches are designed for tightly integrated systems with low latency and high bandwidth, not for loosely connected distributed devices. Therefore, I propose a new paradigm for parallelization that treats the learning algorithm as a black box, training local models on distributed devices and aggregating them into a single strong one. Since this requires only exchanging models instead of actual data, the approach is highly scalable, communication-efficient, and privacy-preserving.

Following this paradigm, this thesis develops black-box parallelizations for two broad classes of learning algorithms. One approach can be applied to incremental learning algorithms, i.e., those that improve a model in iterations. Based on the utility of aggregations it schedules communication dynamically, adapting it to the hardness of the learning problem. In practice, this leads to a reduction in communication by orders of magnitude. It is analyzed for (i) online learning, in particular in the context of in-stream learning, which allows to guarantee optimal regret and for (ii) batch learning based on empirical risk minimization where optimal convergence can be guaranteed. The other approach is applicable to non-incremental algorithms as well. It uses a novel aggregation method based on the Radon point that allows to achieve provably high model quality with only a single aggregation. This is achieved in polylogarithmic runtime on quasi-polynomially many processors. This relates parallel machine learning to Nick's class of parallel decision problems and is a step towards answering a fundamental open problem about the abilities and limitations of efficient parallel learning algorithms. An empirical study on real distributed systems confirms the potential of the approaches in realistic application scenarios.

Contents

1. Introduction

"There can be no center in infinity." (Titus Lucretius Carus, De rerum natura)

Machine learning refers to a class of algorithms that automatically improve their performance for solving a given task based on data. It extracts information from large datasets to produce or improve a model describing the relations governing the data. With the widespread adoption of smart phones and other mobile devices, the instrumentation of our world with smart sensors, the automation of manufacturing machines, and the advent of autonomous driving in the 2010s, the majority of data is generated by a vast amount of loosely connected distributed devices—a phenomenon that has been called the Internet of Things with an estimate of $50.1 \cdot 10^9$ connected devices by 2020 (Mohan and Kangasharju, 2016)). This data is too large and generated too fast to be processed centrally in a system like a cluster or computing cloud, since the network connection often has too low bandwidth and too high latency. For example, an autonomous car generates over 1 Gigabyte of sensor data per second (Shi et al., 2016), but the bandwidth of current mobile connections (e.g., 4G with 300 Mbps) only allows to transmit 37.5 Megabyte in that time. For real-time applications over high velocity datastreams such as online advertisement (Muthukrishnan, 2009) and financial predictions (Kearns and Nevmyvaka, 2013), the time required for communicating the data and receiving a result is prohibitive: In 2010, a fiber-optics cable was constructed between the financial markets in New York and Chicago for $300 million, just to reduce the round-trip communication time from 16 to 13 milliseconds (the cable was later replaced by microwave transmission technology, reducing round-trip time to 8.1 milliseconds) (Budish et al., 2015). Communication furthermore consumes a lot of power and thereby substantially reduces the runtime of battery-powered devices. Moreover, a lot of the data should not be shared since it is privacy-sensitive: sharing audio and video recordings from mobiles violate the user's privacy, and extracting sensor data from industrial machines infringes company secrets.

At the same time, the data-generating devices often have computing power on their own. In order to harness this power, computation is pushed from the cluster towards the devices—an approach termed in-situ processing, edge or fog computing. It avoids the communication bottleneck and allows to handle data for which centralization is infeasible. Only sharing results of locally processed data furthermore can preserve its privacy.

In order to perform machine learning on, or close to the data-generating devices, existing learning algorithms need to be transformed into distributed ones, a process called **parallelization**[1]. In distributed computing, an algorithm is parallelized by analyzing it carefully, identifying independent parts, and reimplementing it in such a way that these parts are executed in parallel. The result of the parallel algorithm is up to machine precision identical to one computed by the serial one. I refer to this form as classical parallelization. Typically, the parallel algorithm requires a high amount of communication to exchange data and intermediate results between the processors. Thus, this form of parallelization requires a tightly coupled system—such as a clusters or computing cloud—so that the communication does not stall the algorithm. Since current distributed learning approaches are typically designed for such tightly coupled systems (Coates et al., 2013; Dekel et al., 2012; Dennis et al., 2018; Hardy et al., 2017; Sparks et al., 2013), they are not suitable for learning on the data-generating devices.

An approach to handle loosely connected distributed systems with low bandwidth and high latency is to run the learning algorithm close to the data-generating devices. The models produced locally are then communicated and aggregated into a single (hopefully better) model (e.g., Lin et al., 2017; Mcdonald et al., 2009; McMahan et al., 2017; Zinkevich et al., 2010). For example in autonomous driving with a fleet of $m \in \mathbb{N}$ vehicles, each vehicle $i \in [m]$ records sensor readings that form local datasets E^i. The learning algorithm $\mathcal{A}$ is applied within the vehicle to its local dataset to produce a local model $f^i = \mathcal{A}(E^i)$. Since the local dataset is only a fraction of the entire data, the quality of the local model is typically low. By sending the local models $f^1, \dots, f^m$ to a central computing system and aggregating them, a single, high quality model can be obtained. In contrast to classical parallelizations, this approach does not necessarily result in the same output as the serial execution of the learning algorithm. The goal is to find an aggregation technique that is generally applicable and achieves high model quality at the same time.

To emphasize the necessity of such generic parallelizations, the following example illustrates the differences between classical and black-box parallelizations. A simple learning algorithm is ordinary least squares regression (OLS). Given a set of instances $x \in \mathbb{R}^d$ from a d-dimensional real-valued vector space, the goal is to predict labels $y \in \mathbb{R}$ using a model f_w parameterized by a vector $w \in \mathbb{R}^d$, i.e., $y = f_w(x) = \langle w, x \rangle$. Given a dataset

$$E = \{(x_1, y_1), \dots, (x_N, y_N)\} \ ,$$

the goal is to find the parameters $w \in \mathbb{R}^d$ of the model such that it minimizes the squared error

$$w = \arg \min_{w' \in \mathbb{R}^d} \sum_{j=1}^{N} \left(\langle w', x_j \rangle - y \right)^2 \ . \tag{1.1}$$

[1] Since the goal is to train on data-generating devices, parallelization here refers only to data parallelism. In contrast, in model parallelism all data is available to all processing nodes and different parts of the model are optimized in parallel.

Algorithm 1 Parallel QR-Decomposition (Demmel et al., 2012) (simplified)

```
Input: $m = 2^n$ processors for some $n \in \mathbb{N}$, local datasets $X^1, \dots, X^m$
at each learner $i \in [m]$:
  compute QR-decomposition $X^i = Q^{i,0}R^{i,0}$
  for $k$ from 1 to $\log m$ do
    if $i \mod 2^k = 2^{k-1}$ then
      send $Q^{i,k-1}$ and $R^{i,k-1}$ to processor $i + 2^{k-1}$
    end if
    if $i \mod 2^k = 0$ then
      receive $R^{i-2^{k-1},k-1}$
      stack into the matrix $C^{i,k} = \left(R^{i-2^{k-1},k-1}, R^{i,k-1}\right)$
      compute QR-decomposition $C^{i,k} = Q^{i,k}R^{i,k}$
    end if
  end for
  if $i = m$ then
    for all $k \in [m]$ set $\mathbf{Q}^k \leftarrow \left(Q^{1\cdot 2^{k-1},k}, Q^{2\cdot 2^{k-1}2,k}, \dots, Q^{m,k}\right)$
    compute $Q \leftarrow \Pi_{k=1}^{\log m} I_{2^{\log m - k+1}} \mathbf{Q}^k$
    return $Q, R^{m,\log m}$
  end if
```

Let $X = (x_1, \dots, x_N)$ be the matrix of instances and $y = (y_1, \dots, y_N)^\top$ the vector of labels. The ordinary least squares approach solves

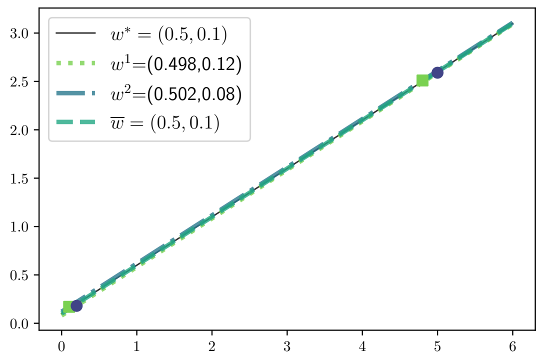
$$w = [X^\top X]^{-1} X^\top y .$$

This can be solved using the QR-decomposition of X, i.e., $X = QR$ with orthogonal matrix Q and upper triangular matrix R, so that the solution is given by $w = R^{-1}Q^\top y$.

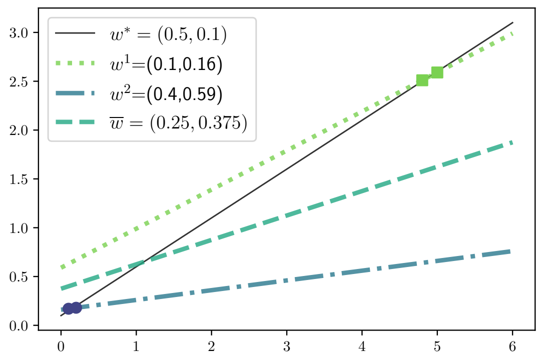
Now assume that this dataset is evenly distributed over $m \in \mathbb{N}$ processing units with each $i \in [m]$ holding a dataset X^i, y^i. Demmel et al. (2012) propose an efficient parallelization of the QR-decomposition, given in Algorithm 1. The resulting QR-decomposition is up to machine precision identical to one obtained by serial QR-decomposition. The method requires $\log_2 m$ parallel QR-decompositions and is communication heavy, sending $m-1$ messages, each containing a matrix of size $(N/m) \times d$.

Algorithm 2 Averaging-At-The-End

```
Input: learning algorithm $\mathcal{A}$, $m$ processors, local datasets $X^1, \dots, X^m$
  compute $w^i \leftarrow \mathcal{A}(X^i)$ at processor $i \in [m]$ in parallel
  return $w \leftarrow \frac{1}{m}\sum_{i=1}^m w^i$
```



(a)



(b)

Figure 1.1.: Averaging-at-the-end (Algorithm 2) for two partitions of the dataset $X = (0.1, 0.2, 4.8, 5.0)^\top$, $y = (0.17, 0.18, 2.51, 2.59)$, where y was generated by a function $f(x) = w_0^* x + w_1^*$ plus noise with $w* = (0.5, 0.1)$.

Instead, consider the following simple black-box parallelization: Obtain local model parameters w^i from the local datasets X^i, y^i using an arbitrary learning algorithm $\mathcal{A}$ (e.g., using QR-decompositions, or gradient descent) and average the parameters (see Algorithm 2). The approach communicates model parameters once at the end, thus requiring only little communication. Since all computation is performed locally, except for the averaging at the end, the speedup is higher than that of the parallel QR-decomposition. However, it does not result in the same parameters as the serial application of the algorithm. On the contrary, averaging once at the end is susceptible to the quality of local models: Figure 1.1 shows the results of averaging for two different partitions of the data, one that provides good results (Figure 1.1(a)), the other (Figure 1.1(b)) for which the resulting model is bad.

This averaging-at-the-end approach was introduced to achieve high speedup and communication efficiency (Mcdonald et al., 2009; Zinkevich et al., 2010), but it turned out that it can lead to arbitrarily bad results (Shamir and Srebro, 2014). However, guarantees on the model quality are crucial for the confident application of these approaches in practice. Therefor, a parallelization is required that is applicable to a broad class of learning algorithms with theoretical guarantees on the quality of their output. At the same time, the speedup through parallelization should scale well with the number of employed processing units. Moreover, it needs to be communication-efficient to be run on the loosely-connected data-generating devices. Given the vast amount of existing, specialized machine learning algorithms, being able to parallelize them for those devices in such a generic way could enable a technology leap.

The remainder of the introduction motivates black-box parallelization and formulates these requirements. It then surveys the contributions of this thesis.

1.1. Requirements for Black-Box Parallelizations

This thesis proposes black-box parallelization as a novel paradigm for distributed machine learning. Such parallelizations aggregate local models generated by a learning algorithm in parallel. The aggregation can be performed once at the end, or scheduled multiple times during the training process. A specific combination of the type of aggregation and its scheduling defines an approach within the paradigm and is later called a distributed learning protocol. The advantage of these protocols is that they can be readily applied to a wide range of learning algorithms without re-implementing them. The protocols should have only minimal constraints on the learning algorithms they can be applied to, such as: (i) the models used have a representation that allows for aggregation, (ii) the nature of the learning problem is benign (e.g., it can be formulated as a convex optimization problem), and (iii) the learning algorithm must produce reasonable outputs (e.g., models that are good with high probability, or model updates that change proportional to their errors). With this, the learning performance of a protocol can be theoretically assessed[2].

As mentioned above, such a protocol does not necessarily produce the same output as the serial execution of the learning algorithm. Instead, the goal is to obtain a model with similar quality. However, this quality is often unobservable, e.g., the generalization error of a model. For batch learning, i.e., learning from a given dataset with random access to its elements, the empirical risk minimization model (see Shalev-Shwartz and Ben-David, 2014) allows to provide high-probability guarantees on the model quality. Similarly, the online learning model (see Cesa-Bianchi and Lugosi, 2006), i.e., learning from a sequence of examples presented to the algorithm, allows to provide worst-case guarantees on the in-place performance of a model. Thus, in cases the model quality is unobservable, black-box parallelizations should achieve the same guarantees on model quality as the serial execution of the learning algorithm.

At the same time, the speedup through parallelization needs to scale well with the number of employed processing units. The best theoretical speedup is linear in the number of processors, but can rarely be achieved: Amdahl's law (Amdahl, 1967) states that with an increasing number of processors, the parts of the algorithm that cannot be parallelized dominate the runtime. A more realistic speedup is given by Nick's class $\mathcal{NC}$ (Greenlaw et al., 1995). It contains decision problems that can be decided in polynomial time, for which a parallelization exists that has polylogarithmic runtime on polynomially many processing units. Similarly, PAC learning problems with such parallelizations are denoted $\mathcal{NC}$-learnable (Long and Servedio, 2011; Vitter and Lin, 1992). A slight relaxation allows to use quasi-polynomially many processors. Consequently, the goal for black-box parallelizations is to achieve polylogarithmic runtime on (quasi-)polynomially many processors

Moreover, the parallelization should achieve high quality and speedup in a communication-efficient way. The amount of communication is given by the amount required for each aggregation and the number of aggregations. Using a dedicated coordinator, local models can be collected and aggregated with only one message per processing unit; aggregation using a tree-like network requires the same amount of messages. Thus, the communication per aggre-

[2] To determine the communication and speedup of a protocol, in addition the network topology of the distributed system and algorithmic details have to be taken into account.

gation should scales at most linearly with the number of processing units. For some learning problems it suffices to aggregate models once and the communication-efficiency only depends on the cost for the aggregation. Others require many iterations over the local datasets to improve local models with multiple aggregations during those iterations. Given that aggregations are beneficial, communication can be invested to improve the models by aggregating more frequently. To be communication-efficient, the amount of communication should depend on its utility: the harder the learning problem, the more communication should be invested. Conversely, for a very simple problem only little communication should be necessary. Therefore, the goal is that the communication is also bounded in the hardness of the learning task.

It is an open problem, whether it is possible to parallelize a broad class of machine learning algorithms in a generic way such that

(R1) the model trained in parallel has a quality similar to one obtained by the serial application of the learning algorithm,

(R2) the parallel algorithm achieves high speedup; ideally it has polylogarithmic runtime on (quasi-)polynomially many processing units, and

(R3) the parallel execution requires communication that is at most linear in the number of nodes and adaptive to the hardness of the learning problem.

Existing black-box parallelizations so far fail to achieve all three goals at once: some provide a good speedup and low communication (e.g., Mcdonald et al., 2009; McMahan et al., 2017), but no theoretical guarantees on the solution quality; others do provide a theoretical analysis (e.g., Lin et al., 2017; Zhang et al., 2013) but their speedup is so low that it is doubted whether they have any benefit over learning on a single chunk (Shamir and Srebro, 2014). Moreover, none of them is able to adapt the communication to the hardness of the learning problem. This thesis develops black-box parallelizations with the goal to achieve all three requirements at once. The following section outlines these contributions.

1.2. Contributions

The main contributions of this thesis are two protocols. The first can be applied to incremental learning algorithms. Based on the utility of aggregations it schedules communication dynamically. It is analyzed for (i) online learning, in particular in the context of in-stream learning and for (ii) batch learning seen as empirical risk minimization. The other one is applicable to non-incremental algorithms as well. It includes a novel aggregation method based on the Radon point (Radon, 1921) that allows to achieve high model quality with only a single aggregation. In addition, a distributed online learning framework was implemented on top of Apache Storm (Apache Software Foundation, 2017) using the first protocol (see Chapter 3), as well as an extension to Apache Spark (Sparks et al., 2013) that uses the second protocol (see Chapter 5). These implementations are presented in Appendix A. In the following, the individual contributions are detailed. After that, some additional contributions are presented that are a byproduct of the main results.

Communication-Efficient Distributed Online Learning: In distributed online learning a set of local processing nodes runs an online learning algorithm in parallel and individually provides predictions. Their quality can be increased by aggregating local models and redistributing the aggregate. The idea is to only aggregate in states where local models have a high divergence, measured by the average distance of local models to their aggregate. If this distance is high, aggregation is most impactful. Monitoring it directly would require constant communication. Instead, by choosing an aggregation method that minimizes this average distance, local conditions can be constructed that allow to monitor the divergence in a communication-efficient way. This approach is studied for averaging as aggregation operator.

The approach, denoted dynamic averaging, allows to dynamically adjust communication to the current hardness of the learning problem. This property is denoted adaptivity. It is shown that no periodic protocol can be efficient, i.e., achieve optimal predictive performance and adaptivity at the same time. Analyzing dynamic averaging in the online learning model allows to show that its regret—that is, its excess in-place error over the best model in hindsight—is in the same order as any periodically averaging protocol. This even holds for scenarios with concept drifts. By furthermore showing that periodic averaging achieves optimal regret for specific learning algorithms it follows that dynamic averaging indeed can achieve the predictive performance of the serial learning algorithm. At the same time, its communication can be bounded in the cumulative error of all processing nodes, thereby showing that dynamic averaging is efficient. The approach is applied to linear and kernel models, as well as neural networks. It is discussed in Chapter 3.

Dynamic Distributed Batch Learning with Incremental Algorithms: In batch learning, the goal is to produce a single model that performs well on unseen data. Incremental learning algorithms update a model in rounds, improving its quality with each iteration. Applying dynamic averaging to incremental algorithms allows to obtain a model with quality similar to a serially computed one in a communication-efficient way. For that, it is shown that dynamic averaging retains optimal convergence rates for certain incremental learning algorithms. Analyzing the resulting model in the empirical risk minimization model then allows to provide generalization bounds for it. The speedup of dynamic averaging over the serial application of the learning algorithm scales well with the number of processing units, i.e., the speedup is in $\mathcal{O}(m/\log m)$, where $m \in \mathbb{N}$ is the number of processors. However, this only holds if m is sublinear in the number of iterations. Thus, the amount of nodes cannot be increased enough to achieve polylogarithmic runtime.

Recently, averaging models was investigated empirically for training (deep) neural networks (McMahan et al., 2017) showing promising results. Studying dynamic averaging for this case poses particular challenges due to the non-convex nature of the training objective. In this case it has to be assumed that local models remain in a locally convex environment in order to proof optimal convergence. An empirical study indicates that by a careful initialization of the networks this assumption holds in practice. The approach is discussed in Chapter 4.

Effective Parallel Batch Learning: In order to parallelize generic algorithm for batch learning—including non-incremental ones—a novel aggregation operator is introduced that provides strong guarantees on the model quality. The idea is to combine models produced by learning algorithms on local datasets by calculating their Radon point.

The Radon point lies within the convex hull of the local models and is more robust to outliers than the average—at least two models have a worse performance than the Radon point, whereas for the average this can be guaranteed only for one. By iteratively replacing sets of models by their Radon point until a single point remains, the number of models with a performance worse than the remaining Radon point grows exponentially.

Given a probabilistic error guarantee for each local model, e.g., provided by the empirical risk minimization model, it can be shown that this aggregation reduces the error probability doubly exponentially in the number of iterations. For that, the protocol requires only a single aggregation, rendering it highly communication-efficient. This approach is denoted the Radon machine. In order to achieve the same probabilistic error guarantee as the serial application of the learning algorithm, it requires more training data. Thus, in theory the speedup is worse than that of dynamic averaging, i.e., it is in $\mathcal{O}(m^{\kappa/\log d})$, where $m \in \mathbb{N}$ is again the number of processing units, $\kappa \in \mathbb{N}$ depends on the sample and runtime complexity of the learning algorithm and $d \in \mathbb{N}$ is the dimension of the model space. However, since there is no restriction on the number of processing units, choosing m quasi-polynomially in the input size $N \in \mathbb{N}$ of the serial algorithm, i.e., $m \in \mathcal{O}(N^{\log d})$, achieves polylogarithmic runtime, i.e., in $\mathcal{O}\left(\log^{\kappa} N + d^3 \log N\right)$. Thus, the Radon machine achieves optimal model quality with minimal communication in polylogarithmic runtime on quasi-polynomially many processing units. It is the first generic approach to achieve all three requirements (R1, R2, and R3 in Section 1.1) at the same time. It is discussed in Chapter 5.

Additional Contributions: In addition to those main points, this thesis contains two additional contributions. The first contribution concerns the application of dynamic averaging with kernel methods. For that, the average of kernel models has to be defined. It is given by the union of the support vectors of all local models and an average of their respective weights. This aggravates the general problem of model sizes for kernel methods: even in the centralized case the number of support vectors is up to linear in the dataset size; this is a particular challenge when applying them to potentially infinite data streams. Averaging kernel models increases their size substantially, up to linear in the size of the union of all local datasets. As a solution, model compression techniques have been proposed for the centralized case that limit the number of support vectors, such as truncation (Kivinen et al., 2004) and projection (Orabona et al., 2009; Wang and Vucetic, 2010). It can be shown that the compression error of these techniques does not deteriorate the model quality for dynamic averaging.

The second contribution concerns the privacy of local datasets. If the data is privacy-sensitive, its centralization should be avoided. Black-box parallelizations only communicate a model trained on that data, effectively reducing the privacy-sensitive information that can be learned. However, if an attacker has access to two consecutive model updates, she can still infer information about the local data. In particular, if linear models are updated by stochastic gradient descent with hinge loss, an attacker can reconstruct the local data from the model update. Applying noise to local data before training (Balcan et al., 2012), or to local models (Chaudhuri et al., 2011), allows to achieve ϵ-differential privacy (Dwork et al., 2006) which bounds the loss in privacy through publication of the models.

1.3. Outline

The remainder of this thesis is structured as follows.

Chapter 2 starts with a formal description of machine learning as optimization, introducing both the empirical risk minimization model for batch learning and the online learning model. In contrast to standard exposition of batch and online learning, here, both are viewed from the perspective of optimization algorithms. That allows to analyze the parallelization of algorithms from both learning setups in a unified way. Subsequently, state-of-the-art parallelizations of such algorithms are discussed. The chapter concludes by introducing black-box parallelizations in a unified framework.

Chapter 3 presents a black-box parallelization for a broad class of incremental learning algorithms and analyzes it in the online learning model. It is shown that the approach is able to retain the predictive performance of the serial application of the learning algorithm, adapting the communication to the hardness of the learning problem.

Chapter 4 applies the previously presented approach to incremental learning algorithms for batch learning and analyzes it in the empirical risk minimization model. The analysis shows that the approach retains the convergence rate of the underlying learning algorithm, producing models with a predictive performance comparable to that obtained by the serial application of the learning algorithm. At the same time it achieves a substantial speedup with the number of employed processing units.

Chapter 5 presents a novel parallelization scheme for a broad class of incremental and non-incremental batch learning algorithms. The scheme maintains theoretical performance guarantees while reducing the runtime of many algorithms from polynomial to polylogarithmic on quasi-polynomially many processing units. This is a significant step towards a general answer to an open question on efficient parallelization of machine learning in the sense of Nick's class ($\mathcal{NC}$).

1.4. Previously Published Work

Parts of this dissertation have already been published in conference and workshop proceedings.

1. Michael Kamp, Mario Boley, Michael Mock, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Adaptive communication bounds for distributed online learning. In *7th NIPS Workshop on Optimization for Machine Learning (OPT)*, 2014.

2. Michael Kamp, Mario Boley, Thomas Gärtner. Beating Human Analysts in Nowcasting Corporate Earnings by using Publicly Available Stock Price and Correlation Features. In *Proceedings of the SIAM International Conference on Data Mining*, 2014.

3. Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, pages 623-639, 2014.

4. Michael Kamp, Mario Boley, and Thomas Gärtner. Parallelizing randomized convex optimization. In *8th NIPS Workshop on Optimization for Machine Learning (OPT)*, 2015.

5. Michael Kamp, Sebastian Bothe, Mario Boley, and Michael Mock. Communication-efficient distributed online learning with kernels. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, pages 805-819, 2016.

6. Michael Kamp, Mario Boley, Olana Missura, and Thomas Gärtner. Effective parallelisation for machine learning. In *Advances in Neural Information Processing Systems - NIPS*, pages 6459-6470, 2017.

7. Michael Kamp, Linara Adilova, Joachim Sicking, Fabian Hüger, Peter Schlicht, Tim Wirtz, Stefan Wrobel. Efficient Decentralized Deep Learning by Dynamic Model Averaging. *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, 2018.

2. Black-Box Parallel Machine Learning

This thesis focuses on machine learning as optimization. This means, a loss function is optimized over training data. This defines a broad family of machine learning algorithms, including batch learning algorithms such as least squares and LASSO regression, support vector machines, and many deep learning algorithms, as well as online learning algorithms, such as stochastic gradient descent and passive aggressive updates. The training data that is either available as a batch and the algorithm has random access to all elements of the training data, or as a stream of data instances, i.e., only the current data example—or a finite moving window of examples—is available to the learning algorithm. In both cases, the data either is inherently distributed, or is distributed in the course of parallelization.

Batch algorithms can be theoretically analyzed using the empirical risk minimization (ERM) model. This allows to provide theoretical guarantees on the generalization error, i.e., the error on unseen data from the same target distribution. However, this model requires that the training data is drawn independently and identically distributed (iid) from a fixed target distribution. For online algorithms, this assumption rarely holds. Instead, they can be analyzed in the online learning model which allows to provide worst-case guarantees on the cumulative error on arbitrary sequences of examples. The goal of this thesis is to investigate black-box parallelizations for which such guarantees can be provided, both for batch and online learning.

Classical parallelization strategies find parts in an algorithm that can be computed independently, and thus in parallel. The black-box approach instead applies the algorithm *as is* on subsets of the training data in parallel and combines the results of each instance of the algorithm. Thus for a black-box parallelization, two essential questions have to be answered: (i) how to combine the outputs of the instances of the algorithm, and (ii) when to combine them. The goal is to combine the results in such a way that the combination has a quality similar to the result of the algorithm applied serially to all data on a single processing unit. At the same time, the runtime of the parallelized algorithm and the amount of communication between the parallel instances of the algorithm should be minimal.

This chapter first introduces machine learning as optimization, followed by the empirical risk minimization and online learning model in Section 2.1. This is followed by a review on existing parallelizations of such algorithms in Section 2.2. The black-box parallelization approach is described in Section 2.3 which then proposes a formal framework that allows to describe and analyze such parallelization.

2.1. Machine Learning as Optimization

Machine learning aims at modeling an unknown dependence between instances and their labels. The instances are elements of an **input space** $\mathcal{X}$, labels are elements of an **output space** $\mathcal{Y}$. The unknown dependence is reflected in an unknown joint **target distribution** $\mathcal{D}\colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$ over the input and output space. A machine learning algorithm $\mathcal{A}$ aims at representing this dependence by a **model** chosen from a **model space** $\mathcal{F}$ of functions $f : \mathcal{X} \to \mathcal{Y}$. For that, the algorithm has access to a **training set** of size $N \in \mathbb{N}$, i.e., a finite sequence of **examples**

$$E = \{(x_1, y_1), \dots, (x_N, y_N)\} \subseteq \mathcal{X} \times \mathcal{Y} \ .$$

The deviation of a **prediction** $f(x)$ to the correct label y is measured by a **loss function** $l\colon \mathcal{F} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$.

Modeling the target distribution can be stated as an optimization problem by crafting an objective function over the model space for which the optimum corresponds to the best model with respect to the loss function. However, there is a difference between learning settings: in **batch learning** the algorithm can process large amounts of data before outputting a model and the goal is for that final model to perform well on new data. That is, the algorithm aims at finding a model that minimizes the **expected error**

$$\mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} l(f, x, y) \ ; \tag{2.1}$$

in contrast, in **online learning** the algorithm has to respond during the learning process and the goal is to minimize the loss suffered in the process. That is, the algorithm aims at minimizing the **cumulative loss**

$$L(T) = \sum_{t=1}^{T} \ell(f_t, x_t, y_t) \ . \tag{2.2}$$

In both cases, to solve such optimization problems efficiently it is typically required that the model space can be parametrized in a fixed, finite dimensional Euclidean space $\mathbb{R}^d$. The following gives examples of such model spaces.

2.1.1. Classes of Model Spaces

Model spaces can be divided into groups, where each group forms a **model class**, or hypothesis class (in machine learning, the terms model and hypothesis are often used interchangeably). Popular classes of model spaces used throughout this thesis are: (i) the class of **linear models**, (ii) the class of non-linear models from reproducing kernel Hilbert spaces in their support vector representation, in the following denoted **kernel models**, and (iii) the class of (deep) **neural networks**.

This division in classes allows to jointly address the common peculiarities of each model space. For example, linear models usually allow to efficiently solve the optimization objective, especially if the loss is convex, however their predictive power is limited. Kernel models allow for efficient optimization, as well, since they are also linear models from the corresponding reproducing kernel Hilbert space. Moreover, they can have a much higher predictive power. However, the size of their representation can be as large as the training set which can be problematic for large data sets or potentially infinite data streams. For both linear and kernel models, it is often possible to find an (approximation to an) optimal model efficiently. Neural networks have a fixed representation size and high predictive power, but the optimization objective is often non-convex, complicating the optimization. We discuss these and other peculiarities and their consequences for parallelizations in the following chapters. In the following, these model classes are formally defined.

Linear Models

The class of linear models consists of linear functions from the input to the output space. If the input space is a Euclidean vector space of dimension $d \in \mathbb{N}$ and the output space is $\mathbb{R}$, the respective model space can be represented as a subset of the same Euclidean vector space. The function $f \in \mathcal{F}$ is given implicitly by a vector $w \in \mathbb{R}^d$ with $f_w(x) = \langle w, x \rangle$ for $x \in \mathcal{X}$. Here, $\langle \cdot, \cdot \rangle$ denotes the standard inner product in $\mathbb{R}^d$. Thus, the model space is defined as

$$\mathcal{F} = \{f_w | w \in \mathbb{R}^d\} \ .$$

With a slight abuse of notation, the function f_w is identified with its representation: We abbreviate $f_w \in \mathcal{F}$ with $w \in \mathbb{R}^d$ as $f \in \mathcal{F} \subseteq \mathbb{R}^d$, where f denotes the linear function as well as its representation as a vector in $\mathbb{R}^d$. If the output space is instead binary, i.e., $\mathcal{Y} = \{-1, 1\}$, then the function can be represented by $f(x) = \operatorname{sign} \langle w, x \rangle$. In order to represent functions $f(x) = \langle w, x \rangle + b$ for some intercept $b \in \mathbb{R}$, a standard trick in machine learning is to extend the input space by one dimension which is always set to 1, i.e., for $\mathcal{X} \subseteq \mathbb{R}^d$ we define

$$\mathcal{X}' = \left\{x' = \begin{pmatrix} x \\ 1 \end{pmatrix} \in \mathbb{R}^{d+1} | x \in \mathcal{X}\right\} \ .$$

The corresponding linear model space is given by

$$\mathcal{F}' = \left\{f_{w'} : w' = \begin{pmatrix} w \\ b \end{pmatrix} \in \mathbb{R}^{d+1} | f_w \in \mathcal{F}, w \in \mathbb{R}^d, b \in \mathbb{R}\right\} \ ,$$

and a model $f_{w'} \in \mathcal{F}'$ is given by

$$f_{w'}(x) = \left\langle w', x' \right\rangle = \langle w, x \rangle + b \ .$$

Kernel Models

The class of kernel models consists of functions from a **reproducing kernel Hilbert space**

$$\mathcal{H}_k = \left\{f : \mathcal{X} \to \mathbb{R} | f(\cdot) = \sum_{j=1}^{\dim F} w_j \phi_j(\cdot)\right\}$$

with positive definite **kernel function** $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, **feature space** F, and a mapping $\phi\colon \mathcal{X} \to F$ into the feature space (Scholkopf and Smola, 2001). The kernel function corresponds to an inner product of input points mapped into the feature space, i.e.,

$$k(x, x') = \sum_{j=1}^{\dim F} \xi_j \phi_j(x) \phi_j(x')$$

for constants $\xi_1, \xi_2, \dots \in \mathbb{R}$. Thus, we can express the model in its **support vector expansion**, or dual representation

$$f(\cdot) = \sum_{x \in S} \alpha_x k(x, \cdot)$$

with a set of **support vectors**

$$S = \{x_1, x_2, \dots\} \subseteq \mathcal{X}$$

and corresponding **coefficients** $\alpha_x \in \mathbb{R}$ for each $x \in S$. This implies that f is a linear function over the feature space F with weights $w = (w_1, w_2, \dots) \in F$ defining f given by $w_i = \sum_{x \in S} \xi_i \alpha_x \phi_i(x)$. If the model is a solution to a regularized risk minimization problem for a given training set $(x_1, y_1), \dots, (x_N, y_N) \subset \mathcal{X} \times \mathcal{Y}$ (where $\mathcal{Y} \subseteq \mathbb{R}$) of size $N \in \mathbb{N}$, then the following representer theorem guarantees that the model can be represented by N support vectors from the training set.

Theorem 2.3 (Nonparametric Representer Theorem (Schölkopf et al., 2001; Wahba, 1990)). *Given a non-empty set $\mathcal{X}$, a set $\mathcal{Y} \subseteq \mathbb{R}$, a positive definite real-valued kernel k on $\mathcal{X} \times \mathcal{X}$, a training set*

$$(x_1, y_1), \dots, (x_N, y_N) \subset \mathcal{X} \times \mathcal{Y} \ ,$$

a strictly monotonically increasing real-valued function Ω on $[0, \infty[$, an arbitrary loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R} \cup \{\infty\}$, and a class of functions

$$\mathcal{F} = \left\{ f : \mathcal{X} \to \mathbb{R} \,\middle|\, f(\cdot) = \sum_{x \in \mathcal{X}} \alpha_x k(\cdot, x), x \in \mathcal{X}, \alpha_x \in \mathbb{R}, \|f\| < \infty \right\} \ .$$

Here, $\|\cdot\|$ denotes the norm in $\mathcal{H}_k$ associated with k, i.e., for any $x \in \mathcal{X}, \alpha_x \in \mathbb{R}$,

$$\left\| \sum_{x \in \mathcal{X}} \alpha_x k(\cdot, x) \right\| = \sum_{x \in \mathcal{X}} \sum_{x' \in \mathcal{X}} \alpha_x \alpha_{x'} k(x, x') \ .$$

Then any $f \in \mathcal{F}$ minimizing the regularized risk functional (see Section 2.1.3)

$$\operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{N} \ell(f, x_i, y_i) + \Omega(f) \ .$$

admits a representation of the form

$$f(\cdot) = \sum_{i=1}^{N} \alpha_{x_i} k(x_i, \cdot) \ .$$

The theorem guarantees that the optimal model lies on the span of the kernel functions on the training examples, i.e.,

$$f(\cdot) \in \operatorname{span}\{k(x_1,\cdot),\dots,k(x_N,\cdot)\}$$

for the training set $(x_1,y_1),\dots,(x_N,y_N)$. This implies that, even if $\mathcal{H}$ is infinite-dimensional, f is from an N-dimensional subspace of $\mathcal{H}_k$ and thus can be efficiently computed.

Neural Networks

The class of neural networks for $\mathcal{Y} \subseteq \mathbb{R}$ can be represented by a weighted composition of a set of $k \in \mathbb{N}$ functions $g_i \colon \mathbb{R} \to \mathbb{R}$[1], i.e.,

$$f(x) = a\left(\sum_{i=1}^{k} w_i g_i(x)\right) \quad ,$$

where $a \colon \mathbb{R} \to \mathbb{R}$ is referred to as **activation function** and $w_1,\dots,w_k$ are weights. The g_i are referred to as **neurons**. If the neuron g_i is a weighted sum of the input $x \in \mathcal{X}$, it is referred to as **input neuron**. If instead it is defined as a weighted composition of functions itself, i.e.,

$$g_i(x) = a\left(\sum_{j\neq i} w_{ij} g_j(x)\right) \quad ,$$

it is referred to as **hidden neuron**. The neuron associated with f is denoted **output neuron**. If $\mathcal{Y} = \{-1,1\}$, the neural network can again be represented as the sign of the activation function.

The neural networks can also be represented as a directed graph $\mathcal{G} = (V,E)$, where each neuron g_i is represented by a vertex $v_i \in V$. Each edge $e = (v_i, v_j, w_{ij}) \in E$ is a weighted connection between the neurons g_i and g_j with weight $w_{ij} \in \mathbb{R}$. If the neural network can be represented by an acyclic graph, is is called a **feed forward network**, if not, it is called a **recurrent neural network**.

Often, neurons are organized in layers, i.e., the set of neurons is partitioned into subsets $\mathbf{g}_1,\dots,\mathbf{g}_l$, each subset being referred to as a **layer**. For these layers, the following constraint is imposed on the weights: For two neurons, one from layer $\mathbf{g}_i$ and one from layer $\mathbf{g}_j$ it holds that $w_{ij} = 0$ if $j \neq i+1$, i.e., only neurons in consecutive layers are connected[2]. Neural networks with a large number of layers are referred to as **deep neural networks**, machine learning algorithms for training such networks are summarized under the term **deep learning**.

This thesis presents black-box parallelizations for a wide range of learning algorithms using these three classes of model spaces. The performance of a model from such a model space is measured by a loss function. The following subsection gives a few examples of popular ones.

[1] We assume neurons to be representable by functions to simplify notation. This does not take into account techniques like dropout (Srivastava et al., 2014), where the output of a neuron is probabilistic.

[2] This definition of layers only holds for feed-forward networks. In practice, recurrent neural networks are also visualized in layers to better highlight their structure, even though there can be arbitrary connections between layers.

2.1.2. Examples of Loss Functions

In case of a classification problem where $\mathcal{Y} = \{-1, 1\}$, a common loss function is the 0-1-**loss**, i.e.,

$$l(f, x, y) = \begin{cases} 0, & \text{if } f(x) = y \\ 1, & \text{otherwise} \end{cases} \quad .$$

Since this function is neither continuous nor convex, optimizing over it is difficult. Thus, it is often replaced by a convex upper bound, e.g., the **logistic loss**

$$\ell_{\text{log}}(f, x, y) = \log\left(1 + e^{-yf(x)}\right) \quad ,$$

or the **hinge loss**

$$\ell_{\text{hinge}}(f, x, y) = \max\{0, 1 - yf(x)\} \quad .$$

For regression problems where $\mathcal{Y} = \mathbb{R}$, the deviation of prediction and label can be measured by the **absolute loss**

$$l(f, x, y) = |f(x) - y|$$

which is convex, but unsuitable for gradient-based solvers (the gradient is the same for large and small losses, so that gradient-based solvers require many steps to reach the minimum). More common loss functions are the **squared loss**

$$\ell_{\text{sq}}(f, x, y) = (f(x) - y)^2$$

and the ϵ**-insensitive loss**

$$\ell_{\text{eps}}(f, x, y) = \max\{0, |f(x) - y| - \epsilon\} \quad .$$

The goal in both batch and online learning is to optimize this loss. In the following, we discuss the case of batch learning, followed by online learning.

2.1.3. Empirical Risk Minimization

In batch learning, the algorithm can optimize an objective over large amounts of data. The goal is to find a model that minimizes the expected error (Equation 2.1). However, this cannot be computed as the distribution $\mathcal{D}$ is unknown. To obtain efficient machine learning algorithms, a common strategy is to craft an objective function that is derived from a high probability upper bound on the expected error. In the following, this approach is detailed.

In order to craft an objective function whose optimum is close to the minimizer of the expected error that can also be efficiently optimized, the expected error is replaced by its empirical counterpart computed from a sample **training set** of size $N \in \mathbb{N}$:

$$E = \{(x_1, y_1), \ldots, (x_N, y_N)\} \subseteq \mathcal{X} \times \mathcal{Y}$$

that is drawn independent and identically distributed (iid) from $\mathcal{D}$. Moreover, the loss is replaced by a convex upper bound $\ell : \mathcal{F} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ (see the previous Section 2.1.2), and complex models are penalized by a convex regularization term $\Omega : \mathcal{F} \to \mathbb{R}$.

Machine learning algorithms are thus often defined by a regularized **empirical risk minimization** problem

$$\operatorname*{argmin}_{f\in\mathcal{F}} \sum_{i=1}^{N} \ell(f, x_i, y_i) + \Omega(f) \ . \tag{2.4}$$

where the objective function is denoted the **empirical risk**[3]

$$\mathcal{L}_{emp}(f) = \sum_{i=1}^{N} \ell(f, x_i, y_i) + \Omega(f) \ . \tag{2.5}$$

The expected error with respect to the loss function employed in the empirical risk minimization problem is called the **risk**

$$\mathcal{L}_{\mathcal{D}}(f) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \ell(f, x, y) \ .$$

The ERM model allows to provide strong guarantees on models that minimize Equation 2.4.

Generalization Bounds for Empirical Risk Minimization

Given a model that minimizes Equation 2.4, a **generalization bound** is a probabilistic guarantee on its risk. That is, for a given $\epsilon > 0$ and $\delta \in (0, 1]$, with probability $1 - \delta$ it holds for a training set $E \subset \mathcal{X} \times \mathcal{Y}$ drawn iid from $\mathcal{D}$ and the model $f = \mathcal{A}(E)$ that

$$\mathcal{L}_{\mathcal{D}}(f) \leq \min_{f'\in\mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') + \epsilon \ .$$

Here, the first component, $\min_{f'\in\mathcal{F}} \mathcal{L}_{\mathcal{D}}(f')$, denotes the error of the best model from the model space which is often termed **approximation error** or **bias** of the algorithm. The second component ϵ is the error due to a lack of training data, denoted **estimation error**. An equivalent form of guarantees is to bound the difference between risk and approximation error, denoted the **regret**

$$\mathcal{R}(f) = \mathcal{L}_{\mathcal{D}}(f) - \min_{f'\in\mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') \ .$$

For most learning algorithms, the generalization bound improves monotonically with the size of E. Such learning algorithms are **consistent** if there is a function $N_{\mathcal{F}}\colon \mathbb{R}_+ \times (0, 1] \to \mathbb{R}_+$ such that for all $\epsilon > 0$, $\delta \in (0, 1]$, $N \geq N_{\mathcal{F}}(\epsilon, \delta)$, and a training set $E \sim \mathcal{D}^N$ of size N, the probability of $\mathcal{A}$ outputting a model with regret larger than ϵ is smaller than δ, i.e.,

$$P(\mathcal{R}(\mathcal{A}(E)) > \epsilon) \leq \delta \ , \tag{2.6}$$

[3] Note that in the literature, empirical risk often refers only to the loss, not the regularization term. In this case, the objective with regularization is denoted structural risk (Shalev-Shwartz and Ben-David, 2014) or regularized risk (Von Luxburg and Schölkopf, 2009).

The function $N_{\mathcal{F}}\colon \mathbb{R}_+ \times (0,1] \to \mathbb{R}_+$ is called the **sample complexity** of $\mathcal{A}$[4]. For an empirical risk minimization algorithm $\mathcal{A}$ with 0-1-loss function and a model space $\mathcal{F}\colon \mathcal{X} \to \{-1,1\}$ with finite Vapnik-Chervonenkis dimension $VC_{\mathbf{dim}}(\mathcal{F}) \in \mathbb{N}$ and finite Rademacher complexity, the sample complexity is (Shalev-Shwartz and Ben-David, 2014)[5]

$$N_{\mathcal{F}}(\epsilon,\delta) \geq C\frac{VC_{\mathbf{dim}}(\mathcal{F}) + \ln\left(\frac{1}{\delta}\right)}{\epsilon^2} \quad .$$

for a constant $C \in \mathbb{R}_+$.

Both, $\mathcal{VC}$-dimension and Rademacher complexity are measures for the capacity of a model space. The $\mathcal{VC}$-dimension of a model space $\mathcal{F}: \mathcal{X} \to \{-1,1\}$ is defined (Von Luxburg and Schölkopf, 2009) as

$$VC_{\mathbf{dim}}(\mathcal{F}) = \max\{n \in \mathbb{N} | \exists Z_n \subset \mathcal{X}, |Z_n| = n \text{ s.t. } |\mathcal{F}_{Z_n}| = 2^n\} \quad .$$

Here $|\mathcal{F}_{Z_n}|$ denotes the number of labellings of the instances in Z_n that can be realized by $\mathcal{F}$, i.e., for $Z_n = \{x_1,\dots,x_n\}$

$$\mathcal{F}_{Z_n} = \{y_1,\dots,y_n \in \{-1,1\} | \exists f_t \in \mathcal{F} : f(x_i) = y_i \text{ for all } i \in [n]\} \quad .$$

The Rademacher complexity of a model space $\mathcal{F}\colon \mathcal{X} \to \{-1,1\}$ is defined as

$$\text{Rad}_N(\mathcal{F}) = \mathbb{E}\sup_{f\in\mathcal{F}} \frac{1}{N}\sum_{i=1}^{N} \sigma_i f(X_i) \quad ,$$

where X_i are random variables drawn iid from $\mathcal{X}$ and σ_i are independent binary random variables drawn from the Rademacher distribution, i.e., $P(\sigma_i = 1) = P(\sigma_i = -1) = 1/2$. The expectation is both over the X_i and σ_i. If a consistent learning algorithm exists for a model space $\mathcal{F}$, then it is **agnostic PAC learnable**.

Definition 2.7 (Agnostic PAC Learnability (Shalev-Shwartz and Ben-David, 2014))**.** *A model space $\mathcal{F}$ is agnostic PAC learnable if there exist a function $N_{\mathcal{F}}\colon (0,1) \times (0,1) \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following property: For every $\epsilon,\delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, when running the learning algorithm on $N \geq N_{\mathcal{F}}(\epsilon,\delta)$ many examples drawn iid from $\mathcal{D}$, $\mathcal{A}$ returns a model $f \in \mathcal{F}$ such that, with probability of at least $1-\delta$ (over the choice of the N training examples), its risk is bounded by*

$$\mathcal{L}_{\mathcal{D}}(f) \leq \min_{f'\in\mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') + \epsilon \quad .$$

These guarantees hold for all instances of the ERM objective. For batch learning, this thesis considers agnostic PAC learnable problems. A particular learning algorithm for such a problem is defined by the optimization objective (Equation 2.5), i.e., the model space, loss function, and regularization term. In the following, a few examples are given.

[4] In the literature, the sample complexity is defined with respect to the model space $\mathcal{F}$, assuming that $\mathcal{A}$ outputs the empirical risk minimizer. Here, the algorithm and model space are

[5] Shalev-Shwartz and Ben-David (2014) show in Chapter 28 that any empirical risk minimization algorithm using a model space $\mathcal{F}$ with finite $\mathcal{VC}$-dimension $d \in \mathbb{N}$ achieves an (ϵ,δ)-guarantee (Equation 2.6) using a dataset of size $N \geq {}^{128d}/_{\epsilon^2}\log({}^{64d}/_{\epsilon^2}) + {}^{8}/_{\epsilon^2}(8d\log({}^{e}/_{d}) + 2\log({}^{4}/_{\delta}))$.

Examples of Empirical Risk Minimization Algorithms

As an example, assume a binary classification problem, i.e., $\mathcal{Y} = \{1, -1\}$, over a d-dimensional vector space, i.e., $\mathcal{X} \subseteq \mathbb{R}^d$. Furthermore, assume a linear model space

$$\mathcal{F} = \left\{ f_w : \mathbb{R}^d \to \{1, -1\} | w \in \mathbb{R}^d, f_w(x) = \operatorname{sign}(\langle w, x \rangle) \right\} \quad ,$$

and the logistic loss

$$\ell_{\log}(f_w, x, y) = \log\left(1 + e^{-y\langle w, x\rangle}\right) \quad .$$

The algorithm solving the corresponding empirical risk minimization problem

$$\operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{N} \ell_{\log}(f_w, x, y) = \operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{N} \log\left(1 + e^{-y\langle w, x\rangle}\right)$$

is known as **logistic regression** which finds the maximum likelihood estimator for the classification problem.

For such a classification problem, kernel models

$$\mathcal{F} = \mathcal{H}_k = \left\{ f : \mathcal{X} \to \mathbb{R} | f(\cdot) = \sum_{j=1}^{\dim F} w_j \phi_j(\cdot) \right\}$$

with kernel function k can be used. Together with the hinge loss

$$\ell_{\text{hinge}}(f, x, y) = \max\{0, 1 - yf(x)\} \quad .$$

and $L2$-regularization, the corresponding ERM problem is

$$\operatorname*{argmin}_{f \in \mathcal{H}_k} \sum_{i=1}^{N} \ell_{\text{hinge}}(f_w, x, y) + \|f\|_{\mathcal{H}_k}^2 \quad .$$

The algorithm solving this problem is known as **support vector machine** (SVM).

As another example, assume a regression problem, i.e., $\mathcal{Y} = \subseteq \mathbb{R}$ over $\mathcal{X} \subseteq \mathbb{R}^d$. For a linear model space the loss can be measured by the squared loss $\ell_{\text{sq}}(f_w, x, y) = (f_w(x) - y)^2$. The corresponding algorithm solving

$$\operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{N} (f_w(x) - y)^2$$

is **least-squares regression**. Adding an L_2-regularization $\Omega(f_w) = \|w\|_2$, i.e.,

$$\operatorname*{argmin}_{f \in \mathcal{F}} \sum_{i=1}^{N} (f_w(x) - y)^2 + \|w\|_2$$

yields the **ridge regression** algorithm. Instead using L_1-regularization $\Omega(f_w) = \|w\|_1$ is known as **LASSO regression**. The support vector machine described above can also be applied to regression problems by using the ϵ-insensitive loss

$$\ell_{\text{eps}}(f, x, y) = \max\{0, |f(x) - y| - \epsilon\} \quad .$$

instead of the hinge loss, yielding **support vector regression** (SVR). These objective functions can be optimized using standard (convex) optimization algorithms.

Examples of Optimization Algorithms

Since the model space allows a representation in the Euclidean space, the empirical risk can be regarded as a function $\mathcal{L}_{emp}\colon \mathbb{R}^d \to \mathbb{R}$ for some $d \in \mathbb{N}$. If the model space $\mathcal{F}$ is convex, i.e., for every $f, f' \in \mathcal{F}$ and any $\alpha \in [0,1]$ it holds that $\alpha f + (1-\alpha)f' \in \mathcal{F}$, and the loss function $\mathcal{L}_{emp}$ is differentiable and convex in f, i.e.,

$$\mathcal{L}_{emp}(\alpha f + (1-\alpha)f') \leq \alpha\mathcal{L}_{emp}(f) + (1-\alpha)\mathcal{L}_{emp}(f') \ ,$$

then this instance of ERM is denoted a **convex learning problem** (Shalev-Shwartz and Ben-David, 2014) that can be solved by a convex optimization algorithm. An example are first-order optimization algorithms that incrementally update the solution based on the gradient of the objective function: The **gradient descent** (GD) algorithm in each round $t \in \mathbb{N}$ takes a step in the direction of the negative gradient at the current model. That is, the update rule is

$$f_{t+1} = f_t - \eta\nabla\mathcal{L}_{emp}(f_t) = f_t - \eta\nabla\left(\sum_{i=1}^{N}\ell\left(f, x_i, y_i\right) + \Omega(f)\right) \ ,$$

where $\nabla\mathcal{L}_{emp}(f_t)$ denotes the gradient of $\mathcal{L}_{emp}$ and $\eta \in \mathbb{R}_+$ is the learning rate. For each update, GD requires to calculate the gradient with respect to the entire training set. In **stochastic gradient descent** (SGD) (Robbins and Monro, 1951), instead only the gradient at a single data point is used which in expectation is in the direction of the gradient over all data. That is, in each round SGD picks an example (x_t, y_t) from the training set and updates

$$f_{t+1} = f_t - \eta\nabla\left(\ell\left(t, x_t, y_t\right) + \Omega(f_t)\right) \ .$$

A middle-ground between GD and SGD is the **mini-batch SGD** algorithm (Dekel et al., 2012). This algorithms draws $B \in \mathbb{N}$ examples from the training set and uses the gradient of this mini-batch in each update, i.e.,

$$f_{t+1} = f_t - \eta\nabla\left(\sum_{i=1}^{B}\ell\left(t, x_i, y_i\right) + \Omega(f_t)\right) \ .$$

These approaches can be generalized to non-differentiable objectives by using a subgradient, instead of the gradient (see, e.g., Shalev-Shwartz and Ben-David (2014), Chapter 14.2).

Another family of optimization algorithms are quasi-Newton methods which not only use the gradient, but also (an approximation to) the Hessian of the objective. An example is the limited-memory Broyden-Fletcher-Goldfarb-Shanno (**l-BFGS**) algorithm (Broyden, 1970) that performs updates in the direction

$$p_t = -B_t^{-1}\nabla\mathcal{L}_{emp}(f_t) \ ,$$

where B_t is the current approximation of the Hessian.

2.1.4. Incremental and Non-Incremental Learning Algorithms

Empirical risk minimization (ERM) algorithms process a dataset and output a model. If the dataset is altered, e.g., by adding new examples, many ERM algorithms are not able to update the model, but have to process the entire dataset again (e.g., because they require random access to the entire dataset). These algorithms are thus **non-incremental**. Examples for non-incremental learning algorithms are support vector machines (Scholkopf and Smola, 2001), Gaussian processes (Rasmussen, 2004), and decision trees (Quinlan, 1986).

If instead the algorithm can be represented as a sequence of model updates, it is denoted **incremental**. That is, given a training set E, the algorithm starts with a model $f_0 \in \mathcal{F}$ and in each round $t \in \mathbb{N}$ draws a subset E_t of E and the current model f_t is updated to $f_{t+1} = \mathcal{A}(E_t, f_t)$. The sequence f_t then converges to the minimizer f^* of the ERM problem. The optimization algorithms presented above can be regarded as examples of incremental learning algorithms.

Let $\mathcal{L}_{emp}$ denote the empirical risk (see Equation 2.5). The error $\widehat{\epsilon} = \|\mathcal{L}_{emp}(f_T) - \mathcal{L}_{emp}(f^*)\|$ after $T \in \mathbb{N}$ rounds of the incremental learning algorithm is denoted the **optimization error**[6]. The rate at which the optimization error decreases with the number of rounds is called the **convergence rate**. It is desirable to have a fast convergence rate so that the incremental algorithm reaches the ERM solution with as few passes over the entire dataset, denoted **epochs**, as possible.

As an example, SGD can be viewed as an incremental learning algorithm $\mathcal{A}^{\text{SGD}}$ for a given ERM objective (see Chapter 14 in Shalev-Shwartz and Ben-David (2014)), i.e.,

$$\mathcal{A}^{\text{SGD}}((x_t, y_t), f_t) = f_t - \eta\nabla\left(\ell\left(t, x_t, y_t\right) + \Omega(f_t)\right)$$

It has a convergence rate of $\widehat{\epsilon} \in \mathcal{O}(1/\sqrt{T})$ for convex loss functions with bounded gradient and bounded models.

If instead of a fixed training set E the algorithm $\mathcal{A}$ processes a stream of datasets $E_t \subset \mathcal{X} \times \mathcal{Y}$ for rounds $t \in \mathbb{N}$ drawn iid according to $\mathcal{D}$, the setting is referred to as **in-stream learning**. Since the stream of datasets is potentially infinite, a common assumption on $\mathcal{A}$ is that its runtime and memory for processing E_t is in $\mathcal{O}(|E_t|)$. As long as the data is drawn iid from a fixed target distribution, a generalization bound can be given also for in-stream learning: Since the learner observed

$$N = \sum_{t=1}^{T} |E_t|$$

many examples in round T, for a given $\delta \in (0, 1]$ the estimation error ϵ is given by solving $N = N_{\mathcal{F}}(\epsilon, \delta)$ for ϵ. Let $\widehat{\epsilon}$ be the optimization error in round T, then the risk of the model f_T can be bounded by

$$\mathcal{L}_{\mathcal{D}}(f_T) \leq \min_{f' \in \mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') + \epsilon + \widehat{\epsilon}\ .$$

Note that the stochastic gradient descent algorithm discussed above has the particular property that in a streaming setting, it can optimize the risk directly. If in each round $t \in \mathbb{N}$ the example (x_t, y_t) is freshly drawn iid from the target distribution, then by the linearity of the gradient

[6] Optimization error sometimes also refers to the difference between the risk and the expected error (Shalev-Shwartz and Ben-David, 2014). This error is a result of the inability to minimize the empirical risk with respect to the original loss function. Its size depends on the specific loss functions and target distribution.

it holds that

$$\mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \nabla_{f_t}\ell(f_t,x_t,y_t) = \nabla_{f_t} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \ell(f_t,x_t,y_t) = \nabla_{f_t}\mathcal{L}_{\mathcal{D}}(f_t) \ .$$

The gradient of the loss function is therefore an unbiased estimate of the gradient of the risk.

For batch learning, both incremental and non-incremental algorithms can be used. Note that in-stream learning can also be regarded a batch learning case and analyzed in the ERM model, if the model is not continuously evaluated and only its performance after processing a number of examples is of interest. However, incremental algorithms can also be used when the model is evaluated in each round and the goal is to optimize the performance over all rounds. Such a scenario can be analyzed in the online learning model. This model is introduced in the following section.

2.1.5. Online Learning

In **online learning** it is assumed that in each round $t \in \mathbb{N}$ the learner observers an instance $x_t \in \mathcal{X}$ and makes a prediction $\widehat{y}_t \in \mathcal{Y}$ using its current model f_t. After that, it observes the correct label $y_t \in \mathcal{Y}$. An online learning algorithm $\mathcal{A}$ then updates the current model

$$f_{t+1} = \mathcal{A}\left(f_t, x_t, y_t\right)$$

with the goal to make as few mistakes as possible during this process.

In principle, any incremental learning (and optimization) algorithm can be applied in this setting. However, generalization bounds for in-stream learning obtained by the ERM model only hold if data is drawn iid from a fixed target distribution. In case the data distribution varies over time or data is not drawn iid, these bounds do not hold. Moreover, they only bound the expected error after a certain training time, not taking into account the loss suffered during the training.

In the online learning model, no assumption is made on the target distribution. The sequence of examples can be deterministic, drawn from a fixed, or time-variant target distribution, or generated by an adversary. In contrast to the PAC case, here the in-place performance of the algorithm is of interest which can be measured by the cumulative loss (Equation 2.2)

$$L(T) = \sum_{t=1}^{T} \ell(f_t, x_t, y_t) \ .$$

While it is possible to provide probabilistic guarantees on the risk of online learning algorithms (Cesa-Bianchi and Lugosi, 2006), it is more common to give a worst-case bound on the cumulative loss, denoted **loss bound** $\mathbf{L}_{\mathcal{A}}(T)$. That is, for all sequences $(x_1, y_1), \dots, (x_T, y_T)$ it holds that

$$L_{\mathcal{A}}(T) = \sum_{t=1}^{T} \ell(f_t(x_t), y_t) \leq \mathbf{L}_{\mathcal{A}}(T) \ .$$

For binary classification, one can analyze the cumulative 0-1-loss of an online learning algorithm, i.e., the number of mistakes it makes. In the realizable case (Shalev-Shwartz and Ben-David, 2014), i.e., for some arbitrary instance $x \in \mathcal{X}$, the label is generated by an element

$f^* \in \mathcal{F}$ of the model space, the number of mistakes can be bounded in the Littlestone's Dimension $Ldim$ of the hypothesis class. Ldim is a measure similar to the VC-dimension and is defined as follows. A shattered tree of depth T is a binary tree of depth T, i.e., it has $2^{T+1} - 1$ nodes, where each node is associated with an instance $v_i \in \{v_1, \ldots, v_{2^{T+1}-1}\}$. A sequence of examples is generated by setting $x_1 = v_1$, and in round $t \in [T]$, $x_t = v_{i_t}$ where i_t is the current node. After round t, the next node is the left child of i_t if $y_t = -1$ and the right one if $y_t = 1$. That is, $i_{t+1} = 2i_t + y_1 t$, or without the recursion

$$i_t = 2^{t-1} + \sum_{j=1}^{t-1} y_j 2^{t-1-j} \quad .$$

A **$\mathcal{F}$-shattered tree** is a shattered tree such that for every labeling $y_1, \ldots, y_T \in \{-1, 1\}$ there exists a $f \in \mathcal{F}$ such that for all $t \in [T]$ it holds that $f(v_{i_t}) = y_t$ where $i_t = 2^{t-1} + \sum_{j=1}^{t-1} y_j 2^{t-1-j}$.

Definition 2.8 (Littlestone's Dimension $Ldim$ (Shalev-Shwartz and Ben-David, 2014))**.** *$Ldim(\mathcal{F})$ is the maximal $T \in \mathbb{N}$ such that there exists an $\mathcal{F}$-shattered tree.*

The **standard optimal algorithm** (SOA) for a model space $\mathcal{F}$ sets $V_1 = \mathcal{F}$ and in each round t receives an instance x_t, partitions V_t into two sets $V_t^r = \{f \in V_t : f(x_t) = r\}$ for $r \in \{-1, 1\}$ and predicts $\widehat{y}_t = \arg\max_{r \in \{-1,1\}} Ldim(V_t^r)$. After receiving the true label y_t it updates $V_{t+1} = \{f \in V_t : f(x_t) = y_t\}$. That is, in each round it predicts according to the class with larger $Ldim$ and discards all models that make a mistake. For this algorithm, it can be shown that the number of mistakes is bounded by $Ldim(\mathcal{F})$ and no other algorithm can have a smaller mistake bound (see Corollary 21.8 in Shalev-Shwartz and Ben-David (2014)). In the unrealizable case, i.e., when the labels are not generated by a member of the model space, no mistake bound sublinear in the number of rounds can be given (an adversary may chose to present always the opposite label to what the learner predicted). Instead, the performance is measured with respect to a reference model. This is captured by the **online regret**

$$R_{\mathcal{A}}(T) = \sum_{t=1}^{T} \ell(f_t, x_t, y_t) - \ell(f^*, x_t, y_t) \quad ,$$

where f^* is the reference model. Note that typically, the optimal model in hindsight

$$f^* = \arg\min_{f \in \mathcal{F}} \sum_{t=1}^{T} \ell(f, x_t, y_t) \quad .$$

is used as reference model.

Both online regret and the regret in the ERM model measure the excess loss over the best model from the model space. The difference is only that the regret in the ERM setting considers the expected loss on an unseen training example, whereas the online regret considers the cumulative regret for a given sequence of examples. In online learning, guarantees are often given as worst-case bounds on this regret.

Online Regret Bounds

A **regret bound** $\mathbf{R}_{\mathcal{A}}(T)$ is a worst-case guarantee on the regret that holds for all possible sequences of examples, i.e.,

$$R_{\mathcal{A}}(T) \leq \mathbf{R}_{\mathcal{A}}(T) = \sup_{(x_1,y_1),\dots,(x_T,y_T)} \left[\sum_{t=1}^{T} \ell(f_t,x_t,y_t) - \ell(f^*,x_t,y_t)\right]$$

Since the regret bound is a worst-case upper bound, it holds for all reference models, including the best model in hindsight.

The definition of regret can be generalized to time variant target distributions. For that, given a sequence of reference models $U = u_1,\dots,u_T$ the **shifting regret** (Herbster and Warmuth, 2001) with respect to this sequence of reference models is defined as

$$R_{\mathcal{A}}(T,U) = \sum_{t=1}^{T} \ell(f_t,x_t,y_t) - \ell(u_t,x_t,y_t) \ .$$

A **shifting regret bound** is a worst-case upper bound on the shifting regret for all sequences of reference models, including the optimal sequence of models. Shifting regret bounds are typically given in the total shift of the reference sequence

$$\sum_{t=2}^{T} \|u_t - u_{t-1}\|_2^2 \ .$$

Without further assumptions, no regret bound can be given that is better than linear in T (Cover, 1965). However, if we assume the model space to be convex and the loss function to be a convex function in the model, then sub-linear regret bounds can be achieved. In the following, a few example algorithms are introduced which have sub-linear regret bounds.

Examples of Online Learning Algorithms

The first example is SGD which can be applied in the online setting as well. Let ℓ be a convex loss function. In round $t \in \mathbb{N}$, the algorithm observes $x_t \in \mathcal{X}$, makes a prediction $\widehat{y}_t = f_t(x_t)$, then observes y_t and suffers loss $\ell(f_t,x_t,y_t)$. As before, SGD with learning rate $\eta \in \mathbb{R}_+$ then updates the model to

$$f_{t+1} = f_t - \eta\nabla_{f_t}\ell(f_t,x_t,y_t) \ .$$

The regret bound for SGD is (Shalev-Shwartz and Ben-David, 2014)

$$\mathbf{R}_{\mathcal{A}^{\text{SGD}}}(T) = \frac{\|f^*\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T} \|\nabla_{f_t}\ell(f_t,x_t,y_t)\|^2 \ .$$

A common assumption is that $\mathcal{F}$ is bounded, i.e., for all $f \in \mathcal{F}$ it holds that $\|f\| \leq \rho$ for a **data radius** $\rho \in \mathbb{R}_+$. In order to relate the distance between models to the difference in their losses, another common assumption is that ℓ is ι-Lipschitz, i.e., for all $f,f' \in \mathcal{F}$ and all $x \in \mathcal{X}, y \in \mathcal{Y}$

$$\left|\ell(f,x,y) - \ell\left(f',x,y\right)\right| \leq \iota \left\|f - f'\right\| \ .$$

Using these two assumptions and setting $\eta = \rho / L\sqrt{T}$ results in a regret bound of

$$\mathbf{R}_{\mathcal{A}^{\mathrm{SGD}}}(T) = \frac{\rho L}{2}\sqrt{T} \in \mathcal{O}\left(\sqrt{T}\right) \ .$$

Another example is the **passive aggressive** algorithm (PA) (Crammer et al., 2006). It is defined for a variety of learning tasks including classification, regression, and uni-class prediction and can be uniformly described by

$$\mathcal{A}^{\mathrm{PA}}(f,x,y) = \arg\min_{f' \in \mathcal{F}} \frac{1}{2}\|f - f'\|^2 \ \ s.t. \ \ \ell(f',x,y) = 0 \tag{2.9}$$

where for classification ℓ is the hinge loss, for regression the ϵ-insensitive loss, and for uni-class prediction (where no x is observed and $y = \mathcal{F}$) the loss is given by $\ell(f,y) = \max(|f - y| - \epsilon, 0)$. For all three variants, a closed form solution of Equation 2.9 can be given. E.g., in case of classification it is

$$\mathcal{A}^{\mathrm{PA}}(f,x,y) = f + \frac{\ell(f,x,y)}{\|x\|^2}yx \ .$$

In this case, the regret bound of PA is in $\mathcal{O}(\sqrt{T})$ (Orabona et al., 2015).

Note that there is a simple connection between the convergence rate of an optimization algorithm and its online regret bound, if data is drawn iid from a fixed target distribution.

Lemma 2.10. *Let $\mathcal{A}$ be an optimization algorithm with convergence rate $g : \mathbb{N} \to \mathbb{R}$ for a loss function ℓ, i.e., after $T \in \mathbb{N}$ examples the optimization error $\widehat{\epsilon}_T$ is in $\mathcal{O}(g(T))$, where g is a strictly monotonic decreasing function. Then for a sequence of examples drawn iid from a fixed target distribution the regret bound of $\mathcal{A}$ is in*

$$\mathcal{O}\left(\sum_{t=1}^{T} g(t)\right) \ .$$

Proof. Since $\mathcal{A}$ has a convergence rate of $g(t)$ it holds that

$$\ell(f_t) - \ell(f^*) \leq \widehat{\epsilon}_t \in \mathcal{O}(g(t)) \ ,$$

where $f_t \in \mathcal{F}$ is the model in round t and $f^* \in \mathcal{F}$ is the optimal model. Thus, the online regret can be bounded by

$$R_{\mathcal{A}}(T) = \sum_{t=1}^{T} \ell(f_t) - \ell(f^*) \leq \sum_{t=1}^{T} \widehat{\epsilon}_t \in \mathcal{O}\left(\sum_{t=1}^{T} g(t)\right) \ .$$

□

For example, if $\mathcal{A}$ has a convergence rate of $g(T) = 1/T$, then the regret bound is

$$\mathbf{R}_{\mathcal{A}}(T) \in \mathcal{O}\left(\sum_{t=1}^{T}\frac{1}{t}\right) = \mathcal{O}(\ln T) \ .$$

The last equality comes from the standard asymptotic formula for the harmonic sum. If instead $\mathcal{A}$ has a convergence rate of $g(T) = 1/\sqrt{T}$, then

$$\mathbf{R}_{\mathcal{A}}(T) \in \mathcal{O}\left(\sum_{t=1}^{T} \frac{1}{\sqrt{t}}\right) .$$

Since it holds that

$$2\sqrt{T} - 2 \leq \sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T}$$

it follows that $\mathbf{R}_{\mathcal{A}}(T) \in \mathcal{O}(\sqrt{T})$

This concludes the introduction of online learning and machine learning as optimization in general. The following section describes distributed machine learning approaches for both batch and online learning.

2.2. Distributed Machine Learning

Traditional learning algorithms are designed to be executed serially on a single processing node. This serial approach becomes infeasible, if (i) the training process on the entire data set takes too much time (for a given application scenario), if (ii) the data is too large to fit into the memory of a processing node, or if (iii) the cost of communication required to centralize distributed data sources is too high. Distributed learning makes use of several processing nodes to circumvent some, or all of these problems. For that, each learner processes parts of the data locally and communicates intermediate results to other nodes [7].

The goal is to obtain a model with a similar quality compared to the model learned (hypothetically) by the serial algorithm on a single processing node with a speedup proportional to the number of employed nodes. At the same time, the amount of communication between nodes must not exceed the capacities of the communication infrastructure. A common approach is to parallelize existing learning algorithms.

2.2.1. Embarrassingly Parallel Algorithms

First-order optimization algorithms allow to efficiently compute the gradients of the loss function in parallel. As an example, the SGD algorithm that can be employed both for batch and online learning can be efficiently parallelized. Assuming $m \in \mathbb{N}$ processing nodes, in round $t \in \mathbb{N}$ **parallel SGD** draws m examples $(x_t^1, y_t^1), \dots, (x_t^m, y_t^m)$ and computes the gradients $g_t^i = \nabla_{f_t} \ell(f_t, x_t^i, y_t^i)$ for each example in parallel on the m nodes. Then, the gradients are collected, summed up, and an update step

$$f_{t+1} = f_t - \eta \sum_{i=1}^{m} g_t^i$$

[7] Thus, this thesis only considers data parallelism. A different approach is model parallelism, where all processing nodes share the same data but update different parameters of the model.

is performed. The resulting model is equivalent to one computed centrally on a mini-batch of m examples (see mini-batch SGD in Section 2.1.3). Parallelizations of these type are referred to as "embarrassingly parallel" (Cevher et al., 2014; Moler, 1986, 1987). The model quality is optimal in this case and the speedup is high—only the averaging of gradients requires an overhead each round. This comes at the expense of having to centralize all gradients in each iteration, i.e., massive communication. This strategy of distributing the gradient computation can also be applied to GD (Mcdonald et al., 2009). GD has a convergence rate in $\mathcal{O}(1/T)$, whereas the one of SGD is in $\mathcal{O}(1/\sqrt{T})$. So while GD converges faster, each round is more computationally expensive, since it requires calculating the gradients for all data points. SGD instead converges slower but only requires calculating a single local gradient per round. Note that, since parallel SGD on m learners is equivalent to mini-batch SGD with a mini-batch size of m, the convergence rate of parallel SGD is actually in between SGD and GD. In serial mini-batch SGD, the mini-batch size $B \in \mathbb{N}$ controls the trade-off between a full GD (i.e., choosing B equal to the dataset size) and SGD (i.e., $B = 1$). The serial algorithm has a convergence rate of $\mathcal{O}(1/\sqrt{BT} + 1/T)$. Thus, parallel SGD on m learners has a convergence rate of $\mathcal{O}(1/\sqrt{mT} + 1/T)$ which is similar to SGD for $m = 1$ and approaches that of GD with growing m.

The middle ground between parallel GD and parallel SGD is the **distributed mini-batch SGD** algorithm (DMB) (Dekel et al., 2012) which performs mini-batch SGD on each node. Given local mini-batches $E_t^i \subset \mathcal{X} \times \mathcal{Y}$ of size $B \in \mathbb{N}$ at processing node $i \in [m]$, the DMB algorithm calculates local gradients

$$g_t^i = \sum_{(x,y) \in E_t^i} \nabla \ell(f_t, x, y)$$

and updates the model similar to parallel SGD. Note that, similar to SGD, DMB with parameter B on m nodes is equivalent to serial mini-batch SGD with $B' = mB$. The regret of DMB can be given with respect to the regret of serial online learning algorithm (i.e., SGD). Let σ^2 denote a bound on the variance of the gradient of the loss function ℓ, i.e., it hods for all $f \in \mathcal{F}$ and all $x \in \mathcal{X}, y \in \mathcal{Y}$ that $\|\nabla_f \ell(f, x, y)\| \leq \sigma^2$. The regret bound of many online learning algorithms, including SGD, depends on this bound of the gradient variance. Therefore, Dekel et al. (2012) consider the regret bound with respect to both the number of rounds $T \in \mathbb{N}$ and σ^2, i.e., a regret bound $\mathbf{R}\left(T, \sigma^2\right)$. However, instead of a classic worst case bound on the online regret (as defined in Section 2.1.5), they consider a bound on the expected regret

$$\exp\left[R\left(T\right)\right] \leq \mathbf{R}\left(T, \sigma^2\right) \quad .$$

Furthermore, they consider a streaming setting where examples are lost because of message passing times (examples received while an update step is performed are lost). Given a bounded message passing time $\mu \in \mathbb{R}$, Dekel et al. (2012) provide the following result on the expected regret.

Theorem 2.11 (Dekel et al. (2012)). *Let $\ell : \mathcal{F} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ be a Lipschitz-smooth convex loss function with σ^2-bounded gradient variance. Let the message passing time be bounded by $\mu \in \mathbb{R}$. If the expected regret of the serial online learning algorithm is bounded by $\mathbf{R}\left(T, \sigma^2\right)$, then the expected regret of the distributed mini-batch algorithm with mini-batch size $B \in \mathbb{N}$ over T*

examples is at most

$$(B + \mu)\boldsymbol{R}\left(\left\lceil \frac{T}{B + \mu} \right\rceil, \frac{\sigma^2}{B}\right) \ .$$

This result implies that for SGD, the DMB algorithm retains the (expected) regret of the serial online learning algorithm.

Using a small B (i.e., similar to parallel SGD) results in low computational costs per iteration, but a higher total amount of iteration and more consequently communication (each learner sends its local gradient per iteration). A large B (i.e., similar to parallel GD) results in high computational costs per iteration, but a lower number of iterations and less communication. In practice, DMB allows to chose a good trade-off between local computation costs and communication, rendering it favorable to both GD and SGD. Common to all variants is that they require a tightly coupled system with a low-latency and high-bandwidth network in order to not stall the iterations (McMahan et al., 2017).

2.2.2. Classical Parallelization of Optimization and Learning Algorithms

Quasi-Newton methods, like l-BFGS (Byrd et al., 1995), can be parallelized by computing the gradients in each iteration in parallel and summing them in the end, as well. Moreover, the approximation of the Hessian, i.e., the matrix of second order derivatives, can be parallelized. For MapReduce systems, there exist efficient parallelizations that require an alteration of the original algorithm in order to avoid constant map-reduce steps in each iteration (Chen et al., 2014). This altered algorithm parallelizes several computation steps in the original one so that the model obtained is similar to a centrally computed model, at the cost of high communication.

Moreover, a combination of first-order online methods and quasi-Newton methods can be used to process large amounts of data without the communication and computation overhead of parallel l-BFGS but with a similar performance. For that, the model generated by parallel SGD is used as starting point for the serial l-BFGS algorithm (Agarwal et al., 2014). Given that the starting point obtained by the first-order method is in a good neighborhood, the l-BFGS solver will converge much faster than with a random starting point.

A more systematic approach to this kind of parallelization is to identify calculations in summation form, i.e., sums over independent terms, and parallelize them using MapReduce (Chu et al., 2006). However, the locally computed terms (or gradients) have to be communicated across nodes in each iteration, which slows down the algorithms in practice. Moreover, all these approaches require an adapted parallel implementation of the learning algorithm. Practical implementations within a parallel computing framework require expert knowledge in the framework and are typically heavily tweaked (Meng et al., 2016).

Ensemble methods often also allow for parallelization. There, a set of models is trained using data from either different target distributions, different (sub-)sets of features, or different labels (Dietterich, 2000). In principle, each member of the ensemble can even be from a different model space. The prediction is obtained by aggregating the individual predictions of the models in the ensemble, e.g., by majority vote or weighted average. For example, bagging can be parallelized efficiently by computing local models in parallel on local data sampled from a common large dataset (with replacement) (Breiman, 1996). The performance of bagging can be improved by adaptive boosting (AdaBoost) (Lazarevic and Obradovic, 2002), i.e., iteratively sampling local datasets and changing the sample distribution in each iteration based on the performance of the local learners. However, it requires an exchange of local predictions and data (Chan et al., 1993).

2.2.3. Parallel Machine Learning Frameworks

The aforementioned parallel learning algorithms can be implemented using scalable parallelization frameworks. This allows to run them efficiently on clusters and clouds. A basic framework is the parameter server (Li et al., 2013) which provides a shared memory of the current model to a set of processing nodes, together with a large set of convenience functions. This also allows for asynchronous model updates (Recht et al., 2011).

Another basic framework is **MapReduce** (Chu et al., 2006). In this framework, the inputs are distributed to the processing nodes which perform a function on the inputs, denoted map operation. The results of the map function are then combined (in a potential tree-like manner) which is called the reduce operation[8]. As an example, assume a dataset $E \subset \mathbb{R}^d$ and the goal is to calculate the average Euclidean norm of all vectors in E. This takes time $\mathcal{O}(d|E|)$. In the MapReduce framework on $m \in \mathbb{N}$ processing nodes, the dataset is partitioned into local dataset $E^i \subset \mathbb{R}^d$ at node $i \in [m]$. The map function is the average Euclidean norm

$$\overline{E^i} = \frac{1}{|E^i|} \sum_{x \in E^i} \|x\|_2$$

of the local dataset. The reduce operation given two nodes $i, j \in [m]$ is $1/2(\overline{E^i} + \overline{E^j})$. For

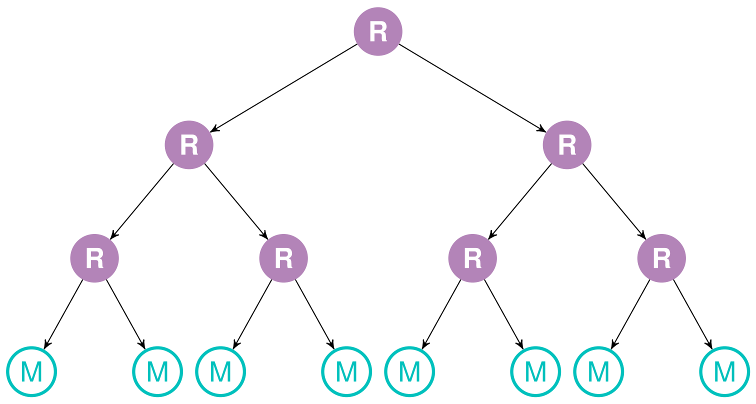


Figure 2.1.: Illustration of the binary aggregation tree in MapReduce. The leaves are associated with the result of the map operation. Each inner node computes the reduce operation on its children. The root node also corresponds to the output of the MapReduce operation.

[8] In an extended version of MapReduce, the output of the map operations is grouped by a so called shuffle operation and the reduce operation is applied per group.

simplicity assume that the size of E is a multiple of m and thus $|E^i| = {}^1\!/\!_m|E|$. Then the cost for the map operation, i.e., calculating the $\overline{E^i}$ in parallel, is in $\mathcal{O}(d\,{}^1\!/\!_m|E|)$. The reduce operation can be performed in the form of a binary (or n-ary) aggregation tree (see Figure 2.1 for an illustration), where the leaves contain $\overline{E^i}$ and each inner node corresponds to a reduce operation of its two children. Since this tree is of height $\log(m)$, the cost of the reduce operation is in $\mathcal{O}(d \log m)$. The overall runtime for the calculation thus is in $\mathcal{O}(d({}^1\!/\!_m|E| + \log(m)))$.

A more sophisticated framework based on MapReduce is Apache Spark. Its machine learning library MLI on Spark (Sparks et al., 2013; Zaharia et al., 2012) has implemented efficient variants of parallel SGD and distributed mini-batch SGD as base solvers for a set of machine learning algorithms (e.g., the ones described in Section 2.1.3). An implementation of the Radon machine which is presented in Chapter 5 in Apache Spark is described in Appendix A.2.

While these frameworks are intended for batch learning, handling data streams requires a framework that allows real-time processing of distributed data streams. Popular frameworks are Apache Storm (Apache Software Foundation, 2017) and Apache Flink (Carbone et al., 2015). An implementation of dynamic averaging that is presented in Chapter 3 in Apache Storm is described in Appendix A.1.

2.2.4. Towards Black-Box Parallelizations: Averaging Models

In a survey on distributed learning algorithms, Shamir and Srebro (2014) found that for algorithms based on SGD the best so far known algorithm is the distributed mini-batch algorithm (Dekel et al., 2012). It performs well in terms of runtime and model quality on tightly connected distributed systems (Chen et al., 2016), e.g., data centers and clusters. For many applications, however, centralization or even periodic sharing of gradients between local devices becomes infeasible due to the large amount of necessary communication. Moreover, the distribution of gradient computation requires low latency networks and are thus not suitable for highly distributed systems or MapReduce-style computations (McMahan et al., 2017).

Instead of sharing gradients, it was suggested to perform local updates and average the models. This model averaging can be performed once at the end (Zinkevich et al., 2010), or periodically (McDonald et al., 2010; McMahan et al., 2017; Zhang et al., 2015), allowing to further reduce communication (Li et al., 2014). Even though, averaging once at the end requires the least amount of communication, the result can be arbitrarily bad (Shamir and Srebro, 2014). Thus, typically periodic averaging of model parameters is used in convex optimization (Mcdonald et al., 2009; Shamir, 2016; Zhang et al., 2012). For convex optimization problems, the objective can also be adapted for faster convergence (Li et al., 2014). In the context of deep learning, periodic averaging is empirically analyzed in McMahan et al. (2017). For decentralized setups it was termed Federated Learning (McMahan et al., 2017) (see Section 4.3.1 for a discussion of Federated Learning).

Averaging model parameters—instead of communicating intermediate results—points towards a more general principle for distributed machine learning: run multiple instances of a machine learning algorithm in parallel and aggregate the resulting models. Aggregating models has three major advantages: (i) sending only the model parameters instead of a set of data reduces communication; (ii) it allows to train a joint model without exchanging or centralizing privacy-sensitive data; (iii) it can be readily applied to a wide range of learning algorithms, since it treats the underlying algorithm as a black-box. The following section defines this general principle, which I term **black-box parallelization**, and defines a formal framework to analyze such black-box parallelizations.

2.3. Black-Box Distributed Machine Learning

A particular approach to distributed machine learning is black-box parallelization. In this approach, an existing learning algorithm—referred to as **base learning algorithm**—is applied in parallel on distributed data to produce a set of models. These models are then aggregated into a single one. The goal for this single aggregated model is to achieve a similar quality compared to one produced by the serial application of base learning algorithm on all data centrally. Since the aggregation is independent of the base learning algorithm, it does not have to be altered or reimplemented—it can be regarded as a black box.

The major difference to ensemble methods—where several models are trained as well—is that the set of models is not maintained, but aggregated into a single model. A single model requires less memory and less computational power for evaluation which is an important advantage in applications with restricted resources. This is an important property in practice: for example, in autonomous driving, individual models are often deep neural networks (Chen et al., 2015) and the computing power within the vehicle is limited. Maintaining and executing an ensemble of multiple deep neural networks within the vehicle is infeasible (Berger and Dukaczewski, 2014). Moreover, in online learning scenarios with real-time constraints, the execution of an ensemble might require too much time. For example in online advertisement, the advertiser has only 100 milliseconds to apply the model to the auctioned ad-placement and make a bid, including message passing time (Muthukrishnan, 2009). Such scenarios require very low computational complexity of the model application. Thus, in such scenarios a method that yields a single strong model is required.

The major difference to existing distributed learning approaches is that the parallelization is decoupled from the learning process. Apart from the generality of such an approach, this has the advantage that the learning process is not stalled by unresponsive learners: either the aggregation of models is delayed or unresponsive learners are simply excluded from the aggregation. This is beneficial in massively distributed or decentralized systems. For example, in autonomous driving on electric cars, vehicles might only communicate when parked or charged. In learning on mobile phones, devices can be frequently disconnected from the network, out of battery, or on slow or expensive connections. Moreover, the data generated on the devices can be privacy-sensitive. In these scenarios, classical parallelizations are not applicable (McMahan et al., 2017).

Black-box parallelizations allow to tackle these difficulties: they produce a single strong model, are able to deal with limited connectivity and network capacity, and do not require to share data. For that, an aggregation method is required that improves locally trained models, as well as a scheduling of this aggregation. Based on this, a framework for black-box parallelizations is formally defined in the following section.

2.3.1. Distributed Learning Protocol

We assume a distributed learning system of $m \in \mathbb{N}$ processing nodes, referred to as **local learners**. These can be the machines in a cluster, or cloud, but also other devices with processing power, such as mobile phones, smart sensors, or vehicles.

Let $\mathcal{A} \colon \mathcal{X} \times \mathcal{Y} \to \mathcal{F}$ be a learning algorithm that, given a dataset $E \subset \mathcal{X} \times \mathcal{Y}$, generates a model $f \in \mathcal{F}$. Each learner $i \in [m]$ runs the learning algorithm $\mathcal{A}$ on a local dataset $E^i \subset \mathcal{X} \times \mathcal{Y}$ and obtains a local model $f^i \in \mathcal{F}$. The tuple of local models forms a **model configuration** $\mathbf{f} = \left(f^1, \dots, f^m\right)$. An **aggregation operator** $\mathfrak{a} \colon \mathcal{F}^m \to \mathcal{F}$ aggregates a model configuration $\mathbf{f} = (f^1, \dots, f^m)$ into a single model $f = \mathfrak{a}(\mathbf{f})$.

The most commonly used aggregation operator is the (weighted) average of local models (e.g., (Liu and Ihler, 2014; McMahan et al., 2017; Polyak and Juditsky, 1992; Shamir, 2016; Zinkevich et al., 2010)), i.e.,

$$\mathfrak{a}(f^1, \dots, f^m) = \sum_{i=1}^{m} \frac{w^i f^i}{\sum_{j=1}^{m} w^j} \quad ,$$

where $w^1, \dots, w^m \in \mathbb{R}$ are arbitrary weights, and the case $w^1 = \dots = w^m = {}^1\!/_m$ is the standard average. An aggregation operator more robust to outliers is the (geometric) median (Hsu and Sabato, 2016; Minsker et al., 2015)

$$\mathfrak{a}(f^1, \dots, f^m) = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{m} \left\| f^i - f \right\|_2 \quad .$$

Chapter 5 presents an aggregation operator based on the Radon point (Radon, 1921) that shares robustness properties with the median and provides strong guarantees on the model regret.

The aggregation operator defines how models are combined; for a parallel learning algorithm it remains to be defined when the aggregation is performed. That is, in a distributed system with m learners, each running algorithm $\mathcal{A}$, and respective local models $f^1, \dots, f^m$, a **synchronization operator** $\sigma \colon \mathcal{F}^m \to \mathcal{F}^m$ schedules the aggregation of some, or all local models. This is called a **synchronization**. Thus, σ has two purposes: it decides when to aggregate and which models are included in the aggregation. Note that σ and $\mathfrak{a}$ are not independent, but some σ can be used with multiple $\mathfrak{a}$.

The question when to aggregate depends on whether the employed learning algorithm is incremental or not. If it is non-incremental, the aggregation can only be performed after training is completed. That is, σ leaves the models unaltered until all local data is processed and then executes $\mathfrak{a}$ on all, or a subset of them. This synchronization strategy is referred to as **aggregation-at-the-end** (e.g., (Mcdonald et al., 2009; Zinkevich et al., 2010)). If instead the algorithm is incremental, aggregations can be performed during the training process. For incremental learning algorithms, a viable strategy is to aggregate models periodically. That is, each local learner trains for a fixed number of rounds in which the synchronization operator leaves the models unaltered. After this number of rounds, the synchronization operator applies the aggregation operator to all locally generated models and the global aggregate is used as a starting point for the next round (Li et al., 2014; McMahan et al., 2017).

Since the aggregation and synchronization operators only require the output of the base learning algorithm $\mathcal{A}$, this form of parallelization treats $\mathcal{A}$ as a black box. I propose to use the synchronization and aggregation operators as a generic framework to describe **black-box parallelizations** of learning algorithms. In this framework, a particular combination of learning algorithm, synchronization, and aggregation operator is denoted a distributed learning protocol.

Definition 2.12. *Let $\mathcal{A}$ be a learning algorithm, $\sigma : \mathcal{F}^m \to \mathcal{F}^m$ be a synchronization operator and $\mathfrak{a} : \mathcal{F}^m \to \mathcal{F}$ be an aggregation operator. Given a distributed system with $m \in \mathbb{N}$ learners, a **distributed learning protocol** $\Pi = (\mathcal{A}, \sigma, \mathfrak{a}, m)$ executes $\mathcal{A}$ in parallel on the m nodes and applies σ to the locally generated models using the aggregation operator $\mathfrak{a}$.*

When the aggregation operator and the number of learners are obvious, the distributed learning protocol can be abbreviated as $\Pi = (\mathcal{A}, \sigma)$.

Note that the learning performance of using specific synchronization and aggregation operators can be theoretically analyzed without considering their actual implementation (including the network topology they are used in). In order to assess the communication and speedup, however, their implementation has to be taken into account. Thus, in the following the implementation of a distributed learning protocol is provided when it is introduced. A discussion of the impact of network topologies on communication and speedup is provided in Section 3.4. In case the implementation is clear, the operators are identified with their implementation.

In the following section, the notion of speedup of such distributed learning protocols is introduced.

2.3.2. Speedup of Distributed Learning Protocols

A major goal in parallelization is to decrease the runtime of an algorithm by using multiple processing units. The **speedup** of a parallelization on m processing units is defined as the runtime of the serial algorithm divided by the runtime of the parallelization (Kruskal et al., 1990). Classical parallelizations of machine learning algorithms produce the same result as the serial algorithm. In order to determine their speedup it suffices to analyze the runtime of the parallelization with respect to the number of learners.

Black-box parallelizations often do not produce the same output as the serial execution of the base learning algorithm. For example, executing stochastic gradient descent for m rounds is not equivalent to executing it for one round in parallel on m learners and averaging or summing the models. The reason is that serial SGD calculates the gradient of the t-th example using the model f_t, whereas the parallel execution determines the gradient for all examples using the initial model f_0. So while for black-box parallelizations of machine learning algorithms it is often not possible to obtain the same output, the parallelization may yet obtain a model of the same quality.

Definition 2.13. *Let $\mathcal{A}$ be a machine learning algorithm using a model space $\mathcal{F}$ and $\Pi = (\mathcal{A}, \sigma, \mathfrak{a}, m)$ be a distributed learning protocol. Let $Q : \mathcal{F} \to \mathbb{R}_+$ be some quality function. Given a desired quality $q \in \mathbb{R}$, let $\mathcal{T}_{\mathcal{A}}$ denote the runtime of $\mathcal{A}$ to produce a model $f \in \mathcal{F}$ with $Q(f) = q$.*

Let $\mathcal{T}_\Pi(m)$ denote the minimal runtime of the distributed learning protocol on $m \in \mathbb{N}$ learners to produce a model $f' \in \mathcal{F}$ of at least the same quality, i.e., $Q(f') \geq q$. Then the speedup of Π on m learners is defined as $\mathcal{T}_\mathcal{A}/\mathcal{T}_\Pi(m)$.

The quality function Q can be arbitrarily defined. For example, given an $\epsilon > 0$, the quality could be measured by the confidence $\delta \in (0,1]$ of a generalization bound achieved by an algorithm. In online learning, the quality is higher the less loss an algorithm suffers, so a natural choice would be the negative (or inverse) cumulative loss.

The runtime of the distributed learning protocol can be decomposed into the runtime of the base learning algorithm executed in parallel and the runtime of all synchronization. As an example, in the following the periodically averaging approach is discussed and its speedup is analyzed.

2.3.3. An Example: Periodic Averaging

Assume an incremental learning algorithm $\mathcal{A}$ for linear models, i.e., $\mathcal{F} = \mathbb{R}^d$ for some $d \in \mathbb{N}$ (see Section 2.1.1). As learning algorithm assume mini-batch SGD $\mathcal{A}^{\text{mSGD}}$ (see Section 2.1.3). This serial learning algorithm is parallelized using averaging as aggregation operation, i.e.,

$$\mathfrak{a}_{AVG}(f^1,\dots,f^m) = \overline{\mathbf{f}} = \frac{1}{m}\sum_{i=1}^m f^i \quad .$$

For linear models the average of $m \in \mathbb{N}$ models $f^1,\dots,f^m$ is the standard vector average in $\mathbb{R}^d$—recall that we identify the model f^i with its parameterization $w \in \mathbb{R}^d$. With this, the periodic averaging operator with period $b \in \mathbb{N}$ for a model configuration $\mathbf{f}_t = (f_t^1,\dots,f_t^m)$ in round $t \in \mathbb{N}$ is defined as

$$\sigma_b(\mathbf{f}_t) = \begin{cases} (\overline{\mathbf{f}}_t,\dots,\overline{\mathbf{f}}_t), & \text{if } b \mid t \\ \mathbf{f}_t, & \text{otherwise} \end{cases} \quad .$$

Here, $b \mid t$ means b divides t. The periodic averaging protocol using mini-batch SGD is then given by $\mathcal{P} = (\mathcal{A}^{\text{mSGD}}, \sigma_b, \mathfrak{a}_{AVG}, m)$. The protocol can be implemented using a single designated coordinator node to perform the averaging. This implementation is given in Algorithm 3. This implementation communicates $2m$ models every b rounds.

In terms of communication cost, every distributed learning protocol lies between two extreme baselines—continuously communicating and quiescence, i.e., no communication at all. Intuitively, investing more communication leads to a better model quality: If the aggregation operator leads to an improvement in model quality over the individual ones (in expectation), the overall quality increases with the number of synchronizations. For incremental learning algorithms, a good trade-off between predictive performance and communication can be achieved by communicating periodically (for non-incremental algorithms, the only viable strategy is aggregation-at-the-end).

To illustrate the trade-off between predictive performance and communication with respect to the parameter b, an experiment was conducted using $m = 256$ learners jointly learning from a synthetic dataset in an online fashion. That is, each learner observes an instance, performs a prediction, receives the true label and updates its local model accordingly. The performance of the learners is measured by their cumulative loss.

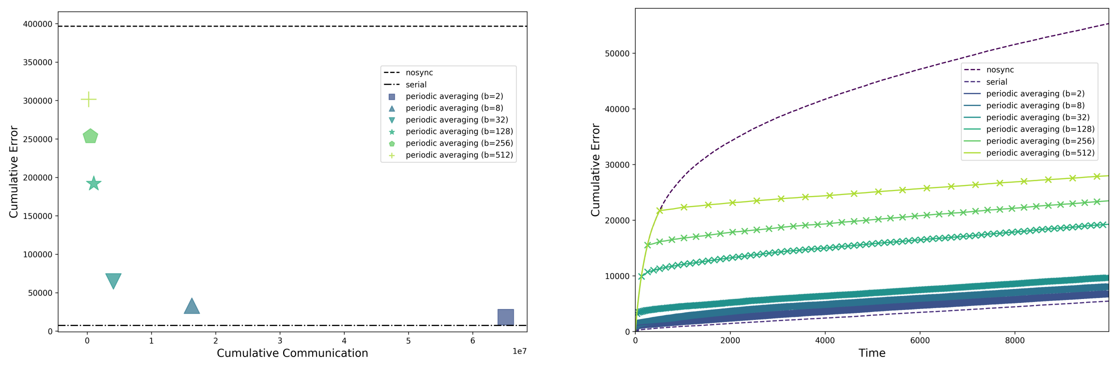
Algorithm 3 Periodic Averaging Protocol

```
Input: learning algorithm $\mathcal{A}$, parameter $b$, $m$ learners
Initialization:
  local models $f_1^1, \dots, f_1^m \leftarrow$ one random $f$
Round $t$ at learner $i$:
  observe $E_t^i \subset \mathcal{X} \times \mathcal{Y}$
  update $f_{t-1}^i$ using the learning algorithm $\mathcal{A}$
  if $t \mod b = 0$ then
    send $f_t^i$ to coordinator
  end if
At coordinator every $b$ rounds:
  receive local models $\mathbf{f}_t = (f_t^1, \dots, f_t^m)$
  for all $i \in [m]$ do
    set $f_t^i \leftarrow \sigma_b(\mathbf{f}_t)^i$ and send to learner $i$
  end for
```

Figure 2.2 shows the results for multiple instances of $\mathcal{P}$ with different periods $b \in \mathbb{N}$, compared to a protocol that doesn't communicate, denoted nosync, and the serial application of $\mathcal{A}^{\text{mSGD}}$ on all data. Figure 2.2(a) shows the cumulative loss and cumulative communication after processing all data, while Figure 2.2(b) shows the development of cumulative zero-one loss over time with communication indicated by crosses.

As expected, the serial application of $\mathcal{A}^{\text{mSGD}}$ has the lowest cumulative loss. However, it requires centralizing all data and processing it on a single processor. The nosync baseline instead processes the data distributedly and does not communicate at all, but has the highest loss. The performance of periodic averaging increases with the amount of communication invested (i.e., with smaller period b) up to a performance similar to the serial baseline. To achieve this, however, it has to average every other round, requiring a substantial amount of communication. Increasing the communication period allows to trade communication for predictive performance and adapt it to the desired trade-off.

After having seen that periodic averaging allows to train models of high quality in parallel, it remains to analyze its speedup. The speedup of periodic averaging depends on how many examples the system with $m \in \mathbb{N}$ learners needs to process to achieve the same model quality as the serial baseline. Here, quality is defined in terms of cumulative loss. The results for $b = 2$ and $b = 8$ indicate that periodic averaging run for $T \in \mathbb{N}$ rounds, thus processing Tm examples, achieves a model quality comparable to the serial baseline run for Tm rounds, processing the same amount of examples (the following chapter will investigate this conjecture both theoretically and empirically). For this example, assume that indeed periodic averaging achieves the same model quality processing the same number of examples. Then, the runtime $\mathcal{T}_{\mathcal{P}}(T)$ for T rounds can be divided into the runtime $\mathcal{T}_{\mathcal{A}^{\text{mSGD}}}$ of $\mathcal{A}^{\text{mSGD}}$ in parallel on the m learners, each processing T examples, and the time required for each synchronization times the number of synchronizations. Mini-batch SGD is a linear time algorithm, thus its runtime for



(a) cumulative loss and communication (b) development of cumulative loss

Figure 2.2.: Experiment with periodic averaging on $m = 256$ learners after training on a dataset of size $|E| = 2\,560\,000$ (i.e., 10 000 examples per learner) from a synthetic classification dataset. The dataset is generated by applying a randomly chosen disjunction to 50-dimensional random binary vectors. Note that the serial baseline observes only 1 example per round, while the distributed approaches jointly process $m = 256$ examples. To align the plots, for each round the sum of 256 rounds of the serial baseline is shown.

processing T examples is in $\Theta(T)$. The time for calculating the average of m models is in $\Theta(m)$. Note that the time for calculating the average can be straight-forwardly reduced to $\Theta(\log m)$ by calculating the average in a MapReduce fashion without additional communication (see Section 2.2). Periodic averaging with parameter $b \in \mathbb{N}$ synchronizes T/b times in T rounds. Thus, viewing b as a constant, the runtime of periodic averaging is

$$\mathcal{T}_{\mathcal{P}} \in \Theta\left(\mathcal{T}_{\mathcal{A}^{\mathrm{mSGD}}}(T) + \frac{T}{b}\log m\right) = \Theta\left(T + T\log m\right) = \Theta\left(T\log m\right) \quad .$$

At the same time, mini-batch SGD processing Tm examples has a runtime of

$$\mathcal{T}_{\mathcal{A}^{\mathrm{mSGD}}} \in \Theta\left(Tm\right) \quad .$$

Thus, the speedup of periodic averaging on m learners using mini-batch SGD is

$$\frac{\mathcal{T}_{\mathcal{A}^{\mathrm{mSGD}}}}{\mathcal{T}_{\mathcal{P}}} \in \Theta\left(\frac{Tm}{T\log m}\right) = \Theta\left(\frac{m}{\log m}\right) \quad .$$

The overhead of averaging is reducing the achievable speedup substantially—averaging models naively would even lead to no theoretical speedup at all. There are two ways of reducing the overhead costs of synchronizations: (i) to further reduce the runtime cost of averaging, or (ii) to decrease the number of synchronizations.

Further reducing the runtime cost of averaging can be achieved by averaging models blockwise in a peer-to-peer fashion. The model parameters are divided in as many blocks as there are learners. Each learner broadcasts its model to all other learners. Each learner then averages the block assigned to it and broadcasts the result. This way, the average can be computed in time $\mathcal{O}(1)$ at the expense of $2m^2$ messages per synchronization (that are only $1/m$ of the size of a normal averaging message). However, this approach is only applicable if the number of model parameters is larger than the number of learners. Instead, sending all models to a central coordinator and broadcasting the average back requires $2m$ full size messages, the same holds for map-reduce-like averaging. Moreover, both approaches can be applied independently of the number of model parameters.

Decreasing the number of synchronizations can be achieved by increasing the parameter b, but as Figure 2.2 shows that leads to substantially worse model quality. For any choice of b, the amount of communication is independent of its utility. In particular, when local learners do not suffer loss at all, communication is unnecessary and should be avoided; similarly, when they suffer large losses, an increased amount of communication should be invested to improve their performances. Ideally, the synchronization operator would invest a lot of communication in the beginning and save communication as soon as the model sufficiently converged. The next chapter presents a dynamic approach that tackles these shortcomings. This approach is first applied to online learning in Chapter 3 and then to batch learning in Chapter 4.

3. Efficient Distributed Online Learning

This chapter investigates black-box parallelizations for online learning algorithms. Recall that in online learning it is assumed that in each round $t \in \mathbb{N}$ the learner observers an instance $x_t \in \mathcal{X}$ and makes a prediction $\widehat{y}_t \in \mathcal{Y}$ using its current model f_t. After that, it observes the correct label $y_t \in \mathcal{Y}$. An online learning algorithm $\mathcal{A}$ then updates the current model $f_{t+1} = \mathcal{A}(f_t, x_t, y_t)$ with the goal to make as few mistakes as possible during this process. In the distributed case, all local learners aim at minimizing their mistakes. Section 3.1 formally defines distributed online learning and provides two criteria for the efficiency of distributed learning protocols: (i) they should be consistent, i.e., retain the in-place performance of the serial online learning algorithm and (ii) they should be adaptive, i.e., invest communication relative to the current hardness of the learning problem. It will be shown that periodically communicating protocols are non-adaptive.

The goal of this chapter is to develop a distributed learning protocol that provides high performance for all local models in each round while explicitly minimizing communication. At the same time, the distribution should yield substantial speedup. The protocol should be theoretically sound, i.e., with strong loss bounds and bounds on the communication.

In the previous chapter, periodic averaging was presented as an example of a distributed learning protocol. With a sufficient amount of communication, this protocol is able to achieve a cumulative loss comparable to the serial execution of the base learning algorithm. However, the amount of communication required for that grows linearly with the number of rounds, independent of the utility of the communication. The main idea to address this problem is to perform model synchronizations only in system states that show a high divergence among the local models, where divergence is defined as the distance to the aggregate of the models. A high divergence indicates that a synchronization would be most effective in terms of its correcting effect on the model quality.

The divergence can be defined with respect to arbitrary distance functions between models and arbitrary aggregation operators. Since the model divergence is a non-linear function in the global system, it cannot be monitored trivially without communicating. However, by decomposing it into local conditions, it can be monitored locally in a communication-efficient way. For specific aggregation operators, i.e., those that minimize the divergence, I derive a general set of local conditions that allow to devise a communication-efficient distributed learning protocol for online learning. This protocol is described and analyzed with respect to averaging as aggregation operator in Section 3.2 and denoted dynamic averaging. It allows communicative quiescence in stable phases, while, in hard phases where divergence reduction is crucial, invests more communication.

Particularly interesting application scenarios for online learning are real-time services on data streams, i.e, timely predictions on arrival of each example from distributed, potentially high-frequency data streams. Each individual learner processes its corresponding data stream and updates its local model accordingly. The synchronizations should improve the in-place performance of the local learners and thus the overall performance of the system—Section 3.3 analyses under which conditions synchronizations provably improve the performance. Such real-time services are of high importance in practice, e.g., in online ad-placement (Muthukrishnan, 2009; Yuan et al., 2013), stock market investment recommendations (Kamp et al., 2013), network intrusion detection for cyber security (Buczak and Guven, 2016), and anomaly detection in sensor networks for, e.g., for health care (Alemdar and Ersoy, 2010). Such applications require a responsive learning system. As discussed in the example of periodic averaging (Section 2.3.3), communication can stall a learning system, reducing not only the achievable speedup but also the responsiveness. Thus, reducing communication is imperative for these applications.

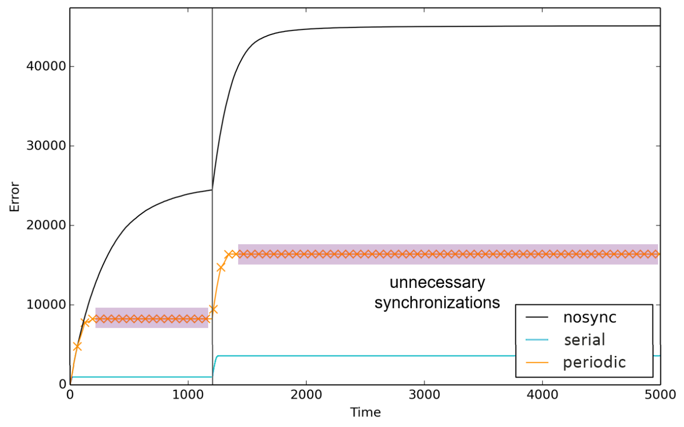


Figure 3.1.: Cumulative error of nosync, serial, and periodic averaging ($b = 50$) using SGD on a synthetic classification task. The task is to learn a random disjunction $dis\colon \{-1,1\}^d \to \{-1,1\}$. That is, a random disjunction dis over d binary inputs is generated; then, random binary vectors $x \in \{-1,1\}^d$ are generated as input with the respective label $y = dis(x)$. The crosses indicate synchronizations. The vertical lines indicate a concept drift, simulated by randomly choosing a new disjunction.

Moreover, when learning from potentially infinite data streams, the underlying target distribution may change[1]. Such a change is referred to as **concept drift**. Formally, assume training sets E_t drawn iid from a time-variant distribution $\mathcal{D}_t : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$, then a concept drift occurs in round $t \in \mathbb{N}$ if $\mathcal{D}_{t-1} \neq \mathcal{D}_t$. Periodic approaches cannot react adequately to such drifts, since they either communicate so rarely that the models adapt too slowly to the change, or they communicate so frequently that they generate an immense amount of unnecessary communication in-between drifts (see Figure 3.1 for an illustrative example).

Section 3.3 analyzes the in-place performance of dynamic averaging and the required amount of communication, showing that for particular learning algorithms, dynamic averaging is indeed efficient, i.e., consistent and adaptive. Moreover, its speedup over the serial application of the learning algorithm is shown to be at least as high as that of periodic averaging.

The protocol can be applied to various network topologies, a few of which are discussed in Section 3.4. The theoretical findings are substantiated by an empirical evaluation in Section 3.5. The results, as well as some practical aspects of dynamic averaging are discussed in Section 3.6.

3.1. Efficient Distributed Online Learning

This chapter considers distributed online learning, where $m \in \mathbb{N}$ **local learners** each run an online learning algorithm $\mathcal{A}$ that maintains a **local model** f^i from a common model space $\mathcal{F}$. Similar to serial online learning (see Section 2.1.5), in each round $t \in \mathbb{N}$ each learner $i \in [m]$ observes a local instance $x_t^i \in \mathcal{X}$ and makes a prediction $\widehat{y}_t^i = f_t^i(x_t^i)$. Upon observing the true label[2] $y_t^i \in \mathcal{Y}$, the quality of the prediction is measured by a loss function ℓ and the local model is updated using $\mathcal{A}$.

The goal of such a distributed learning system is to provide accurate predictions in each round. The quality of the predictions is measured—similar to classical online learning—by the cumulative loss over all learners (see 2.1.5 in the previous chapter)

$$L(T,m) = \sum_{t=1}^{T}\sum_{i=1}^{m} \ell\left(f_t^i, x_t^i, y_t^i\right) \quad .$$

A synchronization operator σ is used to improve the system's performance. An illustration of such a setting is shown in Figure 3.2, in which σ is executed in a network topology with a dedicated node for running it, called the **coordinator**. Different network topologies are discussed in Section 3.4. The performance in terms of communication is measured by its cumulative communication

$$C(T,m) = \sum_{t=1}^{T} c(\mathbf{f}_t) \quad ,$$

[1] The online learning model does not require assumptions on the target distribution. However, if a fixed target distribution is assumed—which is natural, e.g., in a non-streaming setting— and data is drawn iid from it, then the cumulative loss of a distributed learning system is related to its convergence rate in the batch setting (see Lemma 2.10 in Section 2.1.5)

[2] Here it is assumed that the true label is available right after the prediction has been made. In practice, however, this feedback can be delayed severely. In this case, the online learning algorithm has to store instances in memory until the true output arrives. Moreover, the delay in feedback decreases the in-place performance. See Joulani et al. (2013) for a detailed discussion on delayed feedback in online learning.

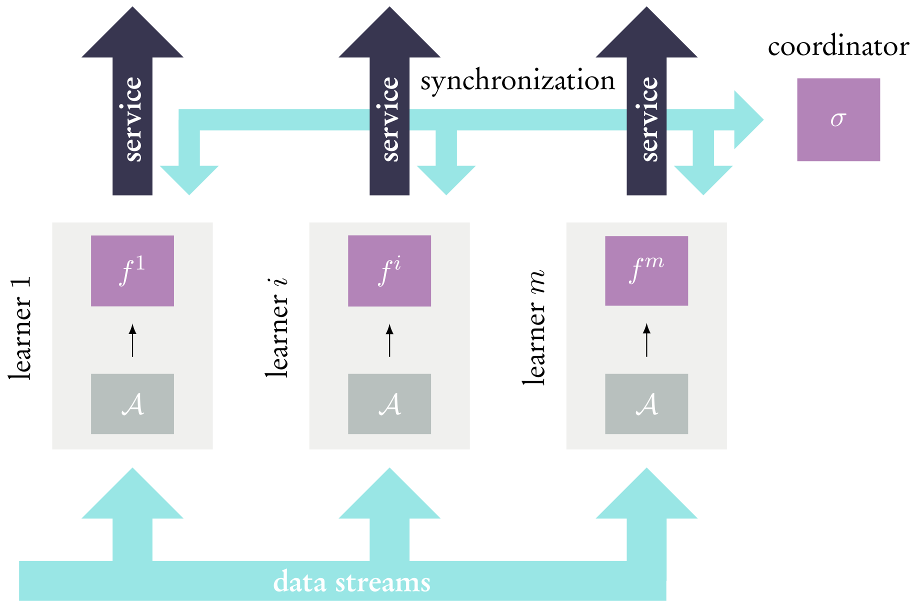


Figure 3.2.: Illustration of of a distributed real-time service: $m \in \mathbb{N}$ learners receive input from individual data streams, each learner $i \in [m]$ running an online learning algorithm $\mathcal{A}$ that maintains a local model f^i which is used to provide a real-time service (e.g., a prediction for each incoming input). Local models are synchronized via a coordinator using a synchronization operator σ to improve the overall service quality.

where $c : \mathcal{F}^m \to \mathbb{N}$ denotes the number of bytes required to synchronize models $\mathbf{f}_t = (f_t^1, \dots, f_t^m)$ in round t. The following section discusses the trade-off between communication and predictive performance and provides an efficiency criterion for distributed online learning protocols.

3.1.1. Efficiency Criterion

There is a natural trade-off between communication and loss of a distributed online learning system. On the one hand, a loss similar to a serial setting can often be achieved by continuous synchronization. On the other hand, communication can be entirely omitted. For these two extreme protocols, the trade-off can be quantified.

If the cumulative loss of an online learning algorithm $\mathcal{A}$ is given by $L_{\mathcal{A}}(T)$, the loss of a continuously communicating system, i.e., one that centralizes all data from the m learners and applies $\mathcal{A}$ serially is $L_{\mathcal{C}}(T, m) = L_{\mathcal{A}}(mT)$. This is equivalent to the loss of a serial online learning algorithm processing mT inputs. The same holds for distributed mini-batch SGD (Dekel et al., 2012) with batch size $b = 1$ and for periodic averaging with SGD and $B = 1$ (the latter is shown in Section 3.3.3). The protocol transmits $\mathcal{O}(m)$ messages in every of the T rounds. The size of the messages depends on whether data, or models, or gradients are centralized. For data and gradient centralization, the message size is in $\mathcal{O}(1)$. Similarly, for model centralization using linear models, or neural networks. For kernel models, the message size is in $\mathcal{O}(t)$ in round $t \in \mathbb{N}$, because the number of support vectors grows with the number of observed examples.

The loss bound of a distributed system without any synchronization is given by $\mathbf{L}_{\text{nosync}}(T,m) = m\mathbf{L}_{\mathcal{A}}(T)$, i.e., the loss bound of m independent learners processing T examples. At the same time, the communication is $C(T,m) = 0$.

Since the loss is typically sublinear in T, not communicating leads to substantially higher cumulative loss. Consider SGD as an example for which not the cumulative loss but the online regret, i.e.,

$$R_{\mathcal{A}}(T) = \sum_{t=1}^{T} \ell(f_t, x_t, y_t) - \ell(f^*, x_t, y_t) \ ,$$

(see Equation 2.1.5 in Section 2.1.5) can be bounded: for SGD the online regret is in $\mathcal{O}(\sqrt{T})$. Thus, the regret bound of a continuously communicating protocol is in $\mathbf{R}_{\mathcal{C}}(T,m) \in \mathcal{O}(\sqrt{mT})$ and that of a non-synchronizing one is in $\mathbf{R}_{\text{nosync}}(T,m) \in \mathcal{O}(m\sqrt{T})$. It follows that not communicating leads to a regret bound worse by a factor of $\sqrt{m}$.

As seen in the example in Section 2.3.3, periodic averaging allows to tune a learning system to a middle ground between these two extremes. However, it invests communication independent of its utility, synchronizing too little when it could substantially improve performance or too much when it does not—a disadvantage that is even more severe when concept drifts occur. An efficient protocol should be adaptive to the utility of synchronizations. That implies that the communication of an adaptive protocol should only depend on $L_{\mathcal{A}}(T)$ (as a proxy for the hardness of the learning problem) and not on T, while at the same time retaining the loss bound of the serial setting. In the following definition, this intuition is formalized in order to provide a strong criterion for efficiency of distributed online learning protocols.

Definition 3.1. *A distributed online learning protocol* $\Pi = (\mathcal{A}, \sigma, \mathfrak{a}, m)$ *processing* mT *inputs is* ***consistent*** *if it retains the cumulative loss of the serial online learning algorithm* $\mathcal{A}$*, i.e.,*

$$L_{\Pi}(T,m) \in \mathcal{O}\left(L_{\mathcal{A}}(mT)\right) \ .$$

The protocol is ***adaptive*** *if its communication is linear in the number of local learners* m *and the cumulative loss* $L_{\mathcal{A}}(mT)$ *of the serial online learning algorithm, i.e.,*

$$C_{\Pi}(T,m) \in \mathcal{O}\left(mL_{\mathcal{A}}(mT)\right) \ .$$

An ***efficient*** *protocol is adaptive and consistent at the same time.*

A periodically communicating protocol cannot be efficient, because it cannot be adaptive: If a serial learner can achieve zero loss, then for a consistent periodic protocol $L_{\mathcal{A}}(mT)$ is in $\mathcal{O}(1)$. However, the communication $C_{\Pi}(T,m)$ is in $\mathcal{O}(mT)$, and not in $\mathcal{O}(m \cdot 1)$.

The following section introduces dynamic averaging, a protocol that adapts communication to the current hardness of the learning problem. In Section 3.3 it is then shown that for particular online learning algorithms, dynamic averaging is indeed efficient as in Definition 3.1.

3.2. Dynamic Averaging Protocol

Intuitively, the communication for performing model averaging is not well invested in situations where all models are already approximately equal—either because they were updated to similar models or have merely changed at all since the last synchronization. Especially, if all local models have already converged to the optimal model, averaging will hardly change the models and their performance. A periodic protocol still averages in such cases (recall Figure 3.1 for an illustration). Ideally, a protocol should only perform model averaging when this communication has a substantial effect.

3.2.1. Partial Synchronization

A simple measure to quantify the effect of synchronizations is given by the **divergence** of the current local model configuration.

Definition 3.2. *Let* $\mathbf{f} = (f^1, \ldots, f^m) \in \mathcal{F}^m$ *be a model configuration of* m *models from a model space* $\mathcal{F}$*,* $d\colon \mathcal{F} \times \mathcal{F} \to \mathbb{R}_+$ *a distance function on* $\mathcal{F}$*, and* $\mathfrak{a}$ *an aggregation operator. Then the* ***divergence*** *of* $\mathbf{f}$ *is defined as*

$$\delta(\mathbf{f}) = \frac{1}{m}\sum_{i=1}^{m} d\left(f^i, \mathfrak{a}(\mathbf{f})\right) \quad .$$

The following definition provides a generic synchronization operator that schedules the aggregation of models based on the model divergence. This operator is a relaxation of periodic aggregation operators, such as periodic averaging.

Definition 3.3. *A* ***partial synchronization operator*** *with positive divergence threshold* $\Delta \in \mathbb{R}_+$ *and batch size* $b \in \mathbb{N}$ *is a synchronization operator* $\sigma_{\Delta,b}$ *such that in round* $t \in \mathbb{N}$ *it holds that* $\sigma_{\Delta,b}(\mathbf{f}_t) = \mathbf{f}_t$ *if* $t \mod b \neq 0$ *and otherwise: (i)* $\mathfrak{a}(\mathbf{f}_t) = \mathfrak{a}(\sigma_{\Delta,b}(\mathbf{f}_t))$*, i.e., it leaves the aggregated model invariant, and (ii)* $\delta(\sigma_{\Delta,b}(\mathbf{f}_t)) \leq \Delta$*, i.e., after its application the model divergence is bounded by* Δ*.*

Note that the divergence threshold $\Delta \in \mathbb{R}_+$ can be chosen fixed, or variable, e.g., $\Delta_t = {}^1\!/_t$. This partial synchronization is a relaxation of the synchronization operator in two ways: (i) it allows to schedule aggregation in a data-dependent way and (ii) it allows to aggregate only a subset of the models. In particular it allows to leave all models untouched as long as the divergence remains below the threshold Δ.

To efficiently reduce communication, both relaxations are important. For example, the Federated Averaging (FedAvg) approach by McMahan et al. (2017) tackles communication-efficiency by averaging only a random fraction of the models. However, this averaging is scheduled periodically, thus reducing communication only by a fixed fraction. A partial synchronization operator instead allows to adjust communication both through dynamic scheduling and partial aggregation. This is the basis for our dynamic averaging protocol.

Every distributed learning algorithm that implements a partial synchronization operator has to implicitly control the divergence of the model configuration. However, the divergence cannot simply be computed by centralizing all local models, because this would require continuous communication.

A strategy to overcome this problem is to first decompose the global condition $\delta(\mathbf{f}) \leq \Delta$ into a set of local conditions that can be monitored at their respective learners without communication (see, e.g., Gabel et al. (2014); Giatrakos et al. (2012); Sharfman et al. (2007) for a more general description of this method). Then, a resolution protocol is defined that transfers the system back into a valid state whenever one or more of these local conditions are violated. This includes carrying out a sufficient amount of synchronization to reduce the divergence to be less or equal to Δ.

For deriving local conditions the domain of the divergence function is considered restricted to an individual model. Here, a condition is identified such that the global divergence can not cross the Δ-threshold as long as all local models satisfy that condition. Since this condition defines a subset of the input space to the divergence, it is referred to as a **safe-zone** (similar to, e.g., Keren et al. (2012, 2014); Lazerson et al. (2015); Sharfman et al. (2008)). Note that a direct distribution of the threshold across the local learners (e.g., setting the local thresholds to Δ/m as in Keralapura et al. (2006)) is infeasible, because the divergence function is non-linear.

For certain aggregation operators, a generic set of local conditions can be constructed. For these operators, the aggregate is a centroid (Nielsen and Nock, 2009) of the model configuration with respect to the distance used in the divergence. This is characterized as follows.

Definition 3.4. *Let $\mathfrak{a}$ be an aggregation operator and d the distance function over $\mathcal{F}$ used to compute the divergence as in Definition 3.2. Then $\mathfrak{a}$ is **central** if for all $\mathbf{f} = (f^1, \ldots, f^m) \in \mathcal{F}^m$ it holds that*

$$\mathfrak{a}(\mathbf{f}) = \arg\min_{f' \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} d\left(f', f^i\right) \quad .$$

If the model space has a representation in the Euclidean space, then a natural distance function is the squared Euclidean distance $d(f, f') = \|f - f'\|_2^2$. The minimizer of this is the average, i.e.,

$$\overline{\mathbf{f}} = \frac{1}{m} \sum_{i=1}^{m} f^i = \arg\min_{f' \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \left\|f' - f^i\right\|_2^2 \quad .$$

If the standard Euclidean distance $d(f, f') = \|f - f'\|$ is used, the minimizer is the geometric median (Minsker et al., 2015).

For central aggregation operators, the distance of each local model to the actual aggregate is always smaller than to any common reference point. With this, the following simple set of local conditions can be given.

Proposition 3.5. *Let $\mathbf{f} = (f^1, \ldots, f^m) \in \mathcal{F}^m$ be a model configuration and $r \in \mathcal{F}$ some **reference model**. Let $\mathfrak{a}$ be a central aggregation operator and d a distance function over $\mathcal{F}$. If for all $i \in [m]$ the **local condition** $d\left(f^i, r\right) \leq \Delta$ holds, then the global divergence is bounded by Δ, i.e.,*

$$\frac{1}{m} \sum_{i=1}^{m} d\left(f^i, \mathfrak{a}(\mathbf{f})\right) \leq \Delta \quad .$$

Proof. The theorem follows directly from the centrality of the aggregation operator, i.e., the fact that $\mathfrak{a}(\mathbf{f})$ minimizes the distances to all f^i, i.e.,

$$\frac{1}{m}\sum_{i=1}^{m} d\left(f^i, \mathfrak{a}(\mathbf{f})\right) \leq \frac{1}{m}\sum_{i=1}^{m} d\left(f^i, r\right) \leq \Delta$$

□

For the local conditions to hold it has to be guaranteed that at any times all learners use the same reference model r. The closer the reference model is to the true aggregate of all local models, the tighter are the local conditions.

3.2.2. Dynamic Averaging

A natural choice for the aggregation operator is the average $\mathfrak{a}_{AVG}$, since it is central as in Definition 3.4. The corresponding distance is the squared Euclidean distance $d(f, f') = \|f - f'\|_2^2$. Averaging also adheres to the first condition in Definition 3.3: averaging a subset of models leaves the global average unchanged so that $\mathfrak{a}_{AVG}(\mathbf{f}) = \mathfrak{a}_{AVG}(\sigma_{\Delta,b}(\mathbf{f}))$. The corresponding local conditions for learner $i \in [m]$ are given by

$$\|f^i - r\|_2^2 \leq \Delta \enspace .$$

Thus in the following, the norm $\|\cdot\|$ refers to the Euclidean norm.

In order to craft a partial synchronization operator, also the second condition of Definition 3.3 needs to be fulfilled, i.e., after synchronization the divergence is smaller than Δ. For that, it remains to design a resolution protocol that specifies how to react when one or several of the local conditions are violated. A direct solution is to average all local models and update the reference model on each violation, denoted a **full synchronization**. Then the divergence is 0 and the second condition holds. The first choice for the reference model then is the average model from the last full synchronization step, which is the same for all local learners. This approach, however, does not scale well with the number of learners in cases where model updates have a non-zero probability even in the asymptotic regime of the learning process. When well-performing models for the current target distribution are present at all local learners, the probability of an individual local violation is very low. However, the probability of having a violation in a round increases exponentially with the number of learners. That is, let p denote the probability of a single learner having a violation and $(1 - p)$ to not have one. Then the probability of all $m \in \mathbb{N}$ not having a violation is $(1 - p)^m$.

A more elaborate approach that scales well with the number of learners is to perform a local balancing procedure: On a violation, the learner tries to balance it by incrementally querying other learners for their models. If the mean of all received models lies within the safe zone, it is transferred back as new model to all participating learners, and the resolution is finished. If all learners have been queried, the result is equivalent to a full synchronization and the reference vector is updated. In both cases, the divergence of the model configuration is bounded by Δ at the end of the balancing process, because all local conditions hold. Thus, the second condition holds. Since partial averaging leaves the global average model unchanged, the first condition holds, as well. Hence, this approach is complying to Definition 3.3. The corresponding synchronization operator is denoted **dynamic averaging** $\overline{\sigma}_{\Delta,b}$.

Algorithm 4 Dynamic Averaging Protocol

```
Input: learning algorithm $\mathcal{A}$, divergence threshold $\Delta$, parameter $b$, $m$ learners
Initialization:
  local models $f_1^1, \dots, f_1^m \leftarrow$ one random $f$
  reference vector $r \leftarrow f$
  violation counter $v \leftarrow 0$
Round $t$ at learner $i$:
  observe $E_t^i \subset \mathcal{X} \times \mathcal{Y}$
  update $f_{t-1}^i$ using the learning algorithm $\mathcal{A}$
  if $t \mod b = 0$ and $\|f_t^i - r\|^2 > \Delta$ then
    send $f_t^i$ to coordinator (violation)
  end if
At coordinator on violation:
  let $\mathcal{B}$ be the set of learners with violation
  $v \leftarrow v + |\mathcal{B}|$
  if $v = m$ then $\mathcal{B} \leftarrow [m]$, $v \leftarrow 0$
  while $\mathcal{B} \neq [m]$ and $\left\|\frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} f_t^i - r\right\|^2 > \Delta$ do
    augment $\mathcal{B}$ by augmentation strategy
    receive models from learners added to $\mathcal{B}$
  end while
  send model $\overline{\mathbf{f}} = \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} f_t^i$ to learners in $\mathcal{B}$
  if $\mathcal{B} = [m]$ also set new reference vector $r \leftarrow \overline{\mathbf{f}}$
```

While balancing can achieve a high communication reduction over full synchronizations, particularly for a large number of learners it potentially degenerates in certain special situations. One can end up in a stable regime in which local violations are likely to be balanced by a subset of the learners; however a full synchronization would strongly reduce the expected number of violations in future rounds. In other words: balancing can delay crucial reference point updates indefinitely. A simple hedging mechanism can be employed in order to avoid this situation: The number of local violations while using the current reference model are counted and a full synchronization is triggered whenever this number exceeds the total number of learners. This concludes our dynamic averaging protocol $\mathcal{D} = (\mathcal{A}, \overline{\sigma}_{\Delta,b}, \mathfrak{a}_{AVG}, m)$. All components are summarized in Algorithm 4.

3.2.3. Averaging of Kernel Models and Neural Networks

The dynamic averaging protocol can be readily applied to learning linear models, i.e., where $\mathcal{F} = \mathbb{R}^d$ for some model dimension $d \in \mathbb{N}$, and $\mathfrak{a}_{AVG}$ is the standard vector average. In the following it is described how dynamic averaging can be applied to kernel models and neural networks.

Kernel Models

In order to apply dynamic averaging to kernel models it is necessary to define their average and distance. Recall that for kernel models, the model space is a reproducing kernel Hilbert space

$$\mathcal{H}_k = \{f\colon \mathcal{X} \to \mathbb{R} | f(\cdot) = \sum_{j=1}^{\dim F} w_j \Phi_j(\cdot)\}$$

with kernel function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, feature space F, and a mapping $\Phi\colon \mathcal{X} \to F$ into the feature space. As described in Section 2.1.1, the kernel function corresponds to an inner product of input points mapped into feature space, i.e.,

$$k(x, x') = \sum_{j=1}^{\dim F} \xi_j \Phi_j(x)^\top \Phi_j(x') \tag{3.6}$$

for constants $\xi_1, \xi_2, \dots \in \mathbb{R}_+$. The model can be expressed in its support vector expansion, or dual representation

$$f(\cdot) = \sum_{x \in S} \alpha_x k(x, \cdot)$$

with a set of support vectors $S = \{x_1, \dots, x_{|S|}\} \subset \mathcal{X}$ and corresponding coefficients $\alpha_x \in \mathbb{R}$ for all $x \in S$. This implies that the linear weights $w = (w_1, w_2, \dots) \in \mathbb{R}^{\dim F}$ defining f are given implicitly by

$$w_i = \sum_{x \in S} \xi_i \alpha_x \Phi_i(x) \ .$$

The following defines the average of kernel models. For that, let $\mathbf{f} = \left(f^1, \dots, f^m\right) \subset \mathcal{H}_k$ be a model configuration with corresponding weight vectors $\left(w^1, \dots, w^m\right) \subset F$, where each model $i \in [m]$ has support vectors $S^i = \{x_1^i, \dots, x_{|S^i|}^i\} \subset \mathcal{X}$ and coefficients α_x^i for all $x \in S^i$. The average is given by

$$\overline{\mathbf{f}}(\cdot) = \frac{1}{m}\sum_{i=1}^m f^i(\cdot) = \frac{1}{m}\sum_{i=1}^m \sum_{j=1}^{\dim F} w_j^i \Phi_j(\cdot) = \frac{1}{m}\sum_{i=1}^m \sum_{j=1}^{\dim F} \sum_{x \in S^i} \xi_j \alpha_x^i \Phi_j(x)^\top \Phi_j(\cdot) \ .$$

Using the definition of the kernel function (Equation 3.6), the above equation can be simplified to

$$\overline{\mathbf{f}}(\cdot) = \frac{1}{m}\sum_{i=1}^m \sum_{x \in S^i} \alpha_x^i k(x, \cdot) \ .$$

Using the union of support vectors $\overline{S} = \bigcup_{i \in [m]} S^i = \{s_1, \dots, s_{|\overline{S}|}\}$ and augmented coefficients $\overline{\alpha}_s^i \in \mathbb{R}$, which are given by

$$\overline{\alpha}_s^i = \begin{cases} \alpha_x^i, & \text{if } x = s \\ 0, & \text{otherwise} \end{cases} ,$$

the dual representation of the average directly follows.

Proposition 3.7. *For a model configuration* $\mathbf{f} = \left(f^1, \ldots, f^m\right) \subset \mathcal{H}_k$, *where each model* $i \in [m]$ *has augmented coefficients* $\overline{\alpha}_s^i$ *for* $s \in \overline{S}$, *the average* $\overline{\mathbf{f}} \in \mathcal{H}$ *is given by*

$$\overline{\mathbf{f}}(\cdot) = \sum_{s \in \overline{S}} \left(\frac{1}{m} \sum_{i=1}^{m} \overline{\alpha}_s^i \right) k(s, \cdot) \ ,$$

with support vectors $\overline{S}$ *and coefficients*

$$\overline{\alpha}_s = \frac{1}{m} \sum_{i=1}^{m} \overline{\alpha}_s^i$$

for all $s \in \overline{S}$.

The inner product in $\mathcal{H}_k$ induces a norm $\|f\|_{\mathcal{H}_k}^2 = \langle f, f \rangle_{\mathcal{H}_k} = \sum_{x \in S} \left(\alpha_x\right)^2 k(x,x)$ for $f \in \mathcal{H}$ with support vectors $S \subset \mathcal{X}$ and corresponding weights $\alpha_x \in \mathbb{R}$ for $x \in S$. Thus, using he definition of the average, the distance between an individual model f^i from a model configuration $\mathbf{f}$ and the average $\overline{\mathbf{f}}$ is given by $\left\|f^i - \overline{\mathbf{f}}\right\|_{\mathcal{H}_k}^2 = \left\langle f^i, f^i \right\rangle + \left\langle \overline{\mathbf{f}}, \overline{\mathbf{f}} \right\rangle - 2 \left\langle f^i, \overline{\mathbf{f}} \right\rangle$, i.e.,

$$\left\|f^i - \overline{\mathbf{f}}\right\|_{\mathcal{H}_k}^2 = \sum_{x \in S^i} \left(\alpha_x^i\right)^2 k(x,x) + \sum_{s \in \overline{S}} \left(\overline{\alpha}_s\right)^2 k(s,s) - 2 \sum_{x \in S^i} \sum_{s \in \overline{S}} \alpha_x^i \overline{\alpha}_s k(x,s) \ .$$

Using this distance, we can compute the divergence for models from a reproducing kernel Hilbert space as in Definition 3.2 and the local conditions as in Proposition 3.5.

Neural Networks

In order to define the average and divergence for neural networks, recall from Section 2.1.1 that a neural network can be represented by a directed weighted graph $G = (V, E)$, also referred to as the **architecture**. The vertices in V represent the neurons, the edges in E the connections, where each connection between two neurons is associated with a weight.

Given an architecture, i.e., a graph $G = (V, E)$, the training of a neural network is performed by adapting the edge weights. These weights can be represented by a vector $w \in \mathbb{R}^{|E|}$. If the architecture G is equal for all local learners, we can represent each model $f \in \mathcal{F}$ by its corresponding weight vector $w \in \mathbb{R}^{|E|}$. Then the average and distance between models is the average and Euclidean distance of their corresponding weight vectors, similar to linear models.

However, in contrast to linear models, the loss function is non-convex in the weight vector—even a single neuron with logistic activation can lead to exponentially many local minima of the squared loss (Auer et al., 1996). For non-convex objectives, a particular problem is that the average of a set of models can have a worse performance than any model in the set (see Figure 3.3 for an illustration).

In the context of deep learning, averaging models was introduced by McMahan et al. (2017). While in general, averaging neural networks can be detrimental (see Figure 3.3(c)), they found that if all local models are initialized to the same starting model, averaging often works in practice. An explanation could be that models with a common initialization remain in the vicinity of the same local minimum, despite using different training data. Since in this case the local models would remain in a locally convex environment, averaging would work similar to the standard convex case. Chapter 4 analyzes this in more detail.

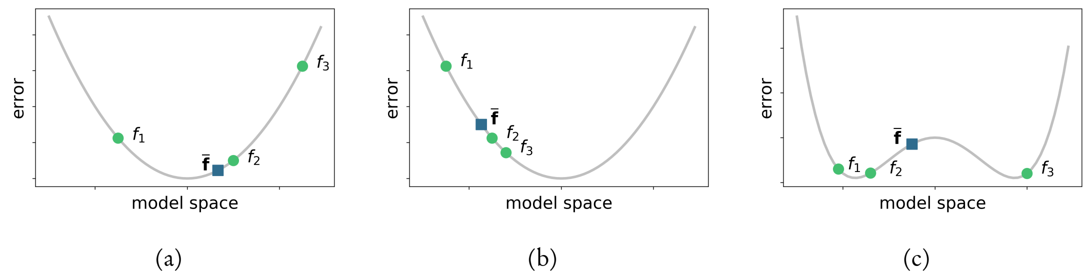


Figure 3.3.: (a) Illustration of an exemplary convex error surface. The x-axis represents the model space, i.e., the space of possible model parameters. The y-axis represents the expected error, or quality, of the respective model. The average $\bar{\mathbf{f}}$ of a set of models $f_1, f_2, f_3 \in \mathcal{F}$ has a lower expected error—and therefore a better quality—than the individual models. (b) In general, the average does not have better quality than all individual models. However, for a convex error surface the average is always at least as good as one of the models in the set. (c) In case of a non-convex error surface, the average can be arbitrarily bad. In particular, if the individual models are scattered around multiple local minima, the average may lie on a local maximum between the minima.

After having shown how to apply dynamic averaging to machine learning algorithms using linear models, kernel methods, and neural networks, the following sections theoretically analyze dynamic averaging in terms of online regret, communication, and speedup.

3.3. Efficiency of Dynamic Averaging

This section aims at analyzing the efficiency of dynamic averaging and its speedup. That is, in which cases it is at the same time (i) consistent, i.e., retains the loss bound of the serial application of the base learning algorithm, and (ii) adaptive, i.e., the communication is tied to the loss (see Definition 3.1). To that end, first the online regret of dynamic averaging is bounded, followed by a bound on the communication.

In order to bound the online regret of dynamic averaging, I consider a class of incremental base learning algorithms that perform regret-proportional convex updates, which is defined in the following.

3.3.1. Regret-Proportional Convex Updates

In principle, dynamic averaging can be applied to a wide range of incremental machine learning algorithms (e.g., SGD, passive aggressive (PA) (Crammer et al., 2006), ADAM (Kingma and Ba, 2014), RMSPROP (Mukkamala and Hein, 2017), or regularized dual averaging (Xiao, 2010)). For the formal results in this thesis, however, some general properties are required.

The first property bounds the distance between two models after they have been updated with the same examples. Since most incremental learning algorithms update a model in the direction of a convex set depending only on the training examples, e.g., in the direction of the training example itself in case of SGD and PA updates, we can bound the distance after updates using properties of orthogonal projections. For $\mathcal{F}$ with norm $\|\cdot\|$, the orthogonal projection $\Pi_E(f)$ of a point $f \in \mathcal{F}$ onto a convex set $\Gamma_E \subseteq \mathcal{F}$ is defined as

$$\Pi_E(f) = \arg\min_{g \in \Gamma_E} \|f - g\| \enspace .$$

This first property is formalized in the following definition.

Definition 3.8. *The update performed by an incremental learning algorithm $\mathcal{A}$ is a* ***convex update*** *for a loss function ℓ if for all $f \in \mathcal{F}$ and $E \subset \mathcal{X} \times \mathcal{Y}$ there is a closed convex set $\Gamma_E \subseteq \mathcal{F}, \Gamma_E \neq \emptyset$, and $\tau_E \in (0,1]$ such that*

$$\mathcal{A}(E, f) = f + \tau_E (\Pi_E(f) - f) \enspace ,$$

where $\Pi_E(f)$ denotes the orthogonal projection of f onto Γ_E. That is, the update direction is identical to the direction of a convex projection that only depends on the training set.

The second property relates the loss value to the update magnitude. The difficulty here is that some incremental learning algorithms might still suffer loss even if they do not update the model any more. For example, gradient-based methods have a zero gradient at the optimal model but the optimal model can still suffer loss. The updates can, however, be related to the regret suffered, i.e., the difference of the loss suffered to the loss of the optimal model. This is formalized as follows.

Definition 3.9. *The update performed by an incremental learning algorithm $\mathcal{A}$ is a* ***regret-proportional update*** *for a loss function ℓ if there are a constant $\gamma > 0$ such that for all $f \in \mathcal{F}$ and $E \subset \mathcal{X} \times \mathcal{Y}$ it holds that the update magnitude is a true fraction of the regret incurred, i.e.,*

$$\|f - \mathcal{A}(E, f)\| \geq \gamma \left(\sum_{(x,y) \in E} \ell(f, x, y) - \ell(f^*, x, y) \right) \enspace ,$$

where $f^* = \arg\min_{f' \in \mathcal{F}} \sum_{(x,y) \in E} \ell(f', x, y)$.

Note that it trivially follows that every incremental learning algorithm with an update length proportional to the loss is also a regret-proportional update.

Corollary 3.10. *If for an incremental learning algorithm $\mathcal{A}$, a loss function ℓ, and a constant $\gamma > 0$ it holds for all $f \in \mathcal{F}$ and $E \subset \mathcal{X} \times \mathcal{Y}$ drawn iid according to a target distribution $\mathcal{D} \colon \mathcal{X} \times \mathcal{Y} \to [0,1]$ that the update magnitude is proportional to the loss, i.e.,*

$$\|f - \mathcal{A}(E, f)\| \geq \gamma \left(\sum_{(x,y) \in E} \ell(f, x, y) \right) \enspace ,$$

then the update performed by $\mathcal{A}$ is also a regret-proportional update.

Proof. Let $f^* = \arg\min_{f\in\mathcal{F}} \mathbb{E}_{(x,y)\sim\mathcal{D}}\,\ell(f,x,y)$ denote the optimal model. Since for all $(x,y)\in\mathcal{X}\times\mathcal{Y}$ it holds that $\ell(f^*,x,y)\geq 0$, it follows that $\ell(f,x,y)\geq \ell(f,x,y)-\ell(f^*,x,y)$ and thus

$$\|f-\mathcal{A}(E,f)\| \geq \gamma\left(\sum_{(x,y)\in E}\ell(f,x,y)\right) \geq \gamma\left(\sum_{(x,y)\in E}\ell(f,x,y)-\ell(f^*,x,y)\right) .$$

□

An incremental learning algorithm performs **regret-proportional convex updates** if both Definition 3.8 and Definition 3.9 hold.

The following shows that machine learning algorithms based on stochastic gradient descent (SGD), as well as those based on passive aggressive updates (Crammer et al., 2006) using linear models and kernel methods perform regret-proportional convex updates.

First, I want to show that incremental learning algorithms based on stochastic gradient descent (SGD), mini-batch SGD (Dekel et al., 2012), and gradient descent (GD) using linear models and kernel models perform regret-proportional convex updates. To see that, recall that these learning algorithms in each round $t\in\mathbb{N}$ perform updates of the form

$$f_{t+1} = f_t - \eta \sum_{(x,y)\in E_t} \nabla_{f_t}\ell(f_t,x,y)$$

for a dataset $E_t\subset\mathcal{X}\times\mathcal{Y}$ and a positive learning rate $\eta>0$ (for SGD, $|E_t|=1$, for GD, $E_t=E$).

For loss functions that depend on the prediction score (for linear models this is the inner product of a linear model with the instance $\langle f,x\rangle$) the derivative of the loss function points in the direction of x, or $-x$. Examples of such loss functions are the **hinge-loss** $\ell(f,x,y)=\max\{1-yf(x),0\}$, the **squared loss** $\ell(f,x,y)=1/2\|f(x)-y\|^2$, and the **$\epsilon$-insensitive loss** $\ell(f,x,y)=\max(|f(x)-y|-\epsilon,0)$. For example, if one uses the hinge loss for classification with a linear model, i.e., $\mathcal{Y}=\{-1,1\}$ and $\ell(f,x,y)=\max\{1-y\langle f,x\rangle,0\}$, we have $\nabla_f\ell(f,x,y)=-yx$. Thus, the convex set $\Gamma_{\{x,y\}}$ is defined by the half-space $\Gamma=\{w\in\mathcal{F}|\langle w,x\rangle y\geq 0\}$. In case $|E|>1$, Γ_E is defined by the intersection of the $\Gamma_{\{x,y\}}$ for all $(x,y)\in E$. This also holds for kernel methods, but here the convex set is in the kernel Hilbert space. Thus, Definition 3.8 is fulfilled. Note that for projected SGD

$$f_{t+1} = \Pi_W\left(f_t - \eta \sum_{(x,y)\in E_t} \nabla_{f_t}\ell(f_t,x,y)\right)$$

with a feasible set $W\subset\mathcal{X}$ the update is either in the direction of x, if the resulting model is in the feasible set W, or in the direction of the feasible set W, which is typically convex. Thus also for projected SGD with a convex feasible set Definition 3.8 is fulfilled.

Let $\rho\in\mathbb{R}$ be the data radius, i.e., for all $x\in\mathcal{X}$ it holds that $\|x\|\leq\rho$. Then these learning algorithms perform regret-proportional updates with $\gamma=\eta\rho/1+\rho^2$. The same holds for regression using squared loss, ϵ-insensitive loss. These results are summarized in the following corollary.

Corollary 3.11. *Stochastic gradient descent (SGD), mini-batch SGD, and gradient descent (GD) using the hinge loss, squared loss, and ϵ-insensitive loss perform regret-proportional convex updates.*

The same can be checked for many other of loss functions and many variants of SGD.

Another example for regret-proportional convex updates is the passive aggressive algorithm (Crammer et al., 2006). Recall from Equation 2.9 that it is defined for a variety of learning tasks including classification, regression, and uni-class prediction and can be uniformly described by

$$\mathcal{A}(f,(x,y)) = \arg\min_{f' \in \mathcal{F}} \frac{1}{2}\|f - f'\|^2 \;\; s.t. \;\; \ell(f',x,y) = 0$$

where for classification ℓ is the hinge loss, for regression the ϵ-insensitive loss, and for uni-class prediction (where no x is observed and $y = \mathcal{F}$) the uni-class loss is given by $\ell(f,y) = \max(|f-y| - \epsilon, 0)$. It can be observed immediately that for linear models and kernel methods, in all three cases these update rules are an actual projection on the convex set $\Gamma_{\{x,y\}} = \{f \in \mathcal{F} : \ell(f,x,y) = 0\}$, which corresponds to a half-space, a 2ϵ-strip, and an ϵ-ball, respectively. Hence, Definition 3.8 follows immediately with $\tau_{x,y} = 1$. Definition 3.9 can then be verified from the closed form solution of Eq. 2.9, which in case of classification is given by

$$\mathcal{A}(f,(x,y)) = f + \frac{\ell(f,x,y)}{\|x\|^2} yx \;\; .$$

Given a data radius $\rho > 0$, the update magnitude can be bounded from below by $\|f - \varphi(f,x,y)\| \geq \rho^{-1}\ell(f,x,y)$, and with Lemma 3.10 the definition holds with $\gamma = \rho^{-1}$. The other cases follow similarly. Crammer et al. (2006) also give other variants of passive aggressive updates that have a reduced learning rate determined by an aggressiveness parameter $C > 0$. These rules also satisfy the conditions of Definition 3.9. For example the rule for classification then becomes

$$\mathcal{A}(f,(x,y)) = f + \frac{\ell(f,x,y)}{\|x\|^2 + \frac{1}{2C}} yx \;\; .$$

Using $\|x\| \in [1/\dim\mathcal{X}, \rho]$, one can show that this variant remains hinge-loss proportional with $\gamma = \dim\mathcal{X}^{-1}(\rho^2 + 1/(2C))^{-1}$, and the update direction is identical to the same convex projection as in the standard case. Again, these results are summarized in the following corollary.

Corollary 3.12. *Passive aggressive updates (PA) and regularized passive aggressive updates (PA-I) with hingle loss, ϵ-insensitive, and uni-class loss perform regret-proportional convex updates.*

Using these properties of regret-proportional convex updates, the following section bounds the online regret of dynamic averaging in relation to that of periodic averaging.

3.3.2. Relating the Online Regret of Dynamic to Periodic Averaging

In the following, the impact on the learning performance of using dynamic averaging instead of periodic averaging is analyzed. Recall that periodic averaging uses the synchronization operator

$$\sigma_b(f^1,\ldots,f^m) = \begin{cases} \left(\overline{\mathbf{f}}_t,\ldots,\overline{\mathbf{f}}_t\right), & \text{if } b \mid t \\ \mathbf{f}_t = (f_t^1,\ldots,f_t^m), & \text{otherwise} \end{cases} \;\; .$$

Thus, every $b \in \mathbb{N}$ rounds, it replaces all local models by their joint average (for more details, see the example in Section 2.3.3).

In order to relate the convergence of dynamic averaging to that of periodic averaging, the following **synchronization lemma** bounds the difference in each round $t \in \mathbb{N}$ between the model configuration $\mathbf{d}_t = \left(d_t^1, \ldots, d_t^m\right)$ maintained by dynamic averaging and the model configuration $\mathbf{s}_t = \left(s_t^1, \ldots, s_t^m\right)$ maintained by periodic averaging. Note that the result is given for a variable divergence threshold Δ_t which includes the fixed threshold setting, i.e., $\Delta_t = \Delta$ for all $t \in \mathbb{N}$.

Lemma 3.13. *Let* $\mathbf{d}_t = (d_t^1, \ldots, d_t^m) \in \mathcal{F}^m$ *be a model configuration in round* $t \in \mathbb{N}$ *maintained by dynamic averaging with parameters* $\Delta_t \in \mathbb{R}$ *and* $b \in \mathbb{N}$ *in round* $t \in \mathbb{N}$ *and* $\mathbf{s}_t = (s_t^1, \ldots, s_t^m) \in \mathcal{F}^m$ *a model configuration maintained by periodic averaging with parameter* b*. Then it holds that*

$$\frac{1}{m}\sum_{i=1}^{m} \|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)^i - \sigma_b(\mathbf{s}_t)^i\|^2 \leq \frac{1}{m}\sum_{i=1}^{m} \|d_t^i - s_t^i\|^2 + \Delta_t \ .$$

Proof. We consider the case $b \mid t$ (i.e, $t \mod b = 0$), otherwise the claim follows immediately. Expressing the pairwise squared distances via the difference to $\overline{\mathbf{d}_t}$ and using the definitions of σ_b and $\overline{\sigma}_{\Delta,b}$ we can bound

$$\begin{aligned}
&\frac{1}{m}\sum_{i=1}^{m} \left\|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)_l - \sigma(\mathbf{s}_t)_l\right\|^2 = \frac{1}{m}\sum_{i=1}^{m} \left\|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)_l - \overline{\mathbf{d}_t} + \overline{\mathbf{d}_t} - \overline{\mathbf{s}_t}\right\|^2 \\
&= \underbrace{\frac{1}{m}\sum_{i=1}^{m} \left\|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)_l - \overline{\mathbf{d}_t}\right\|^2}_{\leq \Delta_t, \text{ by (ii) of Def. 3.3}} + 2\left\langle \underbrace{\frac{1}{m}\sum_{i=1}^{m} \overline{\sigma}_{\Delta,b}(\mathbf{d}_t)_l - \overline{\mathbf{d}_t}}_{=0, \text{ by (i) of Def. 3.3}}, \overline{\mathbf{d}_t} - \overline{\mathbf{s}_t}\right\rangle + \left\|\overline{\mathbf{d}_t} - \overline{\mathbf{s}_t}\right\|^2 \\
&\leq \Delta_t + \left\|\frac{1}{m}\sum_{i=1}^{m}(d_t^i - s_t^i)\right\|^2 \leq \Delta_t + \frac{1}{m}\sum_{i=1}^{m} \left\|d_t^i - s_t^i\right\|^2 \ .
\end{aligned}$$

The last inequality follows from Jensen's inequality. □

While the synchronization lemma bounds the increase in distance from each synchronization, it needs to be shown that this increase in distance cannot separate model configurations too far during the learning process. For that, the following **update lemma** shows that regret-proportional convex updates are contractions and that they reduce the distance between a pair of models proportional to their loss difference.

Lemma 3.14. *Let the updates of an incremental learning algorithm* $\mathcal{A}$ *be regret-proportional convex updates with* $\gamma > 0$*. Then for all models* $d, s \in \mathcal{F}$ *and all examples* $(x, y) \in \mathcal{X} \times \mathcal{Y}$ *it holds that*

$$\|\mathcal{A}(d, x, y) - \mathcal{A}(s, x, y)\|^2 \leq \|d - s\|^2 - \gamma^2 \left(\ell(d, x, y) - \ell(s, x, y)\right)^2 .$$

Proof. For $f \in \mathcal{F}$ we write $\Pi_{(x,y)}(f) = \Pi(f)$ for the projection of f on $\Gamma_{(x,y)}$ and $f' = \mathcal{A}(f,x,y)$. Since $\Pi(\cdot)$ is a projection on a convex set, it holds for all $v, w \in \mathcal{F}$ that

$$\|\Pi(v) - \Pi(w)\|^2 \leq \|v - w\|^2 - \|v - \Pi(v) - w + \Pi(w)\|^2$$

(see Proposition A.1 in the Appendix Section B). Applying this to the models d, s gives

$$\|\Pi(d) - \Pi(s)\|^2 \leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\|^2 \ . \tag{3.15}$$

Applying it to the updated models d', s' yields

$$\|\Pi(d') - \Pi(s')\|^2 \leq \|d' - s'\|^2 - \|d' - \Pi(d') - s' + \Pi(s')\|^2$$

and, since $f' = \tau_{(x,y)}\Pi_E(f) + (1 - \tau_{(x,y)})f$ by (ii) of the definition of regret-proportional convex updates, the idempotence of $\Pi(\cdot)$ implies that $\Pi(f) = \Pi(f')$, this also yields

$$\|\Pi(d) - \Pi(s)\|^2 \leq \|d' - s'\|^2 - \|d' - \Pi(d) - s' + \Pi(s)\|^2 \tag{3.16}$$

By Proposition A.2 in Appendix Section B it holds that

$$\|d' - s'\|^2 - \|d' - \Pi(d) - s' + \Pi(s)\|^2 \leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\|^2 \ ,$$

and it is possible to subtract Eq. 3.16 from Eq. 3.15 to obtain

$$\begin{aligned} 0 &\leq \|d - s\|^2 - \|d' - s'\|^2 - \|d - \Pi(d) - s + \Pi(s)\|^2 + \|d' - \Pi(d) - s' + \Pi(s)\|^2 \\ \Leftrightarrow \|d' - s'\|^2 &\leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\|^2 + \|d' - \Pi(d) - s' + \Pi(s)\|^2 \ . \end{aligned} \tag{3.17}$$

Using $f' = f + \tau_{(x,y)}(\Pi(f) - f)$ for both d' and s' gives

$$\|d' - \Pi(d) - s' + \Pi(s)\|^2 \leq (1 - \tau_{(x,y)})^2 \|(d - \Pi(d)) - s + \Pi(s)\|^2 \ .$$

Inserting this into Equation 3.17 yields

$$\begin{aligned} \|d' - s'\|^2 \leq& \|d - s\|^2 - \|(d - \Pi(d)) - s + \Pi(s)\|^2 \\ &+ (1 - \tau_{(x,y)})^2 \|(d - \Pi(d)) - s + \Pi(s)\|^2 \\ \leq& \|d - s\|^2 - \tau_{(x,y)} \left(\|d - \Pi(d)\| - \|s - \Pi(s)\|\right)^2 \ . \end{aligned} \tag{3.18}$$

This shows that the distance between models is reduced by an update proportional to the update magnitude. It remains to relate the update magnitude to the loss. For that, observe that it follows from condition (i) of the definition of regret-proportionality that

$$\|f - \Pi(f)\| = \frac{1}{\tau_{(x,y)}} \|f - (f + \tau_{(x,y)}(\Pi(f) - f))\| = \frac{\|f - f'\|}{\tau_{(x,y)}} \geq \frac{\gamma}{\tau_{(x,y)}} \ell(f,x,y) \ .$$

Since $\tau_{(x,y)} \in (0,1]$ this yields

$$\|f - \Pi(f)\| \geq \gamma \ell(f,x,y) \ .$$

Finally, inserting this into Equation 3.18 yields the claim

$$\|d' - s'\|^2 \leq \|d - s\|^2 - \gamma^2 \left(\ell(d,x,y) - \ell(s,x,y)\right)^2 \ .$$

□

From the two lemmas above we see that, while each synchronization increases the distance between the static and the dynamic model by at most Δ, with each update step, the distance is decreased proportional to the loss difference. Using this, we can bound the loss of our protocol. For that, the following abbreviations are used. For a model f_t in round $t \in \mathbb{N}$ the application of $\mathcal{A}$ is $\mathcal{A}(f_t) = \mathcal{A}(f_t, x_t, y_t)$. Furthermore, for a model configuration $\mathbf{f}_t$ the abbreviation $\mathcal{A}(\mathbf{f}_t)$ is used for $(\mathcal{A}(f_t^1), \dots, \mathcal{A}(f_t^m))$. Moreover, let $\mathbf{d}_1, \dots \mathbf{d}_T$ and $\mathbf{s}_1, \dots, \mathbf{s}_T$ be two sequences of model configurations such that $\mathbf{d}_1 = \mathbf{s}_1$ and the sequence $\mathbf{d}_t$ is maintained by dynamic averaging $\mathcal{D} = (\mathcal{A}, \overline{\sigma}_{\Delta,b}, \mathfrak{a}_{AVG}, m)$ with $\Delta \in \mathbb{R}_+$ and $b \in \mathbb{N}$ and the sequence $\mathbf{s}_t$ is maintained by periodic averaging $\mathcal{P} = (\mathcal{A}, \sigma_b, \mathfrak{a}_{AVG}, m)$ with the same b. That is, for $t = 1, \dots, T$ the sequence is defined by $\mathbf{d}_{t+1} = \overline{\sigma}_{\Delta,b}(\mathcal{A}(\mathbf{d}_t))$, and $\mathbf{s}_{t+1} = \sigma_b(\mathcal{A}(\mathbf{s}_t))$ respectively.

Theorem 3.19. *Let $\mathcal{A}$ be an online learning algorithm that performs regret-proportional convex updates with$\gamma > 0$. For $m \in \mathbb{N}$ learners, let $\mathbf{d}_1, \dots \mathbf{d}_T$ and $\mathbf{s}_1, \dots, \mathbf{s}_T$ be two sequences of model configurations with $\mathbf{d}_1 = \mathbf{s}_1$ and $\mathbf{d}_t$ is maintained by dynamic averaging $\mathcal{D} = (\mathcal{A}, \overline{\sigma}_{\Delta,b}, \mathfrak{a}_{AVG}, m)$ with $\Delta \in \mathbb{R}_+$ and $b \in \mathbb{N}$ and $\mathbf{s}_t$ is maintained by periodic averaging $\mathcal{P} = (\mathcal{A}, \sigma_b, \mathfrak{a}_{AVG}, m)$ with the same b. Then it holds that*

$$L_{\mathcal{D}}(T,m) \le L_{\mathcal{P}}(T,m) + \frac{T}{b\gamma^2}\Delta \ .$$

Proof. Combining the synchronization lemma (Lemma 3.13) with the update lemma (Lemma 3.14) yields for all $t \in [T]$ that

$$\sum_{i=1}^m \left\| d_{t+1}^i - s_{t+1}^i \right\|^2 \le \sum_{i=1}^m \left\| d_t^i - s_t^i \right\|^2 - \gamma^2 \sum_{i=1}^m \left(\ell(d_t^i) - \ell(s_t^i) \right)^2 + \Delta \ .$$

By applying this inequality recursively for $t = 1, \dots, T$ it follows that

$$\sum_{i=1}^m \left\| d_{T+1}^i - s_{T+1}^i \right\|^2 \le \sum_{i=1}^m \left\| d_1^i - p_1^i \right\|^2 + \left\lfloor \frac{T}{b} \right\rfloor \Delta - \gamma^2 \sum_{t=1}^T \sum_{i=1}^m \left(\ell(d_t^i) - \ell(s_t^i) \right)^2 .$$

Using $\mathbf{d}_1 = \mathbf{s}_1$, we conclude that

$$\sum_{t=1}^T \sum_{i=1}^m \left(\ell(d_t^i) - \ell(s_t^i) \right)^2 \le \frac{1}{\gamma^2} \left(\left\lfloor \frac{T}{b} \right\rfloor \Delta - \underbrace{\sum_{i=1}^m \left\| d_{T+1}^i - s_{T+1}^i \right\|^2}_{\ge 0} \right) \le \frac{1}{\gamma^2} \frac{T}{b} \Delta$$

$$\Leftrightarrow L_{\mathcal{D}}(T,m) - L_{\mathcal{P}}(T,m) \le \frac{T}{b\gamma^2}\Delta$$

□

Theorem 3.19 shows that dynamic averaging retains the regret of periodic averaging. Thus, if periodic averaging is consistent as in Definition 3.1, i.e., it retains the loss of the serial application of the base learning algorithm, then dynamic averaging is consistent, as well. In the following Section 3.3.3 it is shown that periodic averaging is consistent for SGD and mini-batch SGD.

Theorem 3.19 also implies that dynamic averaging retains the optimality of the static mini-batch algorithm of Dekel et al. (2012) for the case of stationary targets[3]: by using a time-dependent variance threshold based on $\Delta_t \in O(1/\sqrt{t})$ the bound of $O(\sqrt{T})$ follows.

Furthermore, from Theorem 3.19 it can be followed that if a shifting regret bound—as defined in Section 2.1.5—exists for periodic averaging, then this bound also applies to dynamic averaging. For that, we consider shifting regret bounds of the form

$$\mathbf{R}_{\mathcal{A}}(T,U) = c_1 \sum_{t=1}^{T}\sum_{i=1}^{m} \|u_t^i - u_{t-1}^i\|^2 + c_2 \ , \tag{3.20}$$

for a reference sequence $\mathbf{U}$ and positive constants $c_1, c_2 \in \mathbb{R}_+$ (as in Herbster and Warmuth (2001)).

Theorem 3.21. *Let the shifting regret $R_{\mathcal{P}}(T,U)$ of using periodic averaging be bounded by $\boldsymbol{R}_{\mathcal{A}}(T,U)$ as in Equation 3.20 for a reference sequence $\mathbf{U}$ and positive constants $c_1, c_2 \in \mathbb{R}_+$. Then the shifting regret of using dynamic averaging is bounded by*

$$R_{\mathcal{D}}(T,U) \le c_1 \sum_{t=1}^{T}\sum_{i=1}^{m} \|u_t^i - u_{t-1}^i\|^2 + c_2 + \frac{1}{b\gamma^2}\left(\Delta + 2\epsilon^2\right) = \boldsymbol{R}_{\mathcal{A}}(T,U) + \frac{1}{b\gamma^2}\Delta \ ,$$

Proof. For the proof let $\mathbf{d}_t$ and $\mathbf{s}_t$ denote the sequence of model configurations produced by $\overline{\sigma}_{\Delta,b}$ and σ_b, respectively. Abbreviating $\ell(f_t, x_t, y_t)$ as $\ell(f_t)$ and using the definition of shifting regret yields

$$\begin{aligned}
R_{\mathcal{D}}(T,U) &= \frac{1}{T}\sum_{t=1}^{T}\frac{1}{m}\sum_{i=1}^{m}(\ell(d_t^i) - \ell(u_t^i))^2 \\
&= \frac{1}{T}\sum_{t=1}^{T}\frac{1}{m}\sum_{i=1}^{m}((\ell(d_t^i) - \ell(s_t^i)) + (\ell(s_t^i) - \ell(u_t^i)))^2 \\
&\overset{Thm.3.19}{\overbrace{\le}} \frac{1}{b\gamma^2}\Delta + \frac{1}{T}\sum_{t=1}^{T}\frac{1}{m}\sum_{i=1}^{m}(\ell(s_t^i) - \ell(u_t^i))^2 \\
&\le \frac{1}{b\gamma^2}\Delta + c_1\sum_{t=1}^{T}\sum_{i=1}^{m}\|u_t^i - u_{t-1,l}\|_2^2 + c_2 = \frac{1}{b\gamma^2}\Delta + \mathbf{R}_{\mathcal{A}}(T,U) \ .
\end{aligned}$$

□

Intuitively, this means that the dynamic protocol only adds a constant to any shifting bound of periodic averaging.

[3] Dekel et al. (2012) consider a slightly modified version of periodic averaging which accumulates updates and then only applies them delayed at the end of a batch. However, the expected loss of eager updates (as used in periodic averaging) is bounded by the expected loss of delayed updates in the stationary setting (as used in Dekel et al. (2012)) as long as the updates reduce the distance to a loss minimizer on average (which is the case for sufficient regularization, see Zhang (2004, Eq. 5)).

3.3.3. Optimal Online Regret of Periodic Averaging

In order to proof the consistency of dynamic averaging using a learning algorithm $\mathcal{A}$, it follows from Theorem 3.19—and using a decreasing divergence threshold—that it suffices to show that periodic averaging using the same learning algorithm is consistent.

A special case of periodic averaging is the **continuous averaging protocol** $\mathcal{C} = (\mathcal{A}, \sigma_1, \mathfrak{a}_{AVG}, m)$, synchronizing every round, i.e.,

$$\sigma_1(\mathbf{f}) = \left(\overline{\mathbf{f}}, \dots, \overline{\mathbf{f}}\right) \ .$$

As base learning algorithm assume the mini-batch SGD algorithm $\mathcal{A}^{\text{mSGD}}_{B,\eta}$ (Dekel et al., 2012) with mini-batch size $B \in \mathbb{N}$ and learning rate $\eta \in \mathbb{R}$. A special case of this with $B = 1$ is the standard SGD algorithm. Recall that one step of this learning algorithm given the model $f \in \mathcal{F}$ and a dataset $E = ((x_1, y_1), \dots, (x_B, y_B)) \subset \mathcal{X} \times \mathcal{Y}$ of size B can be expressed as

$$\mathcal{A}^{\text{mSGD}}_{B,\eta}(E, f) = f - \eta \sum_{j=1}^{B} \nabla \ell(f, x_j, y_j) \ .$$

Let $\mathcal{C}^{\text{mSGD}} = (\mathcal{A}^{\text{mSGD}}_{B,\eta}, \sigma_1, \mathfrak{a}_{AVG}, m)$ denote continuous averaging with mini-batch SGD. For $m \in \mathbb{N}$ learners and mini-batch size $B \in \mathbb{N}$, a training set of size mB, i.e.,

$$E = \{(x_1, y_1), \dots, (x_{mB}, y_{mB})\}$$

can be split into local training sets of size B such that for learner $i \in [m]$ it yields

$$E_i = \{(x_{(i-1)B+1}, y_{(i-1)B+1}), \dots, (x_{(i-1)B+B}, y_{(i-1)B+B})\} \ .$$

Given learning rate $\eta \in \mathbb{R}_+$, and a model configuration $\mathbf{f} = (f, \dots, f)$, one step of $\mathcal{C}^{\text{mSGD}}$ can then be expressed as

$$\sigma_1\left(\left(\mathcal{A}^{\text{mSGD}}_{B,\eta}(E_1, f), \dots, \mathcal{A}^{\text{mSGD}}_{B,\eta}(E_i, f)\right)\right) = \frac{1}{m}\sum_{i=1}^{m}\left(f - \eta \textstyle\sum_{j=1}^{B} \nabla \ell(f, x_{(i-1)B+j}, y_{(i-1)B+j})\right) \ .$$

Note that in every round all local models are replaced by their joint average so that all local models are equal. We compare $\mathcal{C}^{\text{mSGD}}$ to the serial application of mini-batch SGD. It can be observed that continuous averaging with mini-batch SGD on $m \in \mathbb{N}$ learners with mini-batch size B is equivalent to serial mini-batch SGD with a mini-batch size of mB and a learning rate that is m times smaller.

Proposition 3.22. *For $m \in \mathbb{N}$ learners, a learning rate $\eta \in \mathbb{R}_+$, a mini-batch size $B \in \mathbb{N}$, mB training samples $(x_1, y_1), \dots, (x_{mB}, y_{mB})$, corresponding loss functions $\ell^i(\cdot) = \ell(\cdot, x_i, y_i)$, and a model $f \in \mathcal{F}$, it holds that*

$$\sigma_1\left(\left(\mathcal{A}^{mSGD}_{B,\eta}(f), \dots, \mathcal{A}^{mSGD}_{B,\eta}(f)\right)\right) = \mathcal{A}^{mSGD}_{mB,\eta/m}(f) \ .$$

Proof.

$$\sigma_1\left(\left(\mathcal{A}^{\text{mSGD}}_{B,\eta}(f),\dots,\mathcal{A}^{\text{mSGD}}_{B,\eta}(f)\right)\right)=\frac{1}{m}\sum_{i=1}^{m}\left(f-\eta\sum_{j=1}^{B}\nabla\ell^{(i-1)B+j}(f)\right)$$
$$=\frac{1}{m}mf-\frac{1}{m}\eta\sum_{i=1}^{m}\sum_{j=1}^{B}\nabla\ell^{(i-1)B+j}(f)=f-\frac{1}{m}\eta\sum_{j=1}^{mB}\nabla\ell^{j}(f)=\mathcal{A}^{\text{mSGD}}_{mB,\eta/m}(f)$$

□

Since continuous averaging for SGD, mini-batch SGD, and GD is equivalent to the serial application of the algorithms it also has the same loss. It follows that dynamic averaging with $b = 1$ and the same base learning algorithm is consistent.

For periodic averaging (with $b > 1$), it follows from Equation 5 in Zhang (2004) that—as long as the updates reduce the distance to the optimal model on average—the expected loss of periodic averaging is bounded by the expected loss of distributed mini-batch SGD. Since this retains the loss of serial mini-batch SGD (Dekel et al., 2012) (as well as SGD, and GD, depending on the setting of B), periodic averaging with $b > 1$ is consistent as well. Thus, also dynamic averaging with $b > 1$ is consistent.

The empirical evaluation in Section 3.5 shows that in practice, dynamic averaging indeed achieves the same model quality with substantially less communication. The amount of communication dynamic averaging requires is bounded by the amount that periodic averaging requires, since the local condition is only checked every $b \in \mathbb{N}$ rounds. In conclusion, dynamic averaging retains the optimality of periodic averaging using at most as much communication.

After bounding the cumulative loss of dynamic averaging and having shown that it is consistent for SGD and mini-batch SGD, in the following its communication is analyzed.

3.3.4. Communication Bounds

In the following it is analyzed under which conditions dynamic averaging is adaptive. For that, the communication is bounded for models of fixed size.

Proposition 3.23. *Let $\mathcal{A}$ be an online learning algorithm performing regret-proportional convex updates with $\gamma > 0$ and for all $f \in \mathcal{F}$, $x \in \mathcal{X}$, and $y \in \mathcal{Y}$ it holds that*

$$\|f-\mathcal{A}(f,x,y)\|\leq C\ell(f,x,y)$$

for a constant $C \in \mathbb{R}_+$. The communication $C_{\mathcal{D}}(T,m)$ of dynamic averaging with $\Delta \in \mathbb{R}_+$ using $\mathcal{A}$ is bounded by

$$C_{\mathcal{D}}(T,m)=c_m\frac{C}{\sqrt{\Delta}}L_{\mathcal{D}}(T,m)$$

where c_m is an upper bound on the amount of communication per round $t \in [T]$.

Proof. Dynamic averaging communicates only if a violation of a local condition $\|f^i_t - r_t\|^2 \leq \Delta$ occurs. By assumption, at each time point with at least one violation, dynamic averaging has communication costs of at most c_m, i.e., the cost of a full synchronization. Thus, we can bound the amount of communication by bounding the number of violations. That is, we derive a bound for $V^i(T)$, the number of time points $t \in [T]$ where the local condition of learner l is violated. For that, assume that at $t = 1$ all models are initialized with $f_1^1 = \cdots = f_1^m$ and $r_1 = \overline{f}_1$, i.e., for all local conditions it holds that $\|f_1^i - r_1\| = 0$. A violation, i.e., $\|f_t^i - r_t\| > \sqrt{\Delta}$, occurs if one local learner drifts away from r_t by more than $\sqrt{\Delta}$. In the worst case, each violation requires a full synchronization. After a full synchronization, $r_t = \overline{f}_t$, hence $\|f_t^i - r_t\| = 0$ and the situation is again similar to the initial setup for $t = 1$. Again in the worst case, a local learner drifts continuously in one direction until a violation occurs. Hence, the number of violations $V^i(T)$ can be bound by the sum of its drifts divided by $\sqrt{\Delta}$:

$$V^i(T) \leq \frac{1}{\sqrt{\Delta}} \sum_{t=1}^{T} \|f_t^i - f_{t+1}^i\| = \frac{1}{\sqrt{\Delta}} \sum_{t=1}^{T} \|f_t^i - \mathcal{A}\left(f_t^i, x_t^i, y_t^i\right)\| \leq \frac{1}{\sqrt{\Delta}} \sum_{t=1}^{T} C\ell\left(f_t^i, x_t^i, y_t^i\right) \quad .$$

To bound the communication it is necessary to bound the number of rounds $t \in [T]$ where at least one learner i has a violation, denoted $V(T)$. In the worst case, all violations at all local learners occur in different rounds, so that $V(T)$ can be upper bounded by the sum of local violations $V^i(T)$ which is again upper bounded by the cumulative sum of drifts of all local models:

$$V(T) \leq \sum_{i=1}^{m} V^i(T) \leq \frac{1}{\sqrt{\Delta}} \sum_{t=1}^{T} \sum_{i=1}^{m} C\ell(f_t^i, x_t^i, y_t^i) = \frac{C}{\sqrt{\Delta}} L_{\mathcal{D}}(T, m) \quad .$$

Since dynamic averaging has communication costs of at most c_m per round, the total amount of communication is

$$C_{\mathcal{D}}(T, m) = c_m V(T) \leq c_m \frac{C}{\sqrt{\Delta}} L_{\mathcal{D}}(T, m) \quad .$$

□

From this follows that if $c_m \in \mathcal{O}(m)$ and dynamic averaging is consistent, then it is also adaptive.

Theorem 3.24. *Let dynamic averaging $\mathcal{D}$ on $\mathcal{A}$ be consistent and $c_m \in \mathcal{O}(m)$, then the cumulative communication bound is*

$$\mathbf{C}_{\mathcal{D}}(T, m) \in \mathcal{O}\left(m L_{\mathcal{A}}(mT)\right) \quad ,$$

i.e., it is adaptive. Thus, dynamic averaging is efficient.

Proof. From $c_m \in \mathcal{O}(m)$ it follows that

$$C_{\mathcal{D}}(T, m) \leq c_m \frac{C}{\sqrt{\Delta}} L_{\mathcal{D}}(T, m) \leq \underbrace{c_m}_{\in \mathcal{O}(m)} \frac{C}{\sqrt{\Delta}} L_{\mathcal{D}}(T, m) \in \mathcal{O}\left(m L_{\mathcal{D}}(T, m)\right) \quad .$$

Since $\mathcal{D}$ is consistent it holds that $L_{\mathcal{D}}(T, m) \in \mathcal{O}(L_{\mathcal{A}}(mT))$ and thus

$$\mathbf{C}_{\mathcal{D}}(T, m) \in \mathcal{O}\left(m L_{\mathcal{A}}(mT)\right) \quad .$$

□

The loss bounds for online learning algorithms are typically sublinear in T, e.g., optimal regret bounds are in $\sqrt{T}$. In these cases, dynamic averaging has an amount of communication in $\left(m\sqrt{T}\right)$ which is smaller than $\mathcal{O}(mT)$ of periodic averaging by a factor of $\sqrt{T}$.

For linear models and neural networks, c_m is in $\mathcal{O}(m)$. For kernel models, however, both the weights α and the support vectors have to be communicated and the number of support vectors can grow with the number of examples observed. In the following, it is shown that dynamic averaging with kernel models can still be efficient.

For that, assume that the m learners maintain models in their support vector expansion. Let $S_t^i \subset \mathcal{X}$ denote the set of support vectors of learner $i \in [m]$ at time t and $\alpha_t^i \in \mathbb{R}^{|S_t^i|}$ the corresponding coefficients. Let $B_x \in \mathcal{O}(\dim \mathcal{X})$ be the number of bytes required to transmit one support vector and $B_\alpha \in \mathcal{O}(1)$ be the number of bytes required for the corresponding weight. Furthermore, let $I : \mathbb{N} \times [m] \to \{0, 1\}$ be an indicator function that is 1 if for learner i at time t a new support vector has been added during the update.

Again assume that a designated coordinator node performs the synchronizations, i.e., all local learners transmit their models to the coordinator which in turn sends the synchronized model back to each learner. Furthermore, assume that all protocols apply the following trivial communication reduction strategy. Let t' be the time of last synchronization. Assume the coordinator stored the support vectors of the last average model $\overline{S}_{t'}$. Whenever a learner i has to send its model to the coordinator, it sends all support vector coefficients α but only the new support vectors, i.e., only $S_t^i \setminus S_{t'}^i$. This avoids redundant communication at the cost of higher memory usage at the coordinator side. In turn, after averaging the models, the coordinator sends to learner i all support vector coefficients, but only the support vectors $\overline{S}_t \setminus S_t^i$.

We start by bounding the communication of a continuous protocol $\mathcal{C}$, i.e., one that transmits all models from each learner in each round. The trivial communication reduction technique discussed above implies that in each round, a learner transmits its full set of support vector coefficients and potentially one support vector—depending on whether a new support vector was added in this round. Thus, at time t learner i submits

$$\left|S_t^i\right| B_\alpha + I(t,i) B_x \tag{3.25}$$

bytes to the coordinator. The coordinator transmits to learner $i \in m$ all support vector coefficients of the average model and all its support vectors, except the support vectors S_t^i of the local model at learner i. Thus, it transmits the following amount of bytes.

$$\left|\overline{S}_t\right| B_\alpha + \left|\overline{S}_t \setminus S_t^i\right| B_x = \left|\bigcup_{j=1}^m S_t^j\right| B_\alpha + \left|\bigcup_{j=1}^m S_t^j \setminus S_t^i\right| B_x \ . \tag{3.26}$$

With this we can derive the following communication bound.

Proposition 3.27. *The communication of the continuous protocol $\mathcal{C}$ using kernel models on $m \in \mathbb{N}$ learners until time $T \in \mathbb{N}$ is bound by*

$$C_{\mathcal{C}}(T,m) \leq Tm2|\overline{S}_T|B_\alpha + m|\overline{S}_T|B_x \leq m^2T^2B_\alpha + m^2TB_x \in \mathcal{O}\left(m^2T^2\right) \ .$$

Proof. The constantly synchronizing protocol transmits at each time step from each learner a set of support vector coefficients and potentially one support vector to the coordinator. The amount of bytes is given in Eq. 3.25. The coordinator transmits the averaged model back to each learner with an amount of bytes as given in Eq. 3.26. Summing up the communication over $T \in \mathbb{N}$ time points and m learners yields

$$\begin{aligned}
C_{\mathcal{C}}(T,m) &= \sum_{t=1}^{T}\sum_{i=1}^{m}\left(|S_t^i|B_\alpha + I(t,i)B_x + \left|\bigcup_{j=1}^{m} S_t^j\right| B_\alpha + \left|\bigcup_{j=1}^{m} S_t^j \setminus S_t^i\right| B_x\right)\\
&= \sum_{t=1}^{T}\sum_{i=1}^{m}\left(|S_t^i|B_\alpha + \left|\overline{S}_t\right| B_\alpha + I(t,i)B_x + \left|\overline{S}_t \setminus S_t^i\right| B_x\right) \ .
\end{aligned}$$

We analyze this sum separately in terms of bytes required for sending the support vectors and bytes for sending the coefficients. The amount of bytes for sending the support vectors is bounded by $m|S_T^i|B_x$, as we show in the following.

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{i=1}^{m} I(t,i)B_x + \left|\overline{S}_t \setminus S_t^i\right| B_x = \underbrace{\sum_{t=1}^{T}\sum_{i=1}^{m} I(t,i)B_x}_{=|\overline{S}_T|B_x} + \sum_{t=1}^{T}\sum_{i=1}^{m}\left|\overline{S}_t \setminus S_t^i\right| B_x\\
=&|\overline{S}_T|B_x + \sum_{t=1}^{T}\sum_{i=1}^{m}\left|\left(\bigcup_{j=1}^{m} S_t^j \setminus \bigcup_{j=1}^{m} S_{t-1}^j\right) \setminus \left(S_t^i \setminus \overline{S}_{t-1}\right)\right| B_x\\
\leq&|\overline{S}_T|B_x + \sum_{t=1}^{T}\sum_{i=1}^{m}\sum_{\substack{j=1\\ j\neq i}}^{m} I(t,i)B_x \leq |\overline{S}_T|B_x + \sum_{t=1}^{T}\sum_{i=1}^{m}(m-1)I(t,i)B_x\\
\leq&|\overline{S}_T|B_x + (m-1)|\overline{S}_T|B_x = m|\overline{S}_T|B_x \ .
\end{aligned}$$

We now bound the amount of bytes required for sending the support vector coefficients.

$$\sum_{t=1}^{T}\sum_{i=1}^{m} \underbrace{|S_t^i|}_{\leq|\overline{S}_T|} B_\alpha + \underbrace{|\overline{S}_t|}_{\leq|\overline{S}_T|} B_\alpha \leq \sum_{t=1}^{T}\sum_{i=1}^{m} 2|\overline{S}_T|B_\alpha = Tm2|\overline{S}_T|B_\alpha \ .$$

From $\left|\overline{S}_T\right| \leq mT$ and the fact that we regard $B_\alpha \in \mathcal{O}(1)$ and $B_x \in \mathcal{O}(\dim \mathcal{X})$ as constants we can follow that

$$C_{\mathcal{C}}(T,m) \leq 2Tm|\overline{S}_T|B_\alpha + m|\overline{S}_T|B_x \leq m^2T^2B_\alpha + m^2TB_x \in \mathcal{O}(m^2T^2) \ .$$

□

Note that this communication bound implies that—unlike for linear models—synchronizing models in their support vector expansion requires even more communication than centralizing the input data. However, in real-time prediction applications, the latency induced by central computation can exceed the time constraints, rendering continuous averaging a viable approach nonetheless.

Similarly, the communication of periodic averaging $\mathcal{P}$ that communicates every $b \in \mathbb{N}$ rounds can be bounded by

$$C_{\mathcal{P}}(T,m) \le \frac{T}{b} 2m|\overline{S}_T|B_\alpha + m|\overline{S}_T|B_x \le \frac{T}{b} m^2 T B_\alpha + m^2 T B_x \in \mathcal{O}\left(\frac{1}{b} m^2 T^2\right) \ .$$

With this, now a communication bound for the dynamic protocol $\mathcal{D}$ is provided.

In the following theorem the overall communication is bounded by combining the bound on the number of synchronizations from Proposition 3.23 with an analysis of the amount of bytes transferred per synchronization.

Theorem 3.28. *Let $\mathcal{A}$ be an online learning algorithm using kernel models that is performing regret-proportional updates with $\gamma > 0$ for which holds that*

$$\|f - \mathcal{A}(f,x,y)\| \le C\ell(f,x,y)$$

for $C > 0$. The amount of communication $C_{\mathcal{D}}(T,m)$ of the dynamic protocol $\mathcal{D}$ running $\mathcal{A}$ in parallel on $m \in \mathbb{N}$ nodes until time $T \in \mathbb{N}$ with $\Delta \in \mathbb{R}$ and $b \in \mathbb{N}$ is bounded by

$$C_{\mathcal{D}}(T,m) \le \frac{C}{\sqrt{\Delta}} L_{\mathcal{D}}(T,m) \left(2m \left|\overline{S}_T\right| B_\alpha\right) + m \left|\overline{S}_T\right| B_x$$

Proof. Assume that at time T, the dynamic protocol performs a synchronization. Then, similar to the argument for the continuous protocol, the set of support vectors at time T is the same for all learners and independent of the number of synchronization steps before. In particular, it is the same if a synchronization was performed in every time step. Thus, again the amount of bytes required for sending the support vectors is bounded by $m\left|\overline{S}_T\right| B_x$. Let $\theta\colon \mathbb{N} \to \{0,1\}$ be an indicator function such that $\theta(t) = 1$ if at time t the dynamic protocol performed a synchronization and $\theta(t) = 0$ otherwise. Then, the amount of bytes required to send all the support vector coefficients until time T is

$$\sum_{t=1}^{T} \theta(t) \sum_{i=1}^{m} \left(\left|S_t^i\right| + \left|\overline{S}_t\right|\right) B_\alpha \le \underbrace{\sum_{t=1}^{T} \theta(t)}_{=V_{\mathcal{D}}(T)} \sum_{i=1}^{m} 2|\overline{S}_T|B_\alpha \le \underbrace{\frac{\eta}{\sqrt{\Delta}} L_{\mathcal{D}}(T,m)}_{\text{Prop. 3.23}} \left(2m|\overline{S}_T|B_\alpha\right)$$

Together with the amount of bytes required for exchanging all support vectors this yields

$$C_{\mathcal{D}}(T,m) \le \frac{\eta}{\sqrt{\Delta}} L_{\mathcal{D}}(T,m) \left(2m|\overline{S}_T|B_\alpha\right) + m\left|\overline{S}_T\right| B_x \ .$$

□

As mentioned before, loss bounds for online learning algorithms are typically sub-linear in T, e.g., in $\mathcal{O}(\sqrt{T})$. In these cases, dynamic averaging with kernel models has an amount of communication in $\mathcal{O}(m^2 T\sqrt{T})$ which is smaller than $\mathcal{O}(m^2T^2)$ of the continuously and periodic protocols again by a factor of $\sqrt{T}$.

From Theorem 3.28 in combination with Theorem 3.24 follows that if the size of kernel models could be kept constant, then dynamic averaging would be efficient as well. Unfortunately this is usually not the case, because the size of kernel models increases with the examples observed. Even more so in the distributed case, because the average of kernel models contains the union of the support vectors of all local models. In order to render dynamic averaging efficient, the size of the kernel models needs to be restricted. The following section investigates approaches from serial in-stream learning with kernel models that compress the model to a fixed size and shows that using these techniques, dynamic averaging with kernel models is indeed efficient.

3.3.5. Dynamic Averaging for Kernel Methods with Model Compression

In order to bound the size of kernel models, methods from serial kernelized in-stream learning approaches can be applied. These approaches perform model compression by reducing the number of support vectors, e.g., by truncating individual support vectors with small weights (Kivinen et al., 2004), or by projecting a single support vector on the span of the remaining ones (Orabona et al., 2009; Wang and Vucetic, 2010). In the following, some of these techniques are described.

Model Compression for Kernel Models

The goal of model compression is to bound the size of kernel models f by limiting the number of support vectors S in its support vector representation

$$f(\cdot) = \sum_{x\in S} \alpha_x k(x,\cdot) \enspace .$$

Given a limit $\zeta \in \mathbb{N}$ to the number of support vectors, a simple technique to compress a model is to discard the support vector with the smallest weight. That is, in each round, if the number of support vectors exceeds ζ, the support vector $x \in S$ with $\alpha_{x_r} = \min\{\alpha_x | x \in S\}$ is discarded. Let $\widetilde{f}$ denote the truncated model, then the error of this truncation strategy is as follows.

Lemma 3.29. *Let $f \in \mathcal{H}$ be a model in its support vector representation with kernel function k, support vectors $S \subset \mathcal{X}$, and corresponding weights $\{\alpha_x \in \mathbb{R} | x \in S\}$ and let $x_r \in S$ be the support vector with minimal weight, i.e.,*

$$\alpha_{x_r} = \min\{\alpha_x | x \in S\} \enspace .$$

Let $\widetilde{f}$ denote truncated model with support vectors $S \setminus \{x_r\}$ and corresponding weights. Then, the error of truncation is

$$\left\|f - \widetilde{f}\right\|^2 = \alpha_{x_r}^2 k(x_r, x_r) \enspace .$$

Proof.

$$\begin{aligned}\left\|f-\widetilde{f}\right\|^2 &= \langle f,f\rangle - 2\left\langle f,\widetilde{f}\right\rangle + \left\langle \widetilde{f},\widetilde{f}\right\rangle\\
&= \sum_{x\in S}\sum_{x'\in S}\alpha_x\alpha_{x'}k(x,x') - 2\sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x') + \left\langle \widetilde{f},\widetilde{f}\right\rangle\\
&= \sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x')\\
&\quad + \alpha_{x_r}\alpha_{x_r}k(x_r,x_r) + \sum_{x'\in S\setminus\{x_r\}}\alpha_{x_r}\alpha_{x'}k(x_r,x')\\
&\quad - 2\sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x') + \left\langle \widetilde{f},\widetilde{f}\right\rangle\\
&=\alpha_{x_r}^2 k(x_r,x_r) - \sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x')\\
&\quad + \sum_{x'\in S\setminus\{x_r\}}\alpha_{x_r}\alpha_{x'}k(x_r,x') + \sum_{x\in S\setminus\{x_r\}}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x')\\
&=\alpha_{x_r}^2 k(x_r,x_r) - \sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x') + \sum_{x\in S}\sum_{x'\in S\setminus\{x_r\}}\alpha_x\alpha_{x'}k(x,x')\end{aligned}$$

□

If k is the Gaussian kernel $k_\sigma(x,x') = \exp(-\|x-x'\|^2/(2\sigma^2))$, then $k(x_r,x_r) = 1$ and the error of truncation is $\alpha_{x_r}^2$. Kivinen et al. (2004) have shown that for SGD with regularization term $\lambda \in \mathbb{R}_+$ this truncation error is in $\mathcal{O}(\lambda^{-1}(1-\lambda)^\zeta)$.

Another approach to limiting the number of support vectors is to omit one support vector and express is approximately by the remaining ones. That is, if the number of support vectors exceeds the limit ζ, then the support vector $x_r \in S$ with smallest weight α_{x_r} is projected on the set of remaining support vectors (see, e.g., Orabona et al. (2009)). That is, for weights

$$\{\Delta\alpha_x | x\in S\setminus\{x_r\}\} = \arg\min_{\{\Delta\alpha_x|x\in S\setminus\{x_r\}\}\subset\mathbb{R}^{|S|-1}}\left\|\alpha_{x_r} - \sum_{x\in S\setminus\{x_r\}}\Delta\alpha_x x\right\|^2 \tag{3.30}$$

the compressed model is given by

$$\widetilde{f}(\cdot) = \sum_{x\in S\setminus\{x_r\}}(\alpha_x + \Delta\alpha_x)\,x \quad .$$

Setting the derivative of Equation 3.30 to zero and solving for $\Delta\alpha$ yields

$$\Delta\alpha = \alpha_{x_r}K_{x_r}^{-1}k_{x_r} \quad ,$$

where $\Delta\alpha$ denotes the vector of $\Delta\alpha_x$ for $x\in S\setminus\{x_r\}$, K_{x_r} denotes the kernel matrix $(k(x,x'))_{x\in S, x'\in S\setminus\{x_r\}}$ and k_{x_r} denotes the vector $(k(x,x'))_{x'\in S\setminus\{x_r\}}$.

However, model updates using these compression techniques are no longer regret-proportional convex updates. Therefore, the class of possible algorithms is extended to approximately regret-proportional convex updates.

Definition 3.31. *Let $\mathcal{A}$ be an incremental learning algorithm that performs regret-proportional convex updates, then $\widetilde{\mathcal{A}}$ performs* ***approximately loss-proportional convex updates*** *if for all $f \in \mathcal{F}$, $x \in \mathcal{X}$, and $y \in \mathcal{Y}$ it holds that*

$$\|\widetilde{\mathcal{A}}(f,x,y) - \mathcal{A}(f,x,y)\| \leq \epsilon \ .$$

From Lemma 3.29 follows that if $\mathcal{A}$ performs regret-proportional convex updates using kernel models, then using truncation leads to approximately regret-proportional convex updates.

Online Regret under Model Compression

This section shows that dynamic averaging remains efficient under model compression. To that extend, Lemma 3.14 is extended to approximately regret-proportional updates.

Lemma 3.32. *For two models $f, g \in \mathcal{F}$ and an incremental learning algorithm $\widetilde{\mathcal{A}}$ performing approximately regret-proportional convex updates, with $\|\widetilde{\mathcal{A}}(f,x,y) - \mathcal{A}(f,x,y)\| \leq \epsilon$ for the corresponding incremental learning algorithm $\mathcal{A}$ performing regret-proportional convex updates with $\gamma \in \mathbb{R}$, it holds that*

$$\|\widetilde{\mathcal{A}}(f,x,y) - \widetilde{\mathcal{A}}(g,x,y)\|^2 \leq \|f-g\|^2 - \gamma^2 \left(\ell(f,x,y) - \ell(g,x,y)\right)^2 + 2\epsilon^2 \ .$$

Proof. We abbreviate $\mathcal{A}(f,x,y)$ as $\mathcal{A}(f)$. Then $\|\widetilde{\mathcal{A}}(f) - \mathcal{A}(f)\| \leq \epsilon$ implies for $f, g \in \mathcal{F}$ that $\|\widetilde{\mathcal{A}}(f) - \widetilde{\mathcal{A}}(g)\|^2 \leq \|\mathcal{A}(f) - \mathcal{A}(g)\|^2 + 2\epsilon^2$. Together with the result from Lemma 3.14, i.e.,

$$\|\mathcal{A}(f) - \mathcal{A}(g)\|^2 \leq \|f-g\|^2 - \gamma^2 \left(\ell(f) - \ell(g)\right)^2 \ ,$$

follows the result. □

Using this property of approximately regret-proportional convex updates, the loss of dynamic averaging can be tied to that of periodic averaging.

Corollary 3.33. *Let $\mathcal{A}$ be an online learning algorithm that performs approximately regret-proportional convex update rule with loss proportionality $\gamma \in \mathbb{R}_+$ and approximation error $\epsilon \in \mathbb{R}_+$. Let $\mathbf{d}_1, \ldots \mathbf{d}_T$ and $\mathbf{s}_1, \ldots, \mathbf{s}_T$ be as in Theorem 3.19. Then it holds that*

$$L_{\mathcal{D}}(T,m) \leq L_{\mathcal{P}}(T,m) + \frac{T}{b\gamma^2}(\Delta + 2\epsilon^2) \ .$$

Proof. The proof is analogous to that of Theorem 3.19. We combine Lemma 3.32 with Lemma 3.13 which states that

$$\frac{1}{m}\sum_{i=1}^{m} \|\sigma_\Delta(\mathbf{d})^i - \sigma_b(\mathbf{s})^i\|^2 \leq \frac{1}{m}\sum_{i=1}^{m} \|d^i - s^i\|^2 + \Delta \ .$$

This yields for all $t \in [T]$ that

$$\sum_{i=1}^{m} \left\|d_{t+1}^i - s_{t+1}^i\right\|^2 \leq \sum_{i=1}^{m} \left\|d_t^i - s_t^i\right\|^2 - \gamma^2 \sum_{i=1}^{m} \left(\ell(d_t^i) - \ell(s_t^i)\right)^2 + \Delta + 2\epsilon^2 \quad .$$

By applying this inequality recursively for $t = 1, \dots, T$ it follows that

$$\sum_{i=1}^{m} \left\|d_{t+1}^i - s_{t+1}^i\right\|^2 \leq \sum_{i=1}^{m} \left\|d_1^i - p_1^i\right\|^2 + \left\lfloor \frac{T}{b} \right\rfloor (\Delta + 2\epsilon^2) - \gamma^2 \sum_{t=1}^{T} \sum_{i=1}^{m} \left(\ell(d_t^i) - \ell(s_t^i)\right)^2 .$$

Using $\mathbf{d}_1 = \mathbf{s}_1$, we conclude that

$$\begin{aligned}
\sum_{t=1}^{T} \sum_{i=1}^{m} \left(\ell(d_t^i) - \ell(s_t^i)\right)^2 \leq & \frac{1}{\gamma^2} \left(\left\lfloor \frac{T}{b} \right\rfloor (\Delta + 2\epsilon^2) - \sum_{i=1}^{m} \left\|d_{t+1}^i - s_{t+1}^i\right\|^2 \right) \leq \frac{1}{\gamma^2} \frac{T}{b} (\Delta + 2\epsilon^2) \\
\Leftrightarrow L_{\mathcal{D}}(T)^m - L_{\mathcal{P}}(T)^m \leq & \frac{T}{b\gamma^2} (\Delta + 2\epsilon^2)
\end{aligned}$$

□

Corollary 3.33 shows that model compression only adds a constant to the online regret of dynamic averaging. It remains to analyze the communication under model compression.

Communication Bounds under Model Compression

In order to show that dynamic averaging with kernel methods can be efficient, it has to be shown that the communication is bounded in the loss of the serial algorithm. For that, first the number of violations is bounded. Let $\widetilde{\mathcal{A}}$ be an incremental learning algorithm performing approximately regret-proportional convex updates, with $\|\widetilde{\mathcal{A}}(f,x,y) - \mathcal{A}(f,x,y)\| \leq \epsilon$ for the corresponding algorithm $\mathcal{A}$ performing regret-proportional convex updates. Let $\widetilde{C} \in \mathbb{R}_+$ be a constant such that $\|f - \widetilde{\mathcal{A}}(f,x,y)\| \leq \widetilde{C}\ell(f,x,y)$, then Proposition 3.23 can be applied to $\widetilde{\mathcal{A}}$ as well. Thus, Theorem 3.28 holds also for kernel models using model compression. Since in this case c_m is in $\mathcal{O}(m)$, it follows from Theorem 3.24 that it is efficient for SGD, mini-batch SGD and GD.

This concludes the theoretical analysis of both cumulative loss and communication of dynamic averaging. So far, no assumption on the network topology have been made. The following section discusses the application of dynamic averaging in various network topologies in relation to this theoretical analysis.

3.4. Network Topologies

Dynamic averaging as a protocol can be applied to various network topologies. This section discusses a few examples and how dynamic averaging can be applied there.

The most straight-forward architecture is a star topology with a coordinator at the center, i.e., a dedicated computation node that performs the model aggregation (see Figure 3.4(a)). In this topology, learners send violations to the coordinator, which performs the local balancing and sends the average model to the local learners. A full synchronization in this topology requires $c_m = 2m$ messages for $m \in \mathbb{N}$ learners learners.

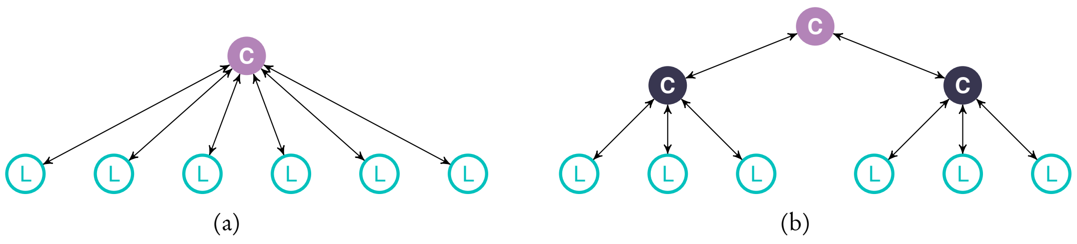


Figure 3.4.: Network topologies of learners with dedicated coordinator nodes: (a) single coordinator as in Federated Learning; (b) hierarchical structure of coordinators.

The star topology can be extended using multiple, hierarchical coordinator nodes. Each coordinator node receives violations from its children—either learners or coordinators. Each local coordinator may perform local balancing, the coordinator at the root performs full synchronizations. Let $c \in \mathbb{N}$ denote the number of coordinators, then a full synchronization in this setup requires $\mathcal{O}(cm)$ messages. Treating c as a constant, then $c_m \in \mathcal{O}(m)$. If the hierarchical topology is a perfect r-ary tree, then this tree has

$$\frac{rm-1}{r-1}$$

many nodes and

$$\frac{rm-1}{r-1} - 1$$

many edges. Since the number of messages for a full synchronization is twice the number of edges, again $c_m \in \mathcal{O}(m)$.

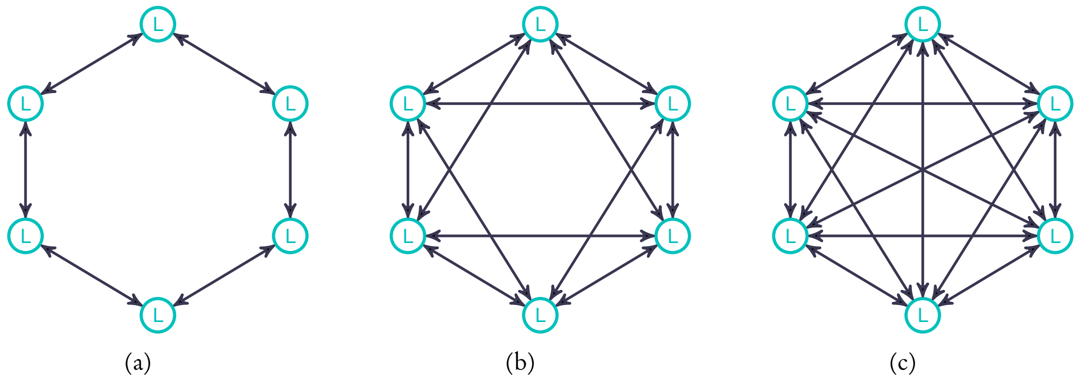


Figure 3.5.: Decentralized network topology of learners without coordinator; (a) only nearest neighbors are connected, (b) 2-nearest neighbors are connected, and (c) all learners are connected.

Dynamic averaging can also be applied to topologies without a dedicated coordinator node (see Figure 3.5 for an illustration). That is, the network topology is a connected undirected graph with m vertices. In this case, learners communicate in a peer-to-peer fashion. For that, a gossip-like protocol can be applied (Kempe et al., 2003) to perform synchronizations. For example, for a full synchronization the learner with the first violation requests all local models from its neighbors. These then also request the local models of their neighbors. Each learner averages all received local models and sends the average to the neighbor that first requested it. At the end, the learner with the first violation can compute the average of all models. This is then send to all learners in a similar fashion. Not counting the requests for models [4], the overall number of messages is twice the number of edges of a spanning tree of the network. Since the number of edges in any tree with m vertices is $m - 1$, again $c_m \in \mathcal{O}(m)$. In all cases $c_m \in \mathcal{O}(m)$ and thus Corollary 3.24 holds, i.e., if dynamic averaging is consistent, then it is efficient.

In the following, dynamic averaging is empirically evaluated. For that, a star topology with a dedicated coordinator node in its center is assumed.

3.5. Empirical Evaluation

This section investigates the practical performance of dynamic averaging for settings ranging from clean linearly separable data, over inseparable data with a reasonable linear approximation, up to real-world data without any guarantee. All experiments are conducted on a simulated distributed environment which allows to ignore practical problems, such as asynchronicity of learners, message passing times and message loss [5]. An implementation of the algorithm within a real distributed online learning framework using Apache Storm is presented in Appendix A.1.

3.5.1. Linearly Separable Data

To investigate the performance of dynamic averaging it is first compared to periodic averaging using linear models on a linearly separable synthetic dataset, denoted **disjunctions**. The dataset is generated by choosing a random disjunction over $d \in \mathbb{N}$ literals. The input space consists of binary vectors of dimension $d \in \mathbb{N}$, i.e., $\mathcal{X} = \{0,1\}^d$. The label is given by the value of the disjunction over the random sample, i.e., $\mathcal{Y} = \{-1,1\}$. Formally, a target disjunction is identified with a binary vector $z \in \{0,1\}^d$. A data point $x \in \mathcal{X}$ is labeled positively $y = 1$ if $\langle x, z\rangle \geq 1$ and otherwise receives a negative label $y = -1$. The target disjunction is drawn randomly at the beginning of the learning process. In order to have balanced classes, the disjunctions as well as the data points are generated such that each coordinate is set independently to 1 with probability $\sqrt{1 - 2^{-1/d}}$.

[4] The number of model requests in this setup can be up to the number of edges in the graph which is at most m^2 for a fully connected graph. At the same time, these messages can be very small. Still, taking them into account, the cost of each synchronization can be in $\mathcal{O}(m^2)$.

[5] The implementation of the experimental framework in python is open source and can be found at `https://bitbucket.org/Michael_Kamp/decentralized-machine-learning`.

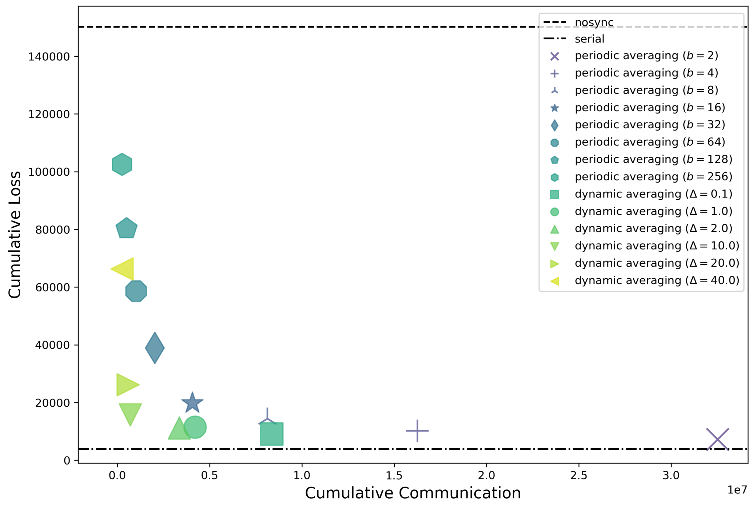
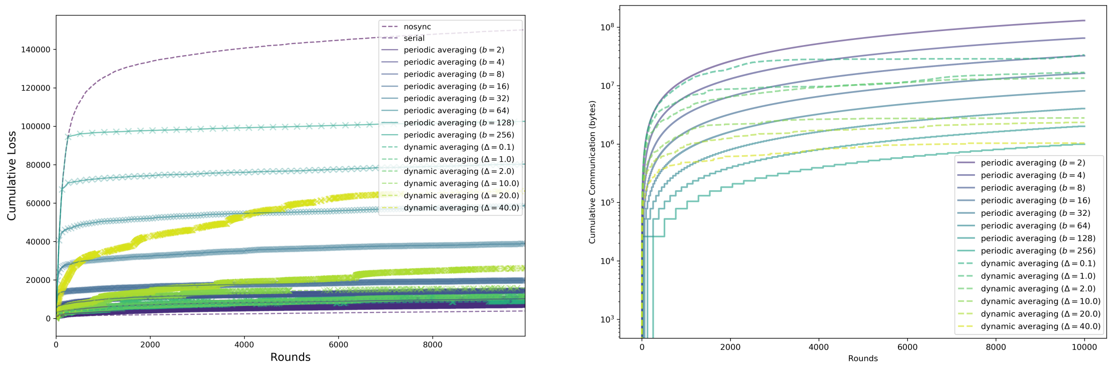


Figure 3.6.: Trade-off between cumulative hinge loss and cumulative communication after processing $T = 10\,000$ rounds for several distributed learning systems with $m = 128$ learners jointly learning disjunctions with $d = 50$ using SGD as base learning algorithm.



(a) (b)

Figure 3.7.: Cumulative hinge loss (a) and cumulative communication (b) (in log-scale) over $T = 10\,000$ rounds of several distributed learning systems with $m = 128$ learners. A synchronization is indicated by a cross in the cumulative loss plot (a).

The experiment is conducted on a distributed system with $m = 128$ learners using multiple distributed learning protocols. The base learning algorithm is regularized SGD with learning rate $\eta = 10.0$ and regularization parameter $\lambda = 1.0$. The loss function used is hinge loss, i.e.,

$$\ell(f, x, y) = \max(1 - yf(x), 0) \quad ,$$

for which the derivative with respect to f is given by

$$\nabla_f \ell(f, x, y) \begin{cases} -yx, & \text{if } yf(x) < 1 \\ 0, & \text{else } . \end{cases}$$

Dynamic averaging is compared to periodic averaging (see Table 3.1 for the configurations), as well as two baselines: **serial** denotes the serial application of the base learning algorithm, **nosync** denotes a distributed setup in which the learners run independently without communication. All protocols are run for $T = 10\,000$—for the serial baseline this means that $Tm = 1\,280\,000$ examples are processed. For the target disjunction the number of literals, and thus the dimension of the input space, is $d = 50$.

Figure 3.6 shows the trade-off between cumulative loss and cumulative communication after processing $T = 10\,000$ rounds. It shows that periodic averaging with $b = 2$ indeed is able to achieve a cumulative loss similar to the serial application of the base learning algorithm. For that, it requires a substantial amount of communication. This can be reduced by increasing b at the cost of higher cumulative loss. Dynamic averaging with $\Delta = 0.1$ achieves the same loss as the best periodic one with an order of magnitude less communication. Increasing Δ further reduces communication at the cost of cumulative loss. However, the increase in loss is far lower than for periodic averaging. For example, dynamic averaging with $\Delta = 10.0$ still achieves a cumulative loss comparable to serial with two orders of magnitude less communication than periodic with $b = 2$. Moreover, for every desired trade-off between loss and communication, there is a setting for dynamic averaging which outperforms periodic averaging.

To illustrate the behavior of dynamic averaging, Figure 3.7 shows the development of cumulative hinge loss and communication over time. The cumulative loss (Figure 3.7(a)) of serial and the periodic averaging variants increases strongly in the beginning and then plateaus after the learners converged to the optimal model. The same holds for the nosync baseline, only that the loss is still increasing after 10 000 rounds (recall that at the end each individual learner has seen 10 000 examples, while the serial baseline has seen 10 000 examples already after round $t = 78$). The cumulative loss of dynamic averaging does not behave as regularly, since it depends on the

protocol configurations

name	**parameters**
periodic protocol σ_b	$b = 2$
	$b = 4$
	$b = 8$
	$b = 16$
	$b = 32$
	$b = 64$
	$b = 128$
	$b = 256$
dynamic protocol $\sigma_{\Delta,b}$	$b = 2, \Delta = 0.1$
	$b = 2, \Delta = 1.0$
	$b = 2, \Delta = 2.0$
	$b = 2, \Delta = 10.0$
	$b = 2, \Delta = 20.0$
	$b = 2, \Delta = 40.0$

Table 3.1.: Overview of protocol configurations for the disjunction dataset using linear models. Except for Δ and b, all parameters are kept constant between configurations.

time and amount of synchronizations. For $\Delta = 40.0$, in phases where less loss is suffered, the amount of synchronizations is reduced (as can be observed from the lower number of crosses indicating the synchronizations).

Moreover, as soon as the loss plateaus for dynamic averaging, the communication is reduced substantially, as can be seen in Figure 3.7(b). In particular for $\Delta = 0.1$, dynamic averaging invests as much communication as periodic averaging with $b = 2$ in the first 2 000 rounds, leading to a quick convergence to the optimal model. After that, communication is substantially reduced, up to the point where it reaches quiescence. This confirms that dynamic averaging indeed invests communication in hard phases, where it is most useful and reduces it when it is not necessary anymore. Moreover, the experiment shows that it indeed retains the performance of periodic averaging with substantially less communication and can achieve a performance similar to serial.

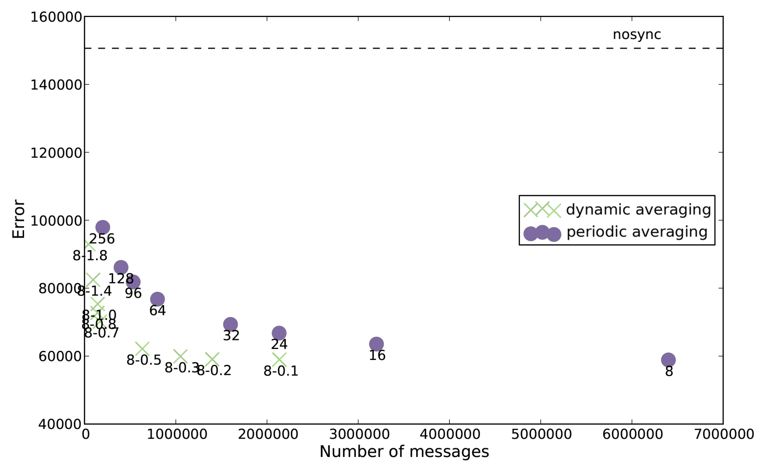
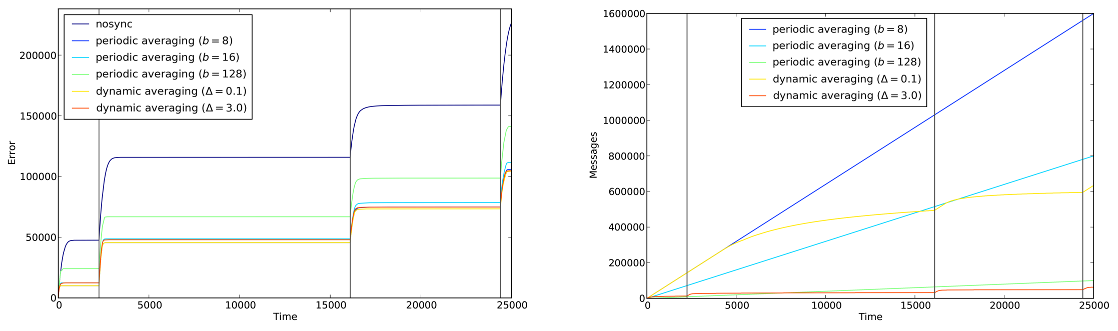


Figure 3.8.: Performance of static and dynamic synchronization for tracking a rapidly drifting disjunction over 100-dimensional data with $m = 512$ learners.

To furthermore show that dynamic averaging is capable of handling concept drift, the data is generated by rapidly drifting random disjunction. That is, the target disjunction is drawn randomly at the beginning of the learning process and is randomly re-set after each round with a fixed drift probability of 0.000001. Figure 3.8 presents the result for dimensionality $d = 100$, with $m = 512$ nodes, processing $12.8M$ data points through $T = 100\,000$ rounds. For divergence thresholds up to 0.3, dynamic averaging can retain the error of periodic averaging with $b = 8$. At the same time the communication is reduced to 9.8% of the original number of messages. An approximately similar amount of communication reduction can also be achieved using periodic averaging with $b = 96$. This approach, however, only retains 61.0% of the accuracy of periodic averaging with $b = 8$.

Figure 3.9(a) and (b) provide some insight into how the two evaluation metrics develop over time. Target drifts are marked with vertical lines that frame episodes of a stable target disjunction. At the beginning of each episode there is a relatively short phase in which additional errors are accumulated and the communicative protocols acquire an advantage over the baseline of never synchronizing. This is followed by a phase during which no additional error is made.



(a) (b)

Figure 3.9.: Cumulative loss (a) and cumulative communication (b) over time for tracking a rapidly drifting disjunction for different synchronization protocols; vertical lines depict drifts.

Here, the communication curve of the dynamic protocols remain constant acquiring a gain over the static protocols in terms of communication.

3.5.2. Non-separable Data with Noise

We now turn to a harder experimental setting, in which the target distribution is given by a rapidly drifting two-layer neural network. For this target even the Bayes optimal classifier per episode has a non-zero error, and, in particular, the generated data is not linearly separable. Intuitively, it is harder in this setting to save communication, because a non-zero residual error can cause the linear models to periodically fluctuate around a local loss minimizer—resulting in crossings of the variance threshold even when the learning processes have reached their asymptotic regime. We choose the network structure and parameter ranges in a way that allow for a relatively good approximation by linear models (see Bshouty and Long (2012)). The process for generating a single labeled data point is as follows: First, the label $y \in Y = \{-1, 1\}$ is drawn uniformly from Y. Then, values are determined for hidden

protocol configurations	
name	**parameters**
periodic protocol σ_b	$b = 8$
	$b = 12$
	$b = 16$
	$b = 24$
	$b = 32$
	$b = 48$
	$b = 64$
dynamic protocol $\sigma_{\Delta,b}$	$b = 8, \Delta = 0.005$
	$b = 8, \Delta = 0.05$
	$b = 8, \Delta = 0.08$
	$b = 8, \Delta = 0.1$
	$b = 8, \Delta = 0.15$
	$b = 8, \Delta = 0.18$
	$b = 8, \Delta = 0.2$
	$b = 8, \Delta = 0.25$

Table 3.2.: Overview of the different communication protocol configurations for the disjunction dataset using linear models. Except for protocol parameters Δ and b, all other parameters are kept constant between configurations.

variables H_i with $1 \leq i \leq \lceil \log n \rceil$ based on a Bernoulli distribution $P[H_i = \cdot | Y = y] = \text{Ber}(p^h_{i,y})$. Finally, $x \in X = \{-1, 1\}^n$ is determined by drawing x_i for $1 \leq i \leq n$ according to $P[X_i = x_i, | H_{p(i)} = h] = \text{Ber}(p^o_{i,h})$ where $p(i)$ denotes the unique hidden layer parent of x_i. In order to ensure that the data can be linearly approximated, the parameters of the output layer are drawn such that $|p^o_{i,-1} - p^o_{i,1}| \geq 0.9$, i.e., their values have a high *relevance* in determining the hidden values. As in the disjunction case all parameters are re-set randomly after each round with a fixed drift probability (here, 0.01).

Linear Models

As a first experiment, linear models are used with regularized passive aggressive updates with hinge loss and $C = 10.0$ as base learning algorithm. The configurations for periodic and dynamic averaging are summarized in Table 3.2. Fig. 3.10 contains the results for dimensionality

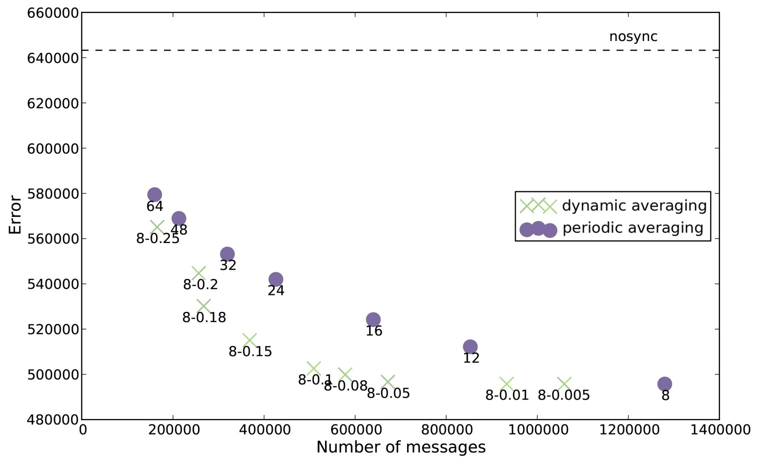


Figure 3.10.: Performance of periodic and dynamic averaging protocols that track a neural network with one hidden layer and 150 output variables using 1024 nodes.

150, with $k = 1024$ nodes, processing $m = 2.56M$ data points through $T = 10000$ rounds. For variance thresholds up to 0.08, dynamic synchronization can retain the error of the baseline. At the same time, the communication is reduced to 45% of the original number of messages. Moreover, even for thresholds up to 0.2, the dynamic protocol retains more than 90% of the accuracy of static synchronization with only 20% of its communication.

Kernel Models

After having seen that dynamic averaging performs well for linear models, the next experiment investigates how using kernel models improves the cumulative loss. As discussed in Section 3.3.5, using kernel models comes at the price of substantially higher computation. The experiment compares nosync, serial, periodic and dynamic averaging for linear and kernel methods on $m = 4$ learners. The learning algorithm used is SGD with $\eta = 1.0$ and $\lambda = 0.1$ using hinge loss. For the kernel models, a Gaussian kernel with

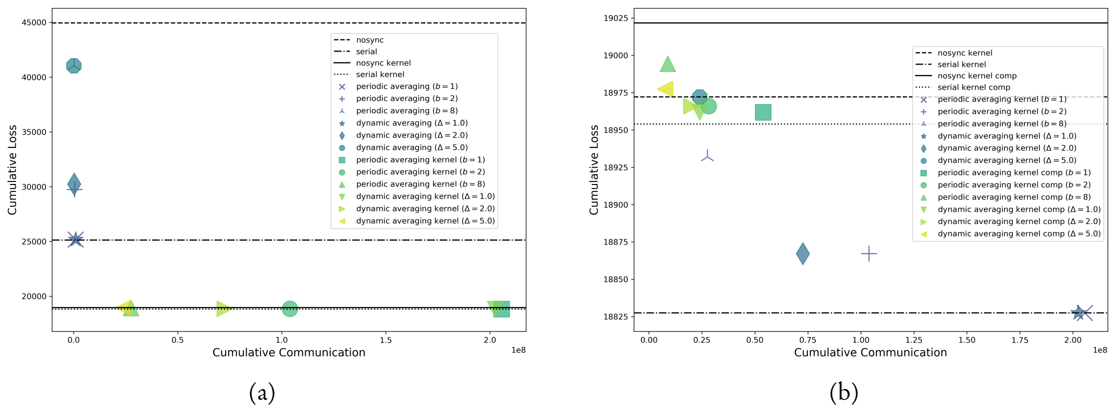


Figure 3.11.: Comparison of periodic and dynamic averaging on $m = 4$ learners that track a neural network with one hidden layer and 150, comparing (a) linear and kernel models, as well as (b) kernel models with and without model compression.

$\sigma = 0.2$ is used. Figure 3.11(a) shows the trade-off between cumulative loss and communication for periodic and dynamic averaging once with linear and once with kernel models. The results indicate that using kernel models substantially reduces the loss to nearly half of the serial baseline using linear models. For that, the amount of communication required is orders of magnitudes larger (periodic averaging kernel ($b = 1$) requires 3 orders of magnitude more communication than periodic averaging ($b = 1$) using linear models). However, whether using linear or kernel models, dynamic averaging allows to reach the same cumulative loss as periodic averaging with substantially less communication.

In order to show the effect of model compression, Figure 3.11(b) compares kernel models without model compression to a support vector truncation approach that only keeps the 50 support vectors with the highest weights. The results show that model compression substantially reduces the communication at the cost of predictive performance. However, using dynamic averaging with $\Delta = 1.0$ and kernel compression only requires an amount of communication comparable to periodic averaging with linear models while suffering significantly fewer loss.

The experiments support the claim that for application scenarios that can be solved better by non-linear models, using periodic

protocol configurations	
name	**parameters**
periodic protocol σ_b	$b = 1$
	$b = 2$
	$b = 4$
dynamic protocol $\sigma_{\Delta,b}$	$b = 1, \Delta = 0.3$
	$b = 1, \Delta = 0.7$
	$b = 1, \Delta = 1.0$

Table 3.3.: Overview of the different communication protocol configurations for the analysis of concept drift with neural networks. Except for protocol parameters Δ and b, all other parameters are kept constant between configurations.

averaging with kernel models achieves good predictive performance at the cost of a prohibitively large amount of communication. By using dynamic averaging the amount of communication can be substantial reduced. Combining this with model compression allows to achieve similar predictive performance with an amount of communication comparable to that of using linear models.

Neural Networks

After having analyzed the performance on linear and kernel models, the following experiment evaluates dynamic averaging using neural networks. For that, a fully connected multi-layer perceptron (MLP) is used. Table 3.4 provides an overview of the network layers and the amount of neurons used. The settings of periodic and dynamic averaging are listed in Table 3.3. Figure 3.12(a) shows that in terms of predictive performance, dynamic and periodic averaging

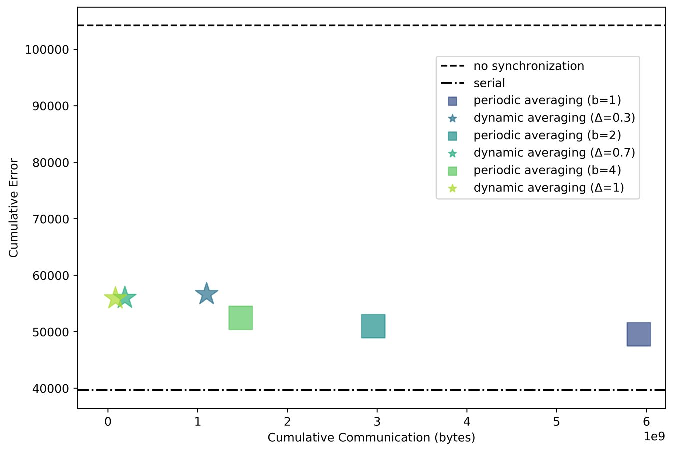
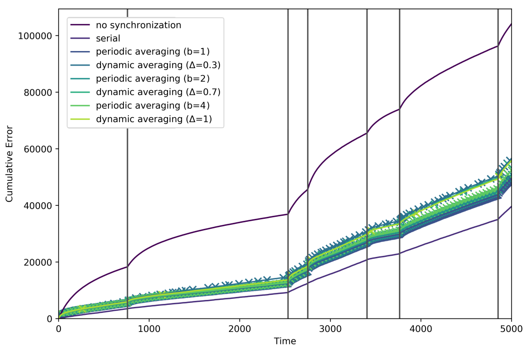


Figure 3.12.: Experiment with periodic and dynamic averaging protocols on $m = 100$ learner after training on 5000 samples per learner from a synthetic dataset with concept drifts.

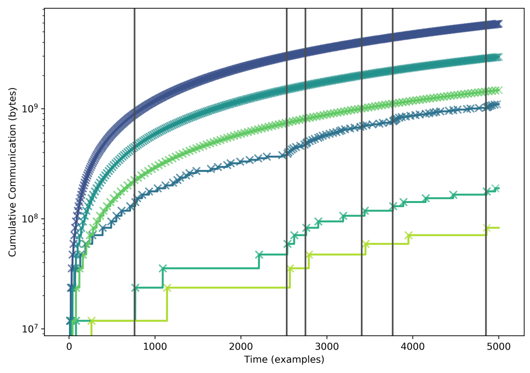
Layer Type	Output Shape	#Weights
Dense	(256)	38 400
Dropout	(256)	65 536
Dense	(64)	16 384
Dropout	(64)	4 096
Dense	(1)	64
Total		124 480

Table 3.4.: The architecture of the neural network used for the non-separable dataset.

perform similarly. At the same time, dynamic averaging requires up to an order of magnitude less communication to achieve it. Figure 3.13, that shows the development of cumulative



(a) cumulative loss



(b) cumulative communication

Figure 3.13.: Development of cumulative loss and communication during training on 5000 examples per node from a synthetic dataset with concept drifts (indicated by vertical lines).

loss and communication over time, confirms that dynamic averaging adapts the amount of communication: right after a concept drift the number of synchronizations (indicated by a cross mark) is high and decreases as soon as the instantaneous loss of the learners—i.e., observable as the growth of the cumulative error—decreases. This indicates that dynamic averaging invests communication when it is most impactful and can thereby save a substantial amount of communication in between drifts.

3.5.3. Real-world Data

The experimental section is concluded with tests on three real-world datasets containing, Twitter short messages, data from the LHC Atlas experiment on supersymmetric particles (SUSY) (Baldi et al., 2014), and stock prices, respectively.

The data from Twitter has been gathered via its streaming API (`https://dev.twitter.com/docs/streaming-apis`) during a period of 3 weeks (Sep 26 through Oct 15 2012). Inspired by the content recommendation task, the problem considered is predicting whether a given tweet will be re-tweeted within one hour after its posting—for a number of times that lies below or above the median hourly re-tweet number of the specific Twitter user. The feature space are the top-1000 textual features (stemmed 1-gram, 2-gram) ranked by information gain, i.e., $X = \{0,1\}^{1000}$. The base learning algorithm used is regularized passive aggressive (PA-I) with aggressiveness parameter $C = 0.25$.

In Fig. 3.14 we present the result for dimensionality $n = 100$, with $k = 512$ nodes, processing $m = 12.8M$ data points through $T = 100000$ rounds. For divergence thresholds up to 0.3, dynamic synchronization can retain the error number of statically synchronizing every 8 rounds. At the same time the communication is reduced to 9.8% of the original number of messages. An approximately similar amount of communication reduction can also be achieved using static synchronization by increasing the batch size to 96. This approach, however, only retains 61.0% of the accuracy of statically synchronizing every 8 rounds.

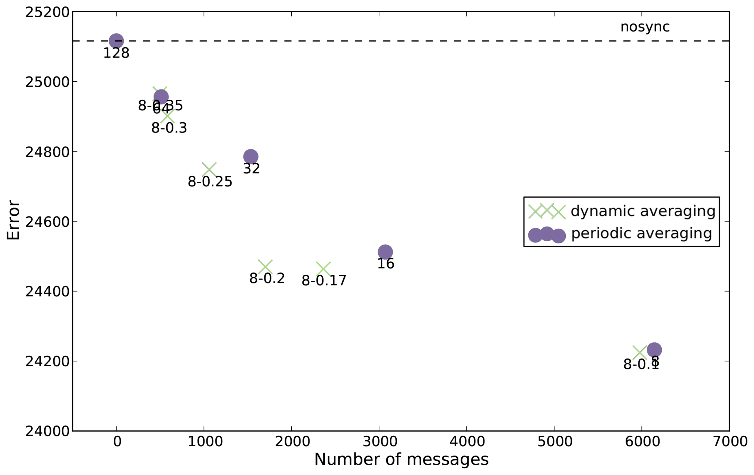


Figure 3.14.: Performance of static and dynamic synchronization with 256 nodes that predict Twitter retweets over 1000 textual features.

For the next set of experiments, the task is to predict the class of instances drawn from the SUSY dataset from the UCI machine learning repository (Lichman, 2013). The first experiment compares linear to kernel models, the latter with and without model compression. The

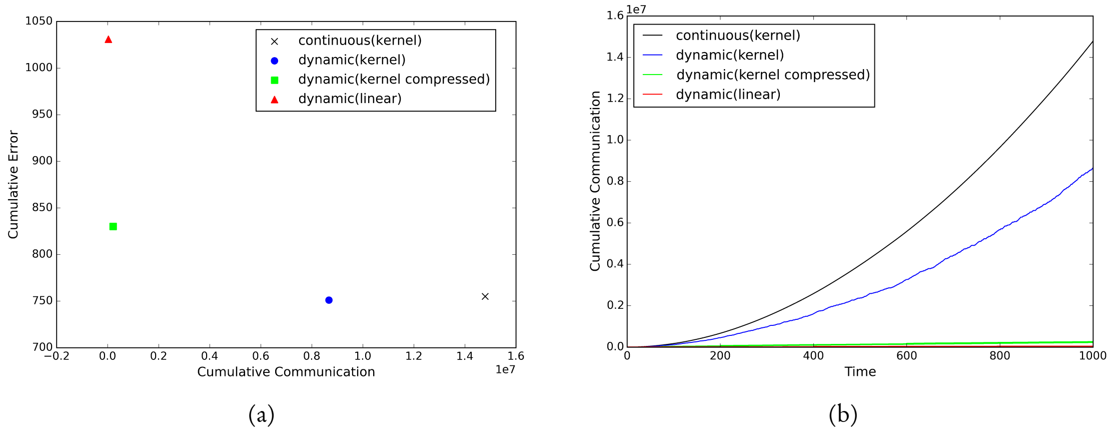


Figure 3.15.: (a) Trade-off between cumulative error and cumulative communication, and (b) detailed view on the cumulative communication over time of dynamic averaging with $m = 4$ learners, each processing 1000 examples. The learning task is classifying instances from the UCI SUSY dataset. Parameters of the learners are optimized on a separate set of 200 instances per learner. The compression technique for kernel models is truncation with a support vector limit of 50.

results are presented in Figure. 3.15. The comparison of cumulative loss and communication in Figure 3.15(a) shows that dynamic averaging using linear models suffers the highest amount of loss, but since the linear models are small compared to support vector expansions, the cumulative communication is small. A continuously synchronizing protocol using support vector expansions has a significantly smaller loss at the cost of very high communication, since each synchronization requires to send models with a growing number of support vectors. Using dynamic averaging, the communication can be reduced without losing in prediction quality. In addition, when using model compression the communication can be further reduced to an amount similar to the linear model, but at the cost of higher cumulative loss.

These results indicate that linear models are not suitable for the SUSY dataset. Kernel models perform substantially better, at the cost of higher communication. Moreover, making a prediction with a model with high number of support vectors has a larger runtime. The same holds for calculating the distance to the reference point for the local conditions of dynamic averaging, as well as the averaging of kernel models. That is why the experiments on SUSY were only conducted using 4 learners.

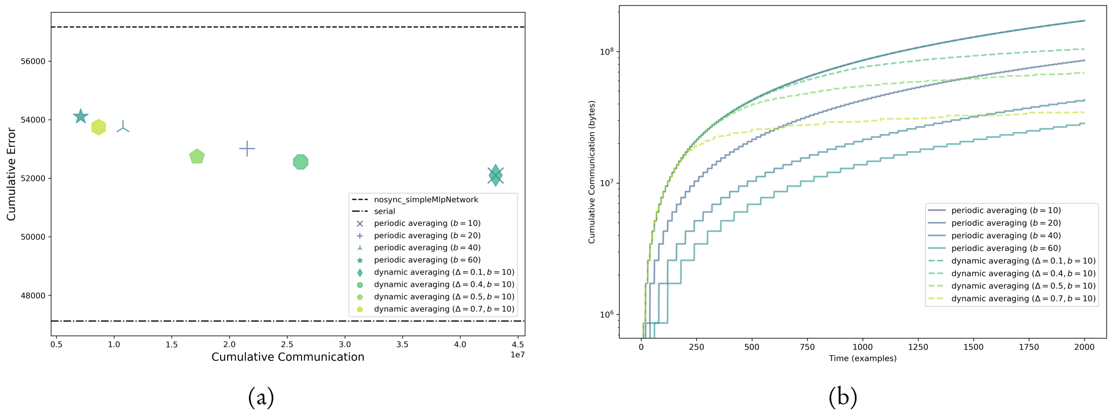


Figure 3.16.: (a) Trade-off between cumulative error and cumulative communication, and (b) detailed view on the cumulative communication over time of dynamic averaging with $m = 50$ learners each processing 2000 examples from the UCI SUSY dataset. Parameters of the learners are optimized on a separate set of 200 instances per learner.

Similarly to kernel models, neural networks allow to model non-linear dependencies, but have a fixed model size. Figure 3.16 shows the results of an experiment with neural networks on the SUSY dataset. Table 3.5 provides an overview of the network layers and the amount of weights used. The network used is a fully connected MLP using rectified linear units (ReLU) as activation function and softmax at the output neuron. Comparing the cumulative loss to the cumulative communication in Figure 3.16(a) shows that, again, dynamic averaging allows to achieve a performance similar to periodic averaging with substantially less communication. Looking at the development of cumulative communication over time in Figure 3.16(b) shows that also on SUSY, dynamic averaging invests more communication in the beginning—comparable to the most frequently communicating periodic averaging protocol—and reduces the amount of communication during the learning process.

Layer Type	Output Shape	#Weights
Dense	(64)	1 152
Dense	(32)	2 048
Dense	(16)	512
Dense	(1)	16
Total		3 728

Table 3.5.: The architecture of the neural network used for the SUSY.

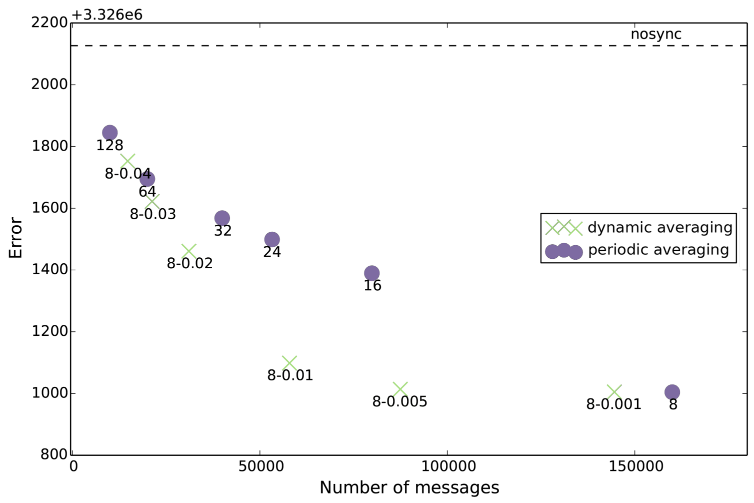
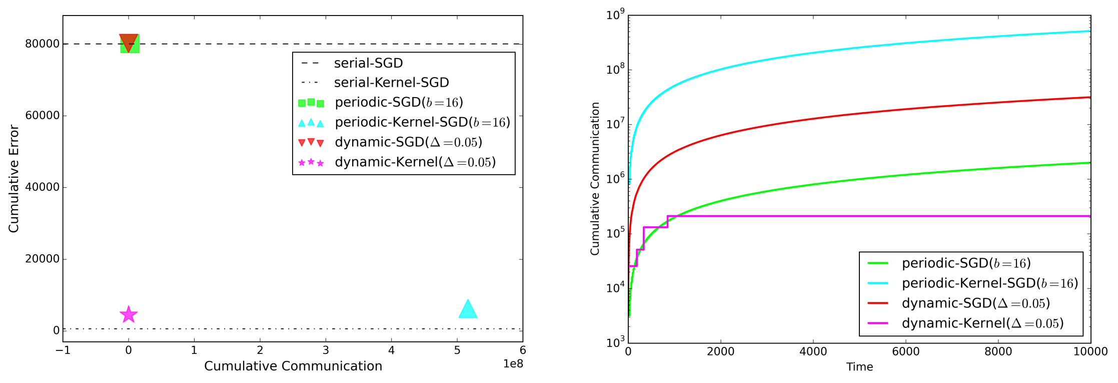


Figure 3.17.: Performance of dynamic and periodic averaging using linear models with 256 learners that predict stock prices based on prices and sliding averages of the stock itself and all other stocks from the S&P100 index.

The last experiment is conducted on a stock price dataset. The data is gathered from Google Finance (`http://www.google.com/finance`) and contains the daily closing stock prices of the S&P100 stocks between 2004 and 2012. Inspired by algorithmic trading, the task is to predict tomorrow's closing price of a single target stock based on all stock prices and their moving averages (11, 50, and 200 days) of today, i.e., $\mathcal{X} = \mathbb{R}^{400}$ and $\mathcal{Y} = \mathbb{R}$. The target stock is switched with probability 0.001. Here, regularized passive aggressive updates with linear models, the epsilon insensitive loss, $\epsilon = 0.1$, and a regression parameter of $C = 1.0$ are used.

The results for 1.28M data points distributed to $k = 256$ nodes are presented in Fig. 3.17. Again, the gap between no synchronization and the baseline is well preserved by partial synchronizations. A threshold of 0.005 preserves 99% of the predictive gain using 54% of communication. The trade-off is even more beneficial for threshold 0.01 which preserves 92% of the gain using only 36% communication.

Figure 3.18 shows a comparison of kernel to linear models on the financial data, where 32 learners predicted the stock price of a target stock. We can see that for this difficult learning task linear models perform poorly compared to non-linear models using a Gaussian kernel function. Simultaneously, the communication required to periodically synchronize these non-linear models is larger than for linear models by more than two orders of magnitude. Using the dynamic protocol with kernel models we could reduce the error by an order of magnitude compared to using linear models (a reduction by a factor of 18). At the same time, the communication is reduced by more than three orders of magnitude compared to the static protocol (by a factor of 2433), which is yet an order of magnitude smaller than the communication when using linear models (by a factor of 10). Moreover, within less than 2000 rounds, the dynamic protocol reaches quiescence, as it is implied by the efficiency criterion.



(a) (b)

Figure 3.18.: (a) Trade-off between cumulative error and cumulative communication, and (b) detailed view on the cumulative communication over time of the dynamic protocol versus a periodic protocol. 32 learners perform a stock price prediction task using SGD (learning rate η and regularization parameter λ optimized over 200 instances, with $\eta = 10^{-10}$, $\lambda = 1.0$ for the periodic protocol, and $\eta = 1.0$, $\lambda = 0.01$ for the dynamic protocol) updates, either with linear models or with non-linear models (Gaussian kernel with number of support vectors limited to 50 using the truncation approach of Kivinen et al. (2004)).

3.6. Discussion

This chapter has proposed a novel type of synchronization operator that allows to aggregate only subsets of models and schedule the aggregation dynamically, based on the model divergence. This partial synchronization operator was analyzed for averaging as aggregation operator. The resulting dynamic averaging protocol is the first ever efficient protocol, i.e., it is consistent and adaptive at the same time. It has been shown both theoretically and practically that dynamic averaging outperforms periodic protocols, in some cases reducing communication by orders of magnitude.

In the following, properties and limitations of the proposed protocol are discussed.

Finding the right divergence threshold for the dynamic protocol, i.e., one that suits the desired trade-off between service quality and communication, is in practice a neither intuitive nor trivial task. **How can the right Δ be determined?** The threshold can be selected using a small data sample, but the communication for a given threshold can vary over time and is also influenced by other parameters of the learner. Thus, a direction for future research is to investigate an adaptive divergence threshold. That is, a scheme for updating the threshold based on the number of violation could be employed. E.g., for every round without a full synchronization, each local learner reduces Δ by a constant factor. In case of a full synchronization, Δ is increased again. This way, all local Δ are equal. Such schemes for Δ could allow for a more direct selection of the desired trade-off between service quality (i.e., predictive performance) and communication.

A limit of the employed notion of efficiency is that it only takes into account the sum of messages. **What about the peak communication?** In large data centers, where the distributed learning system is run next to other processes, the main bottleneck is the overall amount of transmitted bytes and a high peak in communication can often be handled by the communication infrastructure or evened out by a load balancer. In smaller systems, however, high peak communication can become a serious problem for the infrastructure and it remains an open problem how it can be reduced. Note that the frequency of synchronizations in a short time interval can actually be bounded by a trivial modification of the dynamic protocol: local conditions are only checked after a mini-batch of examples have been observed. Thus, the peak communication is upper bounded in the same way as with a periodic protocol, while still dynamically reducing the overall amount of communication.

In order to use dynamic averaging with kernel models, model compression has proven to be a crucial factor. Not only does storing and evaluating models with large numbers of support vectors can become infeasible (even in serial settings), but communicating the ever growing models renders the protocol inefficient. In a distributed setting, transmitting large models furthermore induce high communication costs, which is aggravated by averaging local models, because the synchronized model consists of the union of all local support vectors. **Which model compression is suitable for dynamic averaging with kernel models?** For the model truncation approach of Kivinen et al. (2004), the theoretical analysis has shown that the efficiency criterion is satisfied, but other model compression approaches might be favorable in certain scenarios. Thus, an interesting direction for future research is to study the relationship between loss and model size of those model compression techniques in order to extend the results on efficiency.

Also, alternative approaches to ensuring constant model size could be investigated. **What about finite dimensional approximations of kernel models?** A finite dimensional approximation of the feature map $\Phi : \mathcal{X} \rightarrow \mathcal{H}_k$ of a reproducing kernel Hilbert space $\mathcal{H}_k$, such as random Fourier features (Rahimi and Recht, 2007), allows to represent a kernel model as linear model. It remains an open problem how tight loss bounds combined with communication bounds can be derived in these settings. Moreover, initial experiments (see Figure 3.19) on a serial outlier detection task using random Fourier features shows that compression techniques (e.g., support vector truncation) perform consistently better with the same amount of memory usage.

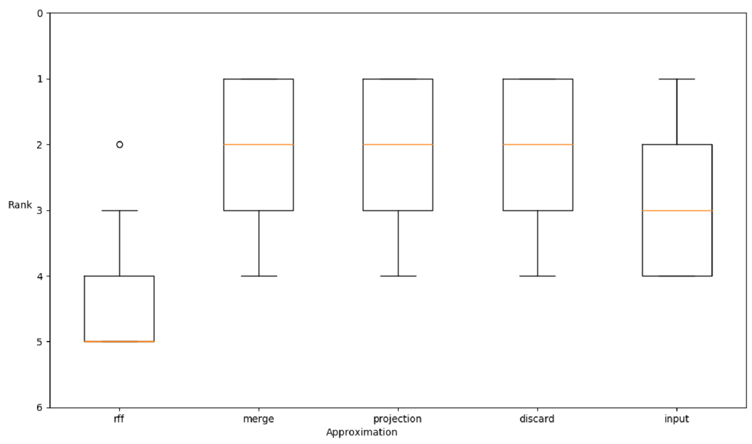


Figure 3.19.: Rank distribution of different approximation techniques over various outlier detection datasets and algorithms using the same memory budget for each model. Approximation techniques are random Fourier features (rff), support vector truncation (discard), projecting a support vector on the span of the remaining ones (projection), merging a new support vector with its nearest neighbor in feature space (merge), and in input space (input). The datasets used are outlier detection datasets from the UCI machine learning repository (Lichman, 2013) containing instances labels as inliers or outliers (adult, annthyroid, arrythmia, ionosphere, pima, wilt). Algorithms used are pegasos (Shalev-Shwartz et al., 2011), passive aggressive for uni-class prediction, and kernel stochastic gradient descent (Kivinen et al., 2004). The performance is measured using the F_1-score. The picture shows the ranking of the approximation technique over all combinations of datasets and algorithms.

Dynamic averaging is a partial synchronization operator using averaging as aggregation operator. **What about other aggregation operators?** As mentioned in Section 3.2, another central aggregation operator is the geometric median

$$f_{med} = \arg\min_{f' \in \mathcal{F}} \sum_{i=1}^{m} \|f^i - f'\|_2 \ ,$$

of a model configuration $\mathbf{f} = (f^1, \dots, f^m)$. The corresponding distance is the Euclidean distance and the divergence is given my

$$\delta(\mathbf{f}) = \sum_{i=1}^{m} \|f^i - f_{med}\|_2 \ .$$

It has the advantage that it is more robust to outliers and has been applied to online learning from noisy data (Feng et al., 2017). The theoretical analysis of the loss of dynamic averaging can be extended to the geometric median: The synchronization lemma (Lemma 3.13) only requires the synchronization to comply with Definition 3.3, the update lemma (Lemma 3.14) has no requirements on the synchronization. Thus, the regret bound in Theorem 3.19 holds for the geometric median as well, i.e., a partial synchronization operator using the geometric median retains the loss of a periodic one. However, it remains an open question whether periodically applying the geometric median is consistent. Even though the communication analysis holds for the geometric median as well, Theorem 3.24 requires it to be consistent in order to proof its adaptivity. Therefore, to show that the geometric median also leads to an efficient protocol, it has to be shown that periodically applying it leads to a consistent one.

The analysis of dynamic averaging constitutes a first example of a setting in which adaptivity and consistency can be achieved at the same time. The following chapter, it is applied to batch learning and analyzed in the ERM model.

4. Dynamic Distributed Batch Learning

After having introduced dynamic averaging for online learning, this chapter investigates its application to incremental algorithms for batch learning. The focus in batch learning is not on the loss suffered during training but on the performance of the final output of the learning algorithm on unseen data. Using the empirical risk minimization model introduced in Chapter 2 allows to give guarantees on this performance in the form of generalization bounds (see Section 2.1.3). Recall that for incremental learning algorithms these bounds guarantee that with probability $\delta \in (0,1]$ it holds for a model $f \in \mathcal{F}$ trained on a training set E drawn iid from a target distribution $\mathcal{D}$ that

$$\mathcal{L}_{\mathcal{D}}(f) \leq \min_{f' \in \mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') + \epsilon + \widehat{\epsilon} \ ,$$

where $\epsilon \in \mathbb{R}_+$ depends on δ, $\mathcal{F}$, and the size of the training set $N = |E|$. The optimization error $\widehat{\epsilon} \in \mathbb{R}_+$ is an additional error induced by the incremental learning algorithm, because it does not output the actual minimizer of the objective function but an approximation to it. This approximation gets better, and thus $\widehat{\epsilon}$ gets smaller, with the amount of rounds the algorithm is run. That the rate at which $\widehat{\epsilon}$ decreases is called the convergence rate.

Thus, instead of the cumulative loss this chapter analyzes the convergence rate of dynamic averaging. For linear and kernel models, standard incremental optimization algorithms used in batch learning (e.g., SGD and mini-batch SGD) perform regret-proportional convex updates. With this, the analysis for online regret in the previous chapter can be extended to the convergence rate of batch learning. For most neural networks, however, these algorithms do not perform convex updates anymore. For those networks I show that if the learning algorithm has a contraction property, a result on the convergence rate similar to that for linear and kernel models can be obtained. Using these two results, this chapter shows that dynamic averaging retains the convergence rate of SGD, mini-batch SGD, and GD for linear and kernel models, as well as neural networks.

Section 4.1 theoretically analyzes the convergence of dynamic averaging for linear and kernel models followed by an extended analysis for neural networks. Section 4.3 then discusses practical aspects of dynamic averaging. The protocol is empirically evaluated on various batch learning tasks, including autonomous driving, in Section 4.4. The chapter concludes with a discussion in Section 4.5.

4.1. Convergence Rate of Dynamic Averaging

The convergence rate of a distributed learning protocol should in the optimal case be similar to that of the base learning algorithm. To that end, this section will provide a convergence rate analysis for dynamic averaging that relates its convergence to that of periodic averaging. Similar to the online regret result from Chapter 3.6, if periodic averaging has an optimal convergence rate (i.e., the same as the base learning algorithm) then dynamic averaging has an optimal convergence rate for that base learning algorithm as well.

4.1.1. Relating the Convergence of Dynamic to Periodic Averaging

To relate the convergence of dynamic to periodic averaging, the proof for online regret from Section 3.3.2 is adapted to the batch setting. For this, the update lemma (Lemma 3.14) is extended from single examples per round to datasets.

Lemma 4.1. *Let the updates of an incremental learning algorithm $\mathcal{A}$ be regret-proportional convex updates with $\gamma > 0$. Then for all models $d, s \in \mathcal{F}$ and all datasets $E \subset \mathcal{X} \times \mathcal{Y}$ it holds that*

$$\|\mathcal{A}(E, d) - \mathcal{A}(E, s)\|^2 \leq \|d - s\|^2 - \gamma^2 \sum_{(x,y)\in E} (\ell(d, x, y) - \ell(s, x, y))^2 \ .$$

The proof is straight-forward and is provided in Appendix B.2.

Using this adapted update lemma together with the synchronization lemma (Lemma 3.13) from Section 3.3.2 it is possible to tie the convergence rate of dynamic averaging to that of periodic averaging. For that it is required that Δ_t is either static or decreasing with t.

Theorem 4.2. *Let $\mathcal{A}$ be a learning algorithm that performs regret-proportional convex updates with $\gamma > 0$ and a Lipschitz-continuous loss function ℓ with Lipschitz constant $\iota \in \mathbb{R}$. Let $\mathcal{D} = (\mathcal{A}, \overline{\sigma}_{\Delta,b}, \mathfrak{a}_{AVG}, m)$ the dynamic averaging protocol and $\mathcal{P} = (\mathcal{A}, \sigma_b, \mathfrak{a}_{AVG}, m)$ denote the periodic averaging protocol both maintaining models on the same distributed learning system with $m \in \mathbb{N}$ learners. In round $t \in \mathbb{N}$, each learner samples a dataset $E_t^i \subset \mathcal{X} \times \mathcal{Y}$ with $|E_t^i| \geq (\gamma^2\iota)^{-1}$. Let furthermore $\mathbf{d}_t = (d_t^1, \ldots, d_t^m) \in \mathcal{F}^m$ be a model configuration maintained by $\mathcal{D}$ at time $t \in \mathbb{N}$ using divergence threshold $\Delta_t \in \mathbb{R}$ with $\Delta_t \geq \Delta_{t+1}$ for all $t \in \mathbb{N}$ and $\mathbf{s}_t = (s_t^1, \ldots, s_t^m) \in \mathcal{F}^m$ be a model configuration maintained by $\mathcal{P}$. Then it holds that*

$$\frac{1}{m}\sum_{i=1}^{m} \|d_t^i - s_t^i\|^2 \leq \Delta_t \ .$$

Proof. Similar to the proof of Theorem 3.19 in Section 3.3.2, Lemma 3.13 and Lemma 4.1 can be combined to show that

$$
\begin{aligned}
\frac{1}{m}\sum_{i=1}^{m}\|d_t^i - s_t^i\|^2 &= \frac{1}{m}\sum_{i=1}^{m}\|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)^i - \sigma_b(\mathbf{s}_t)^i\|^2\\
&\underbrace{\leq}_{\text{Lemma 3.13}} \frac{1}{m}\sum_{i=1}^{m}\|d_t^i - s_t^i\|^2 + \Delta_t\\
&= \frac{1}{m}\|\mathcal{A}(E_{t-1}, d_{t-1}^i) - \mathcal{A}(E_{t-1}, s_{t-1}^i)\|^2 + \Delta_t\\
&\underbrace{\leq}_{\text{Lemma 3.14}} \frac{1}{m}\sum_{i=1}^{m}\|d_{t-1}^i - s_{t-1}^i\|^2 + \Delta_t\\
&\quad - \gamma^2\frac{1}{m}\sum_{i=1}^{m}\sum_{(x,y)\in E_{t-1}^i}\underbrace{\left(\ell(d_{t-1}^i, x, y) - \ell(s_{t-1}^i, x, y)\right)^2}_{\leq \iota\|d_{t-1}^i - s_{t-1}^i\|^2}\\
&\underbrace{\leq}_{\ell \text{ is } \iota\text{-Lipschitz}} \frac{1}{m}\sum_{i=1}^{m}\|d_{t-1}^i - s_{t-1}^i\|^2 + \Delta_t - \gamma^2\frac{1}{m}\sum_{i=1}^{m}\sum_{(x,y)\in E_{t-1}^i}\iota\|d_{t-1}^i - s_{t-1}^i\|^2\\
&= \left(1 - \underbrace{\gamma^2\iota\frac{1}{m}\sum_{i=1}^{m}|E_{t-1}^i|}_{\geq 1}\right)\frac{1}{m}\sum_{i=1}^{m}\|d_{t-1}^i - s_{t-1}^i\|^2 + \Delta_{t+1} \leq \Delta_t \quad .
\end{aligned}
$$

□

From Theorem 4.2 it follows that, using a decreasing divergence threshold $\Delta_t \to 0$, dynamic averaging achieves the same convergence rate as periodic averaging.

Theorem 4.2 holds for machine learning algorithms that perform regret-proportional convex updates, such as SGD, mini-batch SGD, and passive aggressive updates for linear models and kernel methods. However, for (deep) neural networks, these algorithms do not perform convex updates. Thus, the following section provides an alternative result that ties the convergence rate of dynamic averaging to periodic averaging for machine learning algorithms that fulfill a contraction criterion. It then shows that stochastic gradient descent for neural networks fulfills this contraction criterion.

4.1.2. Convergence of Dynamic Averaging for Deep Learning

Theorem 4.2 shows that for regret-proportional convex updates, dynamic averaging retains the convergence rate of periodic averaging. However, the result does not apply to deep learning. For example, a learning algorithm well-suited for training deep neural networks is stochastic gradient descent (Zhang et al., 2017), which is indeed performing regret-proportional convex updates for linear models and kernel methods, as shown above. However, for deep learning SGD does not perform convex updates: The weights of each layer are updated in the direction of the input to that layer. Only for the first layer this is in the direction of the training sample.

For all other layers, the update direction depends on the weights on the previous layer. Thus, even if the update is in the direction of a convex set, this set is not independent of the model. Thus SGD for deep learning does not fulfill Definition 3.8.

In the following it is shown that nonetheless dynamic averaging retains the convergence rate of periodic averaging using SGD for deep learning. For that the notion of contractions is required.

Definition 4.3. *An incremental learning algorithm $\mathcal{A}$ is a contraction with constant $c \in \mathbb{R}_+$ if for a dataset $E \subset \mathcal{X} \times \mathcal{Y}$ and models $f, f' \in \mathcal{F}$ it holds that*

$$\left\|\mathcal{A}(E,f) - \mathcal{A}(E,f')\right\| \leq c\left\|f - f'\right\| \ .$$

Using this notion of contraction the following theorem bounds the average loss difference between dynamic and periodic averaging.

Theorem 4.4. *Let ℓ be an ι-Lipschitz loss function and the incremental learning algorithm $\mathcal{A}$ be a contraction with constant $c \in \mathbb{R}$. Then, for batch sizes $b \geq \log_2^{-1} c^{-1}$ and divergence thresholds $\Delta \leq \epsilon(2\iota)^{-1}$, the average loss difference between using a partial synchronization operator $\overline{\sigma}_{\Delta,b}$ and using σ_b is bounded by ϵ, i.e., for all rounds $t \in \mathbb{N}$ and all examples $(x_t^i, y_t^i)_{i\in[m]} \subset \mathcal{X} \times \mathcal{Y}$ it holds that*

$$\frac{1}{m}\sum_{i=1}^m \left|\ell_t(d_t^i, x_t^i, y_t^i) - \ell_t(s_t^i, x_t^i, y_t^i)\right| \leq \epsilon \ ,$$

where d and s denote the models at learner i and time t maintained by $\overline{\sigma}_{\Delta,b}$ and σ_b, respectively.

Proof. The claim is proven within two steps. First note that a loss bound is induced by a bound on the average distances between pairs of models at the local learners because of the Lipschitz continuity of ℓ. Then it is shown that such a bound is retained between the local model pairs resulting from static and dynamic synchronization.

Using the Lipschitz continuity of ℓ, the loss difference at round t is bounded by

$$\left\|\ell_t(d_t^i, x_t^i, y_t^i) - \ell_t(s_t^i, x_t^i, y_t^i)\right\| \leq \iota\|d_t^i - s_t^i\| \ .$$

Hence, for the desired difference in convergence of ϵ it is sufficient to show that at all times $t \in \mathbb{N}$ it holds that the average pair-wise model distance at the local learners is bounded by $2\Delta = \epsilon/\iota$, i.e.,

$$\frac{1}{m}\sum_{l=1}^m \|d_t^i - s_t^i\| \leq 2\Delta \ . \tag{4.5}$$

In the following it is shown that Eq. (4.5) is retained throughout all rounds $t \in \mathbb{N}$. This is done by induction over t, assuming that for $t = 0$, all models are initialized identically. Before the first synchronization, i.e., for $t \leq b$, both weight sequences are identical and the bound, i.e, the induction hypothesis (IH), holds. Moreover, if $t - 1$ is not a synchronization step, i.e., $t - 1 \mod b \neq 0$, the bound is preserved for $\mathbf{d}_t$ and $\mathbf{s}_t$ due to $\mathcal{A}$ being a contraction. Hence, the crucial case is $t > b$ with $(t-1) \mod b = 0$. Using Lemma 3.13 yields that

$$\frac{1}{m}\sum_{i=1}^m \|\overline{\sigma}_{\Delta,b}(\mathbf{d}_t)^i - \sigma_b(\mathbf{s}_t)^i\|^2 \leq \underbrace{\frac{1}{m}\sum_{i=1}^m \|d_t^i - s_t^i\|^2}_{(*)} + \Delta \ .$$

Applying the contraction property of $\mathcal{A}$ on $(*)$ yields

$$\frac{1}{m}\sum_{i=1}^{m}\|d_t^i - s_t^i\|^2 \leq c^b \underbrace{\frac{1}{m}\sum_{i=1}^{m}\|d_{t-b}^i - s_t^i\|^2}_{\leq\Delta \text{ by IH}} \leq c^b\Delta + \Delta \leq 2\Delta \ .$$

The last inequality follows from the fact that $b \geq \log_2^{-1} c^{-1} = \log_c 1/2$. □

Assume that periodic averaging with learning algorithm $\mathcal{A}$ has a convergence rate in $\mathcal{O}(1/g(T))$, where $T \in \mathbb{N}$ denotes the number of rounds and $g : \mathbb{N} \to \mathbb{R}$ is an arbitrary function. Then it follows from Theorem 4.4 that dynamic averaging has a convergence rate in $\mathcal{O}(1/g(T) + \epsilon)$. By setting $\Delta_t = 1/g(t)$ in each round $t \in [T]$ it follows that $\epsilon \in \mathcal{O}(1/g(T))$ so that dynamic averaging has a convergence rate in $\mathcal{O}(1/g(T))$. That is, dynamic averaging retains the convergence rate of periodic averaging.

It remains to show that SGD is a contraction. For convex loss functions, one can show (Zinkevich et al., 2010) that SGD is a contraction for sufficiently small constant learning rates: for $\eta \leq (\rho\iota + \lambda)^{-1}$ the updates do contract with constant $c = 1 - \eta\lambda$. Here, $\eta, \lambda \in \mathbb{R}_+$ denote the learning rate and regularization parameter, $\iota \in \mathbb{R}_+$ the Lipschitz constant of the loss function, and $\rho \in \mathbb{R}_+$ the data radius (i.e., for all $x \in \mathcal{X}$ it holds that $\|x\|_2 \leq \rho$). Since this argument is independent of the actual gradient, it also holds for the gradient descent (GD) algorithm that computes the gradient over the entire dataset and the mini-batch SGD algorithm that computes the gradient over a mini-batch of examples. That is, both GD and mini-batch SGD are contractions, as well.

This also holds for the non-convex case if for any round $t \in [T]$ the loss function is locally convex in a bounded region around the span of the models in $\mathbf{d}_t$ and $\mathbf{s}_t$. Since the distance of all models in $\mathbf{d}_t$ to the models in $\mathbf{s}_t$ is bounded by 2Δ and the distance of all models in $\mathbf{d}_t$ to their average $\overline{d_t}$ is bounded by Δ, all models lie in a 3Δ bounded region around $\overline{d_t}$. If moreover the update magnitude is bounded by R, then all models remain in a $3\Delta R$ bounded region. Since the update magnitude for SGD is given by the norm of the gradient and the learning rate, the update magnitude can be bounded by $R = \eta\iota$. Thus for all models, if in any round $t \in \mathbb{N}$ the loss function is locally convex in a $3\Delta\eta\iota$-radius around $\overline{d_t}$, then SGD is a contraction. To see that this is a non-trivial but realistic assumption, see Sanghavi et al. (2017); Wang and Srebro (2017) on local convexity and Keskar et al. (2017); Nguyen and Hein (2017) on the loss surface of deep learning. It follows that under these assumptions, dynamic averaging with SGD retains the convergence rate of periodic averaging.

Corollary 4.6. *Let ℓ be an L-Lipschitz loss function, $\lambda \in \mathbb{R}$ a regularization parameter, $\eta \leq (\rho\iota + \lambda)^{-1}$ a learning rate, $\rho \in \mathbb{R}_+$ the data radius, $b \geq \log_2^{-1}(1-\eta\lambda)^{-1}$ the batch sizes and $\Delta \leq \epsilon/2\iota t$ the divergence threshold in round t $\in \mathbb{N}$. Let $\mathbf{d}_t$ and $\mathbf{s}_t$ denote the models maintained by dynamic and periodic averaging, respectively. If in all rounds $t \in \mathbb{N}$ the loss function is locally convex in a $3\Delta\eta\iota$-radius around $\overline{d_t}$, then dynamic averaging with stochastic gradient descent retains the convergence rate of periodic averaging.*

From Proposition 3.22 in Chapter 3.6 it follows that periodic averaging with $b = 1$ on $m \in \mathbb{N}$ learners for mini-batch SGD with mini-batch size $B \in \mathbb{N}$ and classic SGD has an optimal convergence rate—that is, it is equivalent to serial mini-batch SGD with a batch size mB. The convergence rate of serial mini-batch SGD is in $\mathcal{O}(1/BT)$ (Li et al., 2014). Thus, running periodic averaging on $m \in \mathbb{N}$ learners using mini-batch SGD with mini-batch size $B \in \mathbb{N}$ has a convergence rate in $\mathcal{O}(1/mBT)$. It follows that dynamic averaging with the same base learning algorithm has an optimal convergence rate, as well. For that it invests communication only when beneficial, effectively reducing the overall amount of communication load.

The empirical evaluation in Section 4.4 shows that in practice, dynamic averaging indeed achieves the same model quality with substantially less communication. The convergence result for neural networks (Corollary 4.6) requires a local convexity around the model in each round. This assumption holds, if all local models remain in the same local minimum during the entire training process. McMahan et al. (2017) argue that in practice this is often the case if all neural network models are initialized to the same initial model. This is investigated empirically in Section 4.4.2. The results in this thesis support the argument of McMahan et al. (2017), i.e., equal initialization leads to good convergence results, whereas a highly different initialization deteriorates it. At the same time, the results extend the findings of McMahan et al. (2017), because they also show that for periodic communication, having slight variations in the initialization is actually beneficial to the convergence.

After having analyzed that dynamic averaging retains optimal convergence rates (for certain algorithms), the next section analyzes its speedup in those cases.

4.2. Speedup of Dynamic Averaging

Assessing the speedup of dynamic averaging requires determining the runtime for a given incremental learning algorithm. That is, for a dynamic averaging protocol $\mathcal{D} = (\mathcal{A}, \overline{\sigma}_{\Delta,b}, \mathfrak{a}_{AVG}, m)$ with learning algorithm $\mathcal{A}$ on $m \in \mathbb{N}$ learners, with parameters $\Delta \in \mathbb{R}_+$ and $b \in \mathbb{N}$, its runtime $\mathcal{T}_{\mathcal{D}}(T)$ with respect to the runtime $\mathcal{T}_{\mathcal{A}}$ and the number of learners m needs to be determined. This runtime $\mathcal{T}_{\mathcal{D}}(T)$ for processing T rounds can be decomposed into the runtime $\mathcal{T}_{\mathcal{A}}$ of $\mathcal{A}$ in parallel on m learners, each processing T examples, and the time required for the synchronizations. For dynamic averaging (ignoring message passing times), each synchronization takes at most as much time as a full synchronization. Thus, we can estimate the actual time required for synchronization as the number of synchronizations times the time required for averaging all models.

Assuming a constant model size (i.e., linear models, neural networks, and kernel models with compression), the time for calculating the average of m models on a single machine is in $\Theta(m)$. Note that the time for calculating the average can be straight-forwardly reduced to $\Theta(\log m)$ by calculating the average in a map-reduce fashion without additional communication (see the example on periodic averaging in Section 2.3.3). The impact of the network topology on this runtime is discussed in the following section. For the remainder of this section, assume that the runtime of averaging is in $\Theta(\log m)$.

Let $S \in \mathbb{N}$ be the number of synchronizations until round T. Then, the runtime of dynamic averaging is

$$\mathcal{T}_{\mathcal{D}}(T) = \mathcal{T}_{\mathcal{A}}(T) + S \cdot \Theta(\log m) \ .$$

Note that since $S \leq T/b$, the runtime of dynamic averaging is upper bounded by the runtime of periodic averaging.

Let $T' \in \mathbb{N}$ denote the number of rounds required by the serial application of $\mathcal{A}$ to achieve the same model quality as $\mathcal{D}$ after T rounds. Then the serial application of $\mathcal{A}$ has runtime $\mathcal{T}_\mathcal{A}(T')$ and the speedup of dynamic averaging is

$$\frac{\mathcal{T}_\mathcal{A}(T')}{\mathcal{T}_\mathcal{D}(T)} = \frac{\mathcal{T}_\mathcal{A}(T')}{\mathcal{T}_\mathcal{A}(T) + S\Theta(\log m)} \enspace .$$

The previous section has shown that dynamic averaging retains the convergence rate of specific incremental machine learning algorithms. For such algorithms, $\Theta(T') = \Theta(Tm)$. Assuming that $\mathcal{A}$ has linear runtime, i.e., $\mathcal{T}_\mathcal{A}(T) \in \Theta(T)$, leads to a speedup in

$$\Theta\left(\frac{Tm}{T + S\log m}\right) \enspace .$$

In the worst case, $S = T/b \in \Theta(T)$ and thus the worst-case speedup is in

$$\Theta\left(\frac{m}{\log m}\right) \enspace , \tag{4.7}$$

similar to periodic averaging. Note that the number of processing units used is larger than m: In order to compute the average in time $\log_r m$, additional processors are required (see the MapReduce example in Section 2.2). The number of additional processors is equal to the amount of inner nodes of a perfect r-ary tree with m leafs which is ${}^{rm-1}\!/_{r-1} - m$, and thus the total number of processing units (including the learners) is ${}^{rm-1}\!/_{r-1}$. Since the speedup is usually given as a function of the total number of employed processing units, Equation 4.7 becomes

$$\frac{m}{\log_r m} = \frac{\frac{cr-c+1}{r}}{\log_r \frac{cr-c+1}{r}} \geq \frac{\frac{c}{r}}{\log_r \frac{cr}{r}} = \frac{c}{r\log_r c} \enspace .$$

In case $\mathcal{A}$ has polynomial runtime, i.e., $\mathcal{T}_\mathcal{A}(T) \in \Theta(T^\kappa)$ for $\kappa \in \mathbb{N}$, the speedup is even larger than for linear time algorithms. In this case, the speedup is in

$$\Theta\left(\frac{(Tm)^\kappa}{T^\kappa + m\log m}\right) = \Theta\left(m^\kappa\left(\frac{1}{1 + \frac{\log m}{T^{\kappa-1}}}\right)\right)$$

for $S \in \mathcal{O}(m)$.

Note that the speedup is influenced by the network topology used. If the star topology is used (as discussed in Section 3.4), the coordinator requires time $\Theta(m)$ to compute the average of all local models. This implies that the speedup does not grow with m. Using a hierarchical topology instead allows computing the average in a MapReduce fashion, requiring only time logarithmic in m which is necessary to achieve the speedup of $\Theta\left({}^{m}\!/_{\log m}\right)$. For the decentralized setup, the runtime for calculating the average depends on the actual topology. In the worst case, the network is (close to) a star topology and thus the runtime is in $\mathcal{O}(m)$. The closer the topology is to a perfect r-ary tree, the closer the runtime is to $\mathcal{O}(\log_r m)$. The same holds if the spanning tree implicitly generated by the model request is close to a perfect r-ary tree.

4.2.1. Efficient Parallel Runtime

As stated in the introductory chapter, a parallelization is considered to be efficient if it achieves polylogarithmic runtime on (quasi-)polynomially many processing units (see requirement R2 in Chapter 1). For that, assume for simplicity that in each round, each learner samples a new example iid from the target distribution. Then, dynamic averaging on $m \in \mathbb{N}$ learners after $T \in \mathbb{N}$ rounds observed $N = mT$ examples. The runtime of dynamic averaging in the input size N is in

$$\Theta\left(\frac{N}{m} + \frac{N}{m}\log m\right) \ .$$

If we could chose $m = T$ (which is even less than polynomial in N), then the runtime would be in $\Theta(\log N)$, i.e., polylogarithmic in N. However, m cannot be chosen arbitrary large: Optimal convergence has been shown for SGD and mini-batch SGD. However, these algorithms require that the learning rate in round $t \in \mathbb{N}$ is in $\mathcal{O}(1/\sqrt{t})$ in order to achieve their convergence rate. Averaging with m reduces the learning rate by a factor of $1/m$ (see Proposition 3.22). Thus, in order to obtain a similar model as the serial algorithm, the learning rate must be chosen m times larger. It follows that the number of learners has to be in $\mathcal{O}(\sqrt{T})$, because otherwise the learning rate is larger than $\mathcal{O}(1/\sqrt{t})$. Now, if m is in $\mathcal{O}(\sqrt{T})$, the runtime cannot be reduced to polylogarithmic in $N = mT$. Therefore, even though dynamic averaging allows for a large speedup per learner it does not achieve polylogarithmic runtime on (quasi-)polynomially many processing units.

After having analyzed the speedup of dynamic averaging, the following section discusses practical aspects, such as non-iid training data and different local training set sizes, as well as privacy aspects of black-box parallelizations.

4.3. Practical Aspects of Dynamic Averaging

Dynamic averaging can be applied to a wide range of learning algorithms. In practice, several aspects need to be addressed, such as non-iid training data and different sampling rates. On the other hand, black-box parallelizations do not require to share privacy-sensitive data. In the following, these practical aspects are discussed.

4.3.1. Federated Learning: Different Sampling Rates and Non-IID Data

In order to derive generalization bounds using the ERM model, local datasets are assumed to be drawn iid from a common target distribution. This does not necessarily hold in practice. Moreover, the data rate at each local learner, i.e., the size of the local datasets E_t^i, is assumed to be fixed. In contrast, McMahan et al. (2017) introduced **federated learning**, a distributed learning task with (i) non-iid data, (ii) unbalanced data sampling rates, (iii) massively distributed systems, and (iv) limited communication infrastructure. The section discusses to what extend dynamic averaging already fits to that problem setup and how it can be extended to better suit it.

Algorithm 5 Federated Averaging (McMahan et al., 2017)

```
Input: Number of epochs $e$, mini-batch size $B$, fraction $C$
Server executes:
 1. initialize $f_0$
 2. for each round $t = 1, 2, \dots$ do
 3.    $k \leftarrow \max\{Cm, 1\}$
 4.    $S_t \leftarrow$ random set of $k$ learners
 5.    for each learner $i \in S_t$ in parallel do do
 6.       $f^i_{t+1} \leftarrow ClientUpdate(i, f_t)$
 7.    end for
 8.    $N^i \leftarrow |E^i|$
 9.    $N \leftarrow \sum_{i_t \in S_t} N^i$
10.    $f_{t+1} \leftarrow \sum_{i \in S_t} \frac{N^i}{N} f^i_{t+1}$
11. end for
ClientUpdate$(i, f)$ //run on client $i$
  split $E^i$ into mini-batches $E^i_1, E^i_2, \dots$ of size $B$
  for each epoch from 1 to $e$ do
     for each mini-batch $E^i_j$ do
        $f \leftarrow \mathcal{A}^{\text{SGD}}(E^i_j, f)$
     end for
  end for
  return $f$ to the server
```

McMahan et al. (2017) propose a black-box approach for distributed deep learning to address these issues. This approach assumes a dedicated coordinator node that synchronizes locally trained neural networks and is termed **Federated Learning**. They propose a modified periodic averaging protocol, denoted **Federated Averaging** (FedAvg), that is able to handle massively distributed systems with limited communication infrastructure by communicating only the parameters of neural networks. The major difference to periodic averaging is that FedAvg only synchronizes a fraction $C \in (0, 1]$ of the learners. This reduces communication by C at the cost of a loss in predictive performance.

McMahan et al. (2017) found that in practice it is beneficial to train the local neural networks for multiple epochs on the local training data. Let all local learners have a local dataset of the same size n. Then locally running mini-batch SGD with a mini-batch size $B \in \mathbb{N}$ and training for $e \in \mathbb{N}$ epochs before synchronizing as suggested by McMahan et al. (2017) is equivalent to running periodic averaging with $b = \lceil n/B \rceil e$. The pseudo-code of FedAvg is given in Algorithm 5.

Both dynamic averaging and FedAvg may only synchronize a subset of learners. Still, FedAvg is not a partial synchronization operator as in Definition 3.3. While FedAvg fulfills condition (i) of the definition, because averaging only a subset of the local models leaves the global average invariant, it does not fulfill condition (ii): for any given threshold Δ, averaging a random subset does not guarantee that the divergence over all models is below Δ. For example, if the divergence before averaging is above Δ, the random sample could consist of a set of equal models. Then averaging them does not change the divergence, so it remains above Δ after synchronization.

Dynamic averaging addresses the issues of massively distributed systems and limited communication infrastructure. In the worst case it performs similar to periodic averaging, in the best case it has similar predictive performance with orders of magnitude less communication. As will be shown in the empirical evaluation in Section 4.4.2, it substantially outperforms FedAvg in terms of communication while achieving a similar predictive performance.

Regarding non-iid data, McMahan et al. (2017) provide empirical indications that differences in local data distributions do not significantly deteriorate the learning process. Note that by losing the iid assumption on the local datasets the ERM guarantees on the generalization error do not hold anymore. McMahan et al. (2017) show that FedAvg achieves a high predictive performance even if local datasets deviate strongly from the target distribution: In an experiment on MNIST (where the task is to classify images of hand-written digits), local datasets only contained examples from two out of ten digits. Still, FedAvg was able to train a model that is capable of classifying all digits with high accuracy. This result indicates that averaging models is robust to non-iid data. Thus, these results should hold for dynamic averaging as well. However, further research is required to substantiate this claim, both for FedAvg and dynamic averaging (cf. Smith et al. (2017)).

FedAvg tackles the problem of unbalanced sampling rates by weighting each local model with the amount of examples it has processed (line 10 of Algorithm 5). This approach can be used with dynamic averaging as well. For that, assume that each local learner $i \in [m]$ observes $n^i \in \mathbb{N}$ samples. Similar to FedAvg, a weighted average is used where the weight depends on the number of observed examples. Let $N = \sum_{i=1}^{m} n^i$ be the total number of samples observed in each round. Then, the weighted average of models is given by

$$\overline{\mathbf{f}} = \frac{1}{N}\sum_{i=1}^{m} n^i f^i \quad .$$

Note that this can be generalized analogously to time-dependent sampling rates n_t^i. Using this weighted average, dynamic averaging for unbalanced data is given in Algorithm 6. Since black-box parallelizations like FedAvg and dynamic averaging only share models, they do not directly share privacy-sensitive data. In the following, this aspect is analyzed in more detail.

4.3.2. Privacy Aspects of Black-Box Parallelizations

Sharing only models and not data between learners substantially reduces the privacy risks (McMahan et al., 2017). However, the level of privacy depends on many factors, including the model class and distributed learning protocol.

For linear models it is possible to reconstruct the local gradients from the set of local models, if learners are synchronized every round (Yan et al., 2013). From these gradients, local data can be partially reconstructed. To see that, consider the following simple example. Assume a classification task from d-dimension binary vectors, i.e., $\mathcal{X} = \{0,1\}^d$ and $\mathcal{Y} = \{-1,1\}$. Using

Algorithm 6 Dynamic Averaging Protocol for Unbalanced Data

```
Input: divergence threshold $\Delta$, batch size $b$
Initialization:
    local models $f_1^1, \dots, f_1^m \leftarrow$ one random $f$
    reference vector $r \leftarrow f$
    violation counter $v \leftarrow 0$
Round $t$ at node $i$:
    observe $E_t^i \subset \mathcal{X} \times \mathcal{Y}$ with $|E_t^i| = n^i$
    update $f_{t-1}^i$ using the learning algorithm $\varphi$
    if $t \mod b = 0$ and $\|f_t^i - r\|^2 > \Delta$ then
        send $f_t^i$ and $n^i$ to coordinator (violation)
    end if
At coordinator on violation:
    let $\mathcal{B}$ be the set of nodes with violation
    $v \leftarrow v + |\mathcal{B}|$
    if $v = m$ then $\mathcal{B} \leftarrow [m]$, $v \leftarrow 0$
    $N \leftarrow \sum_{i \in \mathcal{B}} n^i$
    while $\mathcal{B} \neq [m]$ and $\left\| \frac{1}{N} \sum_{i \in \mathcal{B}} n^i f_t^i - r \right\|^2 > \Delta$ do
        augment $\mathcal{B}$ by augmentation strategy
        receive models from nodes added to $\mathcal{B}$
        $N \leftarrow \sum_{i \in \mathcal{B}} n^i$
    end while
    send model $\overline{\mathbf{f}} = \frac{1}{N} \sum_{i \in \mathcal{B}} n^i f_t^i$ to nodes in $\mathcal{B}$
    if $\mathcal{B} = [m]$ also set new reference vector $r \leftarrow \overline{\mathbf{f}}$
```

the continuous averaging protocol $\mathcal{C} = (\mathcal{A}^{\text{SGD}}, \sigma_1, \mathfrak{a}_{AVG}, m)$ with SGD allows to reconstruct local data if local models are known to an attacker. To see this, the following result shows how to reconstruct a binary vector from a model update.

Proposition 4.8. *Assume stochastic gradient descent with hinge loss ℓ_{hinge} is applied to a learning task with $\mathcal{X} = \{0,1\}^d$ and $\mathcal{Y} = \{-1,1\}$ using linear models. If in round $t \in \mathbb{N}$ an attacker has access to f_t and f_{t+1} and $f_t \neq f_{t+1}$, then she can reconstruct $x_t \in \mathcal{X}$ and $y_t \in \mathcal{Y}$.*

Proof. The gradient of the hinge loss is given by

$$\nabla_f \ell_{\text{hinge}}(f,x,y) = \nabla_f \max\{0, 1 - y f(x)\} \underbrace{=}_{f \text{ is linear model}} \nabla_f \max\{0, 1 - y\langle f, x\rangle\} \ .$$

Since $f_t \neq f_{t+1}$ it follows that

$$\nabla_f \ell_{\text{hinge}}(f,x,y) = \nabla_f (1 - y\langle f, x\rangle) = -yx \ .$$

Since SGD is used as learning algorithm with some learning rate $\eta \in \mathbb{R}_+$ it follows that

$$
\begin{aligned}
&f_{t+1} = f_t - \eta \nabla_f \ell_{\text{hinge}}(f, x, y) = f_t + \eta y x \\
\Leftrightarrow\ & x = \frac{1}{\eta y}(f_{t+1} - f_t) \ .
\end{aligned}
$$

Since $y \in \{-1, 1\}$ and $\eta > 0$ it holds for the j-th component of x that $x_j = 0 \Leftrightarrow (f_{t+1} - f_t)_j = 0$. Similarly $x_j = 1 \Leftrightarrow (f_{t+1} - f_t)_j \neq 0$. Thus, x can be reconstructed. Since $\eta > 0$ and $x \geq 0$ it also follows that if for all components $j \in [d]$ it holds that $(f_{t+1} - f_t)_j > 0$ then $y = 1$. Otherwise, if $(f_{t+1} - f_t)_j < 0$ then $y = -1$. □

From this it follows that if an attacker knows the model configuration in a distributed learning system for two consecutive rounds, she can reconstruct local data.

Corollary 4.9. *Assume a distributed learning system running stochastic gradient descent with hinge loss ℓ_{hinge} on $\mathcal{X} = \{0, 1\}^d$ and $\mathcal{Y} = \{-1, 1\}$. If in rounds t and $t + 1$ an attacker has access to the model configuration $\mathbf{f}_t$ and $\mathbf{f}_{t+1}$, then for every learner $i \in [m]$ with $f^i_{t+1} \neq f^i_t$, the attacker can reconstruct the local example (x^i_t, y^i_t).*

Note that this result also holds for a subset of the model configuration. Using the continuous averaging protocol with a dedicated coordinator node, an attacker that infiltrates the coordinator has access to the entire model configuration. Thus, she can reconstruct local data as in Proposition 4.8.

Corollary 4.10. *Let $\mathcal{C} = (\mathcal{A}^{SGD}, \sigma_1, \mathfrak{a}_{AVG}, m)$ be executed in a network topology with dedicated coordinator node. If in rounds t and $t + 1$ an attacker has access to the coordinator node, she can reconstruct the local example*

$$(x^i_t, y^i_t)$$

at learner i if $f^i_{t+1} \neq f^i_t$.

If instead of SGD a mini-batch SGD is used, or the attacker only knows f_t and f_{t+b} for some $b > 1$ (e.g., because periodic averaging with $b > 1$ is used), then the local data cannot be reconstructed in this manner. For kernel models, the problem of inferring local data is even larger since the model itself consists of data instances (the support vectors) and thus sharing the model is inherently non-private.

It follows from this example that in certain scenarios it does not suffice to protect the local data, it is also necessary to protect the local updates (e.g., the gradients in case of SGD). However, gradients can be reconstructed even if the attacker does not have access to a dedicated coordinator node but only a subset of learners, or a partial average (Yan et al., 2013). These privacy issues can be tackled by randomly perturbing models, or model updates, e.g., by adding zero-mean noise to the gradient. Chaudhuri et al. (2011) have shown that adding zero-mean noise to the output of an empirical risk minimization algorithm achieves ϵ_p-**differential privacy**. This notion of privacy demands that for two datasets only differing in a single example, the outputs of the learning algorithm on the two datasets do not differ much.

Definition 4.11 (Dwork et al. (2006)). *An algorithm $\mathcal{A}\colon \mathcal{X} \to \mathcal{F}$ provides ϵ_p-differential privacy if for any two datasets $E, E' \subseteq \mathcal{X}$ that differ in a single element and for any set $S \subseteq \mathcal{F}$ it holds that*

$$e^{-\epsilon_p} P\left(\mathcal{A}(E') \in S\right) \leq P\left(\mathcal{A}(E) \in S\right) \leq e^{\epsilon_p} P\left(\mathcal{A}(E') \in S\right) \quad .$$

In particular, adding a noise vector $v \in \mathcal{F}$ to the output $\mathcal{A}(E)$ of an empirical risk minimization algorithm $\mathcal{A}$ with density

$$p(v) = e^{-\frac{\lambda|E|\epsilon_p}{2}\|v\|_2}$$

achieves ϵ_p-differential privacy without changing the order of the generalization error of the resulting model. Here $\lambda \in \mathbb{R}_+$ is the regularization parameter of the regularized ERM objective. Similarly, adding noise to the gradient of SGD achieves ϵ_p-differential privacy when training neural networks (Abadi et al., 2016).

Adding noise to the data also allows to achieve ϵ_p-differential privacy: Balcan et al. (2012) have shown that using so-called statistical queries instead of standard access to the local dataset guarantees that the model obtained is ϵ_p-differentially private. Such statistical queries can be simulated by input noise (Kearns, 1998).

Another way of tackling the privacy issues is by computing the average in a privacy-preserving manner, i.e., using secure multi-party computation protocols (cf. (Kairouz et al., 2016; Lindell, 2005)). They guarantee that an attacker can at most learn about the model and data of a learner it has gained access to.

These techniques can be combined with dynamic averaging. Especially adding input noise or adding noise to local models can be straight-forwardly applied. Using secure multi-party computation to compute the average, however, requires a change in the communication protocol and leads to an increased amount of communication.

After having discussed these practical issues, the following section empirically evaluates dynamic averaging. This includes a comparison with periodic averaging, as well as FedAvg, and an evaluation on a real-world use-case from autonomous driving.

4.4. Empirical Evaluation

This section empirically evaluates dynamic averaging and compares it to periodic averaging, as well as two baselines: the serial execution of the base learning algorithm on the dataset, again denoted serial, and a distributed learning protocol that does not communicate at all, denoted nosync. In order to control the execution and exclude side effects of a real distributed system, the experiments are conducted in the simulated environment used in Chapter 3.

In order to analyze the convergence of dynamic averaging with respect to the amount of communication it uses, a set of experiments using linear models, kernel models, and neural networks is applied to multiple datasets. As a proxy for the convergence rate the cumulative error is reported—its slope indicates convergence and the final value allows to compare the convergence rates: the higher the final value, the slower the convergence. For each experiment, the cumulative error is contrasted to the cumulative communication.

As an estimate of the generalization error of the resulting model, the predictive performance on an independent test set with respect to the amount of communication is depicted. Moreover, the development over time of these measures is analyzed in detail.

As an example for a real-world application that fits naturally to dynamic averaging, **in-fleet learning** of autonomous driving functionalities is investigated. In this scenario, each vehicle constitutes a learner and the goal is to jointly learn a driving functionality, such as predicting the steering angle from front-camera images. In this scenario, concept drifts occur naturally, since properties central for the modeling task may change—changing traffic behavior both over time and different countries or regions introduce constant and unforeseeable concept drifts. Moreover, large high-frequency data streams generated by multiple sensors per vehicle renders data centralization prohibitive in large fleets.

4.4.1. Linear and Kernel Models

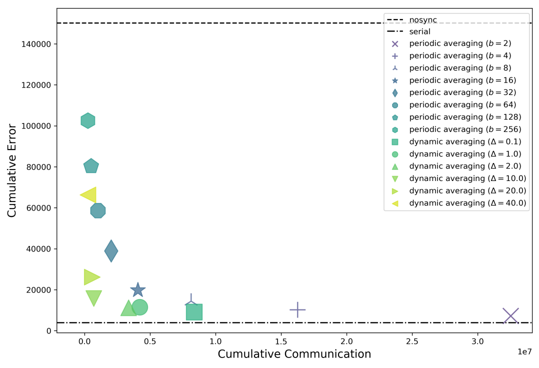
In order to study the convergence and generalization error of dynamic averaging, it is first compared to periodic averaging on the synthetic disjunctions dataset (see Section 3.5.1). Again, a random disjunction over $d = 50$ literals is drawn randomly at the beginning of the learning process, and in order to have balanced classes, the disjunctions as well as the samples $x \in \mathcal{X} = \{0,1\}^d$ are generated such that each coordinate is set independently to 1 with probability $\sqrt{1-2^{-1/d}}$, and the label is set to $y = 1$ if the disjunction over the sample is true and $y = -1$ otherwise. The prediction error of a model is measured by its zero-one-loss, i.e.,

$$\ell_{01}(f,x,y) = \begin{cases} 0, & \textbf{if } f(x) = y_i \\ 1, & \textbf{else} \end{cases} .$$

The model space $\mathcal{F} = \mathbb{R}^d$ is the space of linear models in $\mathbb{R}^d$. Recall that the model $f \in \mathcal{F}$ is a function $f : \mathcal{X} \to \mathcal{Y}$ that is identified with its representation as a vector in $\mathbb{R}^d$. The function value is given by (the sign of) the inner product of the vector with the sample $x \in \mathcal{X}$.

The incremental learning algorithm used in the experiments is stochastic gradient descent (SGD) $\mathcal{A}^{\text{SGD}}$ with learning rate $\eta \in \mathbb{R}_+$ and hinge loss. For the disjunction experiment, $m = 128$ learners ran for $T = 10\,000$ rounds using the distributed learning protocols periodic averaging with $b \in \{2,4,8,16,32,64,128,256\}$ and dynamic averaging with $b = 2$ and $\Delta \in \{0.1, 1.0, 2.0, 10.0, 20.0, 40.0\}$, as well as the nosync and the serial baseline.

Figure 4.1(a) shows the cumulative training error in contrast to the cumulative communication, with serial having the best predictive performance and nosync having the worst. The instances of periodic averaging vary in their trade-off between predictive performance and communication depending on the batch size b (see Table 4.1 for details). Protocols that communicate less than every $b = 16$ rounds have a high cumulative error, with $b = 256$ being close to nosync already. Periodic averaging which communicates more, i.e., with $b = 16$ and $b = 8$, achieves a low cumulative error (a zero one loss of less than 20.000 on a total of 1 280 000 means that roughly 1.5% of examples are misclassified, i.e., a training accuracy of 98.5%). The training performance gets even closer to that of the serial baseline (which misclassifies only 3 877 examples and thus has a training accuracy of 99.7%) with lower batch sizes, up to $b = 2$ which achieves a training accuracy of 99.4%. This comes at the cost of a comparatively huge amount of communication: Periodic averaging with $b = 2$ sends a total of 640 000 mes-



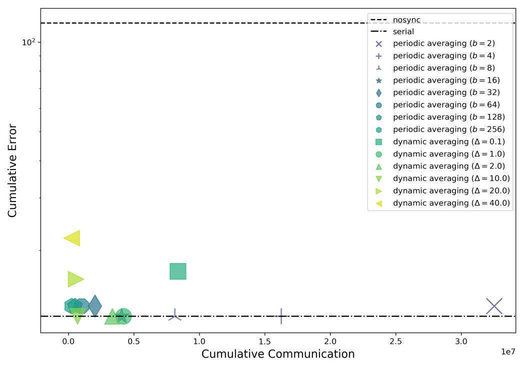


Figure 4.1.: Cumulative error (zero-one-loss) and cumulative communication of several instances of periodic and dynamic averaging, as well as the serial and nosync baseline. Figure (a) shows the cumulative training error after $T = 10\,000$ rounds of training (i.e., each instance processing 1 280 000 examples), Figure (b) shows the cumulative error of the final model f_T on an independent test set of size 10 000.

sages, each consisting of a 50-dimensional vector, resulting in a cumulative communication of $122MB$. In contrast, periodic averaging with $b = 16$ achieves a training accuracy of 98.5% with only 80 000 messages and thus a cumulative communication of only $15MB$.

At the same time, dynamic averaging allows to achieve similar results to periodic averaging with substantially less communication. Dynamic averaging with $\Delta = 10.0$ achieves a training error of $15,782$ (i.e., an accuracy of 98.8%)—which is even better than periodic averaging with $b = 16$—with only 13 707 messages, that is roughly 6 times less communication. With a similar amount of communication (dynamic averaging with $\Delta = 1.0$ uses 82 592 messages), dynamic averaging achieves an error of $11,466$ (that is 42% less error compared to periodic averaging with $b = 16$). Comparing periodic averaging with $b = 2$ to dynamic averaging with $\Delta = 1.0$ shows that, while periodic averaging with $b = 2$ achieves a training error of 7 287 using 640 000 messages, dynamic averaging with $\Delta = 0.1$ achieves a slightly higher training error of $9,099$, but to achieve this it only requires 164.224. That is roughly 4 times less communication.

Thus, for every setup of periodic averaging (associated with a specific trade-off between error and communication) a setup of dynamic averaging can be found that achieves a comparable error with substantially less communication. These results also translate to the generalization error of the final model obtained after training for $T = 10.000$ rounds. Figure 4.1(b) shows that all protocols, except for nosync achieve a very high test accuracy (over 99% accuracy, with dynamic averaging with $\Delta = 40.0$ being the worst with an accuracy of 99.56%). Such high accuracies are to be expected, since the synthetic dataset was designed to be easily learnable by linear models—even the nosync baseline achieves a test accuracy of 97.68%.

The convergence of the protocols in relation to the employed amount of communication over time is detailed in Figure 4.2. The cumulative error of periodic averaging increases substantially in the beginning of the learning process, even more so the less the protocols communicate (see Figure 4.2(a)). By investing substantially more communication in the

Protocol	Cumulative Error	Total Messages	Total Comm.
serial	3 877	1 280 000	$244.14MB$
nosync	150 200	0	$0MB$
periodic averaging (b = 2)	7 287	640 000	$122.07MB$
periodic averaging (b = 4)	10 247	320 000	$61.04MB$
periodic averaging (b = 8)	14 472	160 000	$30.52MB$
periodic averaging (b = 16)	19 798	80 000	$15.26MB$
periodic averaging (b = 32)	38 955	40 000	$7.63MB$
periodic averaging (b = 64)	58 659	20 000	$3.82MB$
periodic averaging (b = 128)	80 305	10 000	$1.92MB$
periodic averaging (b = 256)	102 543	5 000	$0.95MB$
dynamic averaging (Δ = 0.1)	9 099	164224	$31.32MB$
dynamic averaging (Δ = 1.0)	11 466	82592	$15.75MB$
dynamic averaging (Δ = 2.0)	11 229	65840	$12.56MB$
dynamic averaging (Δ = 10.0)	15 782	13707	$2.61MB$
dynamic averaging (Δ = 20.0)	26 187	11491	$2.19MB$
dynamic averaging (Δ = 40.0)	66 301	5086	$0.97MB$

Table 4.1.: Cumulative training zero-one-loss and and cumulative communication of the disjunction experiment.

beginning (see Figure 4.2(b)), the dynamic averaging variants suffer less loss in that phase compared to periodic averaging, resulting in an overall lower cumulative loss. This is achieved by investing substantially more communication in the beginning. As soon as the models converge, dynamic averaging reduces communication substantially, so that the overall amount of communication is low. It should be noted though that dynamic averaging with large Δ tends to not synchronize, even though it might have been beneficial, as indicated by the sudden jumps in cumulative error during the later training process.

The results from the disjunction experiment suggest that indeed dynamic averaging achieves a substantial improvement both in terms of convergence and communication by investing communication when necessary. In order to test whether these results hold for more challenging datasets as well, the protocols are evaluated on the real-world dataset **SUSY** (Baldi et al., 2014) from the UCI machine learning repository (Lichman, 2013). The dataset consists of features derived from a Monte Carlo simulation of the sensor values of particle detectors for a particle collision within the LHC experiment at CERN. The label for each collision indicates whether a supersymmetric particle has been produced in that collision or not. The dataset contains 4 999 999 examples with d = 18 features, which have been normalized to zero mean and unit variance for this experiment.

Figure 4.3 shows the results of several periodic and dynamic averaging protocols with m = 10 learners after T = 5 000 rounds. The training is lower the more communication is invested with dynamic averaging being more communication-efficient. The performance on an independent

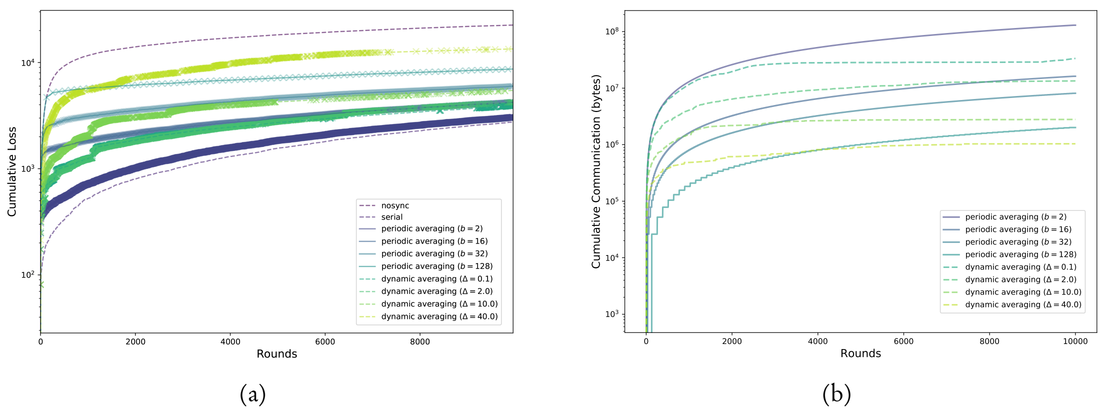


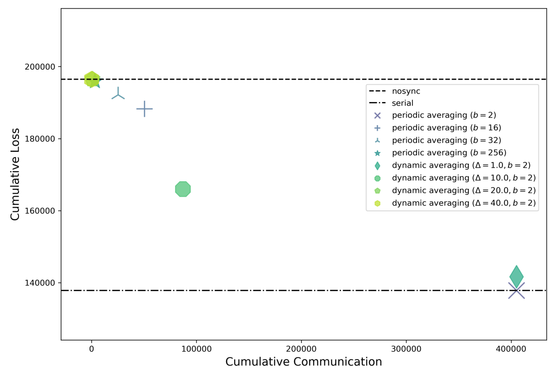
Figure 4.2.: Cumulative error (zero-one-loss) in log-scale over time (a) and cumulative communication in log-scale over time (b) of the disjunction experiment for a subset of the protocols.

test set of size 5 000 shows that dynamic averaging results in better models that periodic averaging with less communication. Surprisingly, the models obtained by periodic averaging with $b = 32$, as well as dynamic averaging with $\Delta = 10$ and $\Delta = 20$ achieve better performance than the serial baseline.

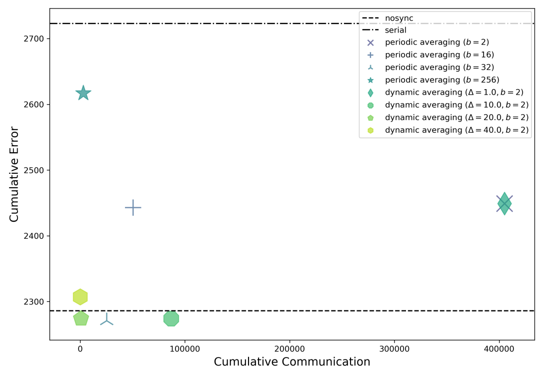
In the following it is investigated, whether kernel methods achieve higher prediction quality and how large the additional communication is. Figure 4.4 shows the results on the same SUSY dataset with the same number of learners, rounds, and the same protocols. The training error of all protocols shown in Figure 4.4(a) is similar, with dynamic averaging with $\Delta = 40$ and $\Delta = 1.0$, as well as periodic averaging with $b = 2$ achieving a performance comparable to serial learning. Moreover, the error is an order of magnitude smaller than that of linear models. On the test set, dynamic averaging with $\Delta \in \{1, 10, 40\}$ achieves lower error than all periodic protocols, for $\Delta = 10$ it performs even better than the serial baseline. Moreover, its error is only half of the error of the best performing protocol with linear models. This comes at the price of an amount of communication that is two orders of magnitude higher than that for linear models. As discussed in Chapter 3 the reason for the high amount of communication is the model size, which is up to linear in the dataset size. Another model class that is able to model non-linear dependencies is the class of neural networks. Even though those networks have a larger amount of parameters than linear models, the amount is fixed and thus the communication does not depend on the dataset size. The following section investigates dynamic averaging for neural networks.

4.4.2. Dynamic Averaging for Training Deep Neural Networks

After having studied dynamic averaging for batch learning in convex settings, it is now evaluated for the non-convex setting of training neural networks. To emphasize the theoretical result from Section 4.1, it is shown that dynamic averaging indeed retains the performance of periodic averaging with substantially less communication. This is followed by a comparison



(a)



(b)

Figure 4.3.: Cumulative error (zero-one-loss) and cumulative communication with $m = 10$ learners on the SUSY dataset using linear models. Figure (a) shows results $T = 5\,000$ rounds of training (i.e., each instance processing 50 000 examples), Figure (b) shows the cumulative error of the final model f_T on an independent test set of size 5 000.

of the approach with Federated Averaging (FedAvg) which poses a state-of-the-art communication approach. The protocol to is then evaluated in a realistic application scenario from autonomous driving[1].

Throughout this section, if not specified separately, we consider mini-batch SGD $\mathcal{A}^{\text{mSGD}}_{B,\eta}$ as learning algorithm, since recent studies indicate that it is particularly suited for training deep neural networks (Zhang et al., 2017). That is, we consider communication protocols $\Pi = (\mathcal{A}^{\text{mSGD}}_{B,\eta}, \sigma, \mathfrak{a}_{AVG}, m)$ with various synchronization operators σ, averaging $\mathfrak{a}_{AVG}$ as aggregation, and a number of learners $m \in \mathbb{N}$. The hyper-parameters of the protocols and the mini-batch SGD have been optimized on an independent dataset.

To evaluate the performance of dynamic averaging in deep learning, it is first compared to periodic averaging for training a convolutional neural network (CNN) on the MNIST classification dataset (LeCun, 1998). The chosen architecture of a network is a simple convolutional network with two convolutional layers, a max pooling layer, and two dense layers with final softmax activation. The overview of the network layers and the amount of

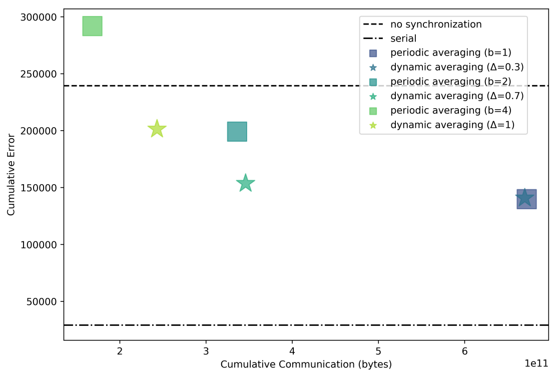
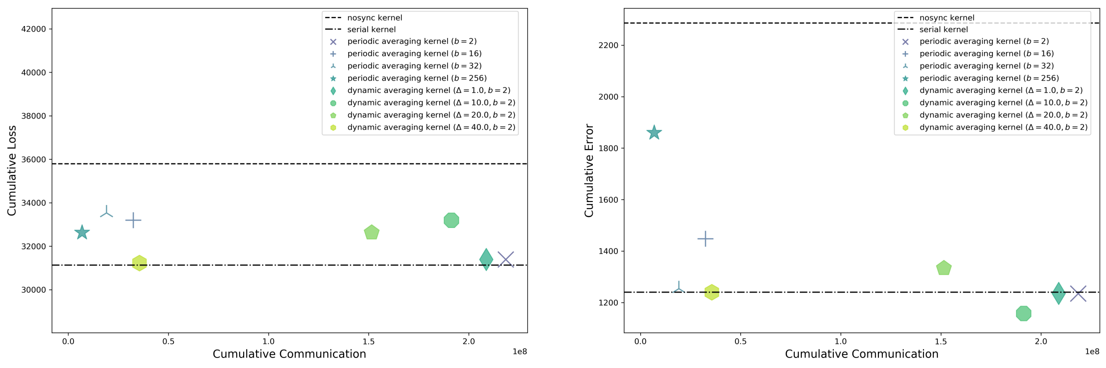


Figure 4.5.: Cumulative loss and communication of distributed learning protocols with $m = 100$ learners with mini-batch size $B = 10$, each observing $T = 14000$ samples (corresponding to 20 epochs for the serial baseline).

[1] The code of the experiments is available at `https://bitbucket.org/Michael_Kamp/decentralized-machine-learning`.



(a) (b)

Figure 4.4.: Cumulative error (zero-one-loss) and cumulative communication with $m = 10$ learners on the SUSY dataset using kernel models with Gaussian kernel ($\sigma = 2.4$). Figure (a) shows results $T = 5\,000$ rounds of training, Figure (b) shows the cumulative error of the final model f_T on an independent test set of size $5\,000$.

weights are described in the Table 4.3. The employed loss function is categorical cross-entropy:

$$H_y(\widehat{y}) := -\sum_i y_i \log(\widehat{y}_i)$$

where $\widehat{y}_i$ is the predicted probability for the class i and y_i is 1 if the correct class is i and 0 otherwise. The optimal parameters are a batch size of $B = 10$ samples and learning rate $\eta = 0.1$ for the serial learner. For the local learners the optimal learning rate is $\eta = 0.25$. The experiment is performed in a setup with $m = 100$ learners.

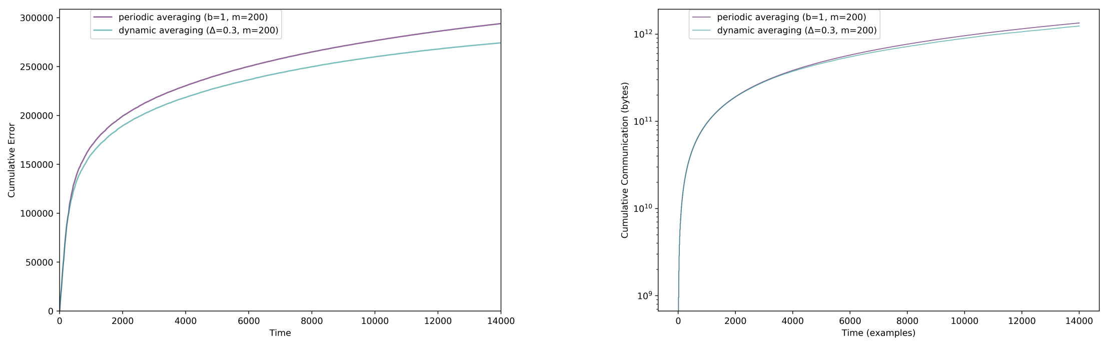


Figure 4.6.: (a) The cumulative loss development during the training on MNIST dataset for 40 epochs with one dynamic and one periodic protocols. (b) The cumulative communication development during the training on MNIST dataset for 40 epochs using dynamic averaging ($\Delta = 0.3, b = 1$) and periodic averaging ($b = 1$).

protocol configurations	
type	**parameters**
	$b = 1$
periodic protocol (σ_b)	$b = 2$
	$b = 4$
	$b = 1, \Delta = 0.3$
dynamic protocol ($\sigma_{\Delta,b}$)	$b = 1, \Delta = 0.7$
	$b = 1, \Delta = 1.0$

Table 4.2.: Overview of the different communication protocol configurations used for MNIST experiment. Except for protocol parameters Δ and b, all other parameters are kept constant between configurations.

Layer Type	Output Shape	#Weights
Conv2D	(None, 26, 26, 32)	320
Conv2D	(None, 24, 24, 64)	18,496
MaxPooling2D	(None, 12, 12, 64)	0
Dropout	(None, 12, 12, 64)	0
Flatten	(None, 9216)	0
Dense	(None, 128)	1,179,776
Dropout	(None, 128)	0
Dense	(None, 10)	1,290
	Total	1,199,882

Table 4.3.: The architecture of the Convolutional Neural Network used for MNIST dataset. Printout of the parameters made via Keras library.

Figure 4.5 shows the cumulative error of several setups of dynamic and periodic averaging, as well as the nosync and serial baselines. In this experiment the amount of examples shown to each of $m = 100$ learners was fixed to 14 000. Thus the amount of epochs that serial baseline model was trained for is 20, since the overall size of the dataset is 70 000. All the considered protocols can be seen from Table 4.2. The experiment confirms that for each setup of the periodic averaging protocol a setup of dynamic averaging can be found that reaches a similar predictive performance with substantially less communication (e.g., a dynamic protocol with $\sigma_{\Delta=0.7}$ reaches a performance comparable to a periodic protocol with $\sigma_{b=1}$ using only half of the communication). The more learners communicate, the lower their cumulative loss, with the serial baseline performing the best.

The advantage of the dynamic protocols over the periodic ones in terms of communication is in accordance with the convex case. For large synchronization periods, however, synchronizing protocols ($\sigma_{b=4}$) have even larger cumulative loss than the nosync baseline. This behavior

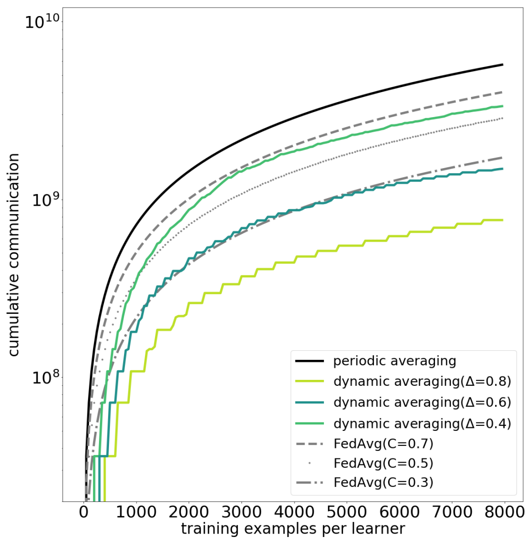


Figure 4.7.: Evolution of cumulative communication for different dynamic averaging and FedAvg protocols on $m = 30$ learners using a mini-batch size $B = 10$.

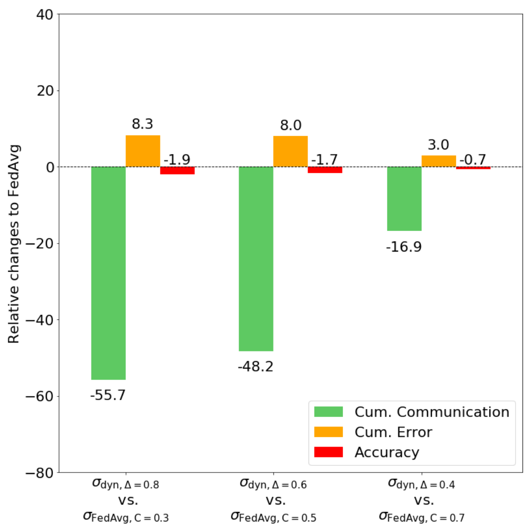


Figure 4.8.: Comparison of the best performing settings of the dynamic averaging protocol with their FedAvg counterparts.

cannot happen in the convex case, where averaging is always superior to not synchronizing. In contrast, in the non-convex case this behavior can be explained by the convergence of the local models to different local minima. Then their average might have a higher loss value than each one of the local models (recall Figure 4.14 for an illustration).

Figure 4.6 gives further details on the experiment, showing the development over time of the cumulative loss and the cumulative communication for $\sigma_{\Delta=0.3}$ and $\sigma_{b=1}$ in the setup with $m = 200$ learners. In the beginning, dynamic averaging invests a lot of communication (Figure 4.6(b)), but once the learners are trained on a larger amount of examples the dynamic protocol does not require much communication anymore keeping almost the same level of cumulative loss (Figure 4.6(a)).

Comparison of the Dynamic Averaging Protocol with FedAvg

Having shown that dynamic averaging outperforms standard periodic averaging, it is now compared to a highly communication-efficient variant of periodic averaging, denoted **FedAvg** (McMahan et al., 2017) (see Section 4.3.1), which poses a state-of-the-art for decentralized deep learning under communication-cost constraints—more recent approaches are interesting from a theoretical perspective but show no practical improvement (Jiang et al., 2017), or tackle other aspects of federated learning, such as non-iid data (Smith et al., 2017) or privacy aspects (McMahan et al., 2018).

Using the terminology of this thesis, FedAvg is a periodic averaging protocol that uses only a randomly sampled subset of nodes in each communication round. This subsampling leads to a reduction of total communication by a constant factor compared to standard periodic averaging. In order to compare dynamic averaging to FedAvg, the MNIST classification is repeated using CNNs and multiple configurations of dynamic averaging and FedAvg.

Table 4.4 shows the settings of dynamic averaging and FedAvg used for their comparison. The experiment runs with $m = 30$ learners, mini-batch size $B = 10$ and $b = 5$. The FedAvg parameter $C \in (0, 1]$ McMahan et al. (2017) is the fraction of learners involved in a particular model synchronization and their parameter E, the number of local batches, corresponds to the parameter b. Each learner is trained on 8000 training examples. Accuracy is calculated on the last 100 training examples.

protocol configurations	
type	**parameters**
periodic protocol (σ_b)	$b = 5$
dynamic protocol ($\sigma_{\Delta,b}$)	$b = 5, \Delta = 0.1$
	$b = 5, \Delta = 0.2$
	$b = 5, \Delta = 0.4$
	$b = 5, \Delta = 0.6$
	$b = 5, \Delta = 0.8$
FedAvg (σ_{FedAvg})	$b = 5, C = 0.3$
	$b = 5, C = 0.5$
	$b = 5, C = 0.7$

Table 4.4.: Overview of the different communication protocol configurations for the comparison with FedAvg. Except for protocol parameters Δ and C, all other parameters are kept constant between configurations.

Figure 4.7 shows the evolution of cumulative communication during model training comparing dynamic averaging to the optimal configuration of FedAvg with $b = 5$ and $C = 0.3$ for MNIST (see Section 3 in McMahan et al. (2017)) and variants of this configuration. Between the communication curves a noteworthy spread can be found, while all approaches have comparable losses. The communication amounts of all FedAvg variants increase linearly during training. The smaller the fraction of learners, $C \in (0, 1]$, involved in synchronization, the smaller the amount of communication. In contrast, the curves for all dynamic averaging protocols increase step-wise which reflect their inherent irregularity of communication. Dynamic averaging with $\Delta = 0.6$ and $\Delta = 0.8$ beat the strongest FedAvg configuration in terms of cumulative communication, the one with $\Delta = 0.8$ even with a remarkable margin. These improvements in communication efficiency come at virtually no cost: Figure 4.8 compares the three strongest configurations of dynamic averaging to the best performing FedAvg ones, showing a reduction of over 50% in communication with an increase in cumulative loss by

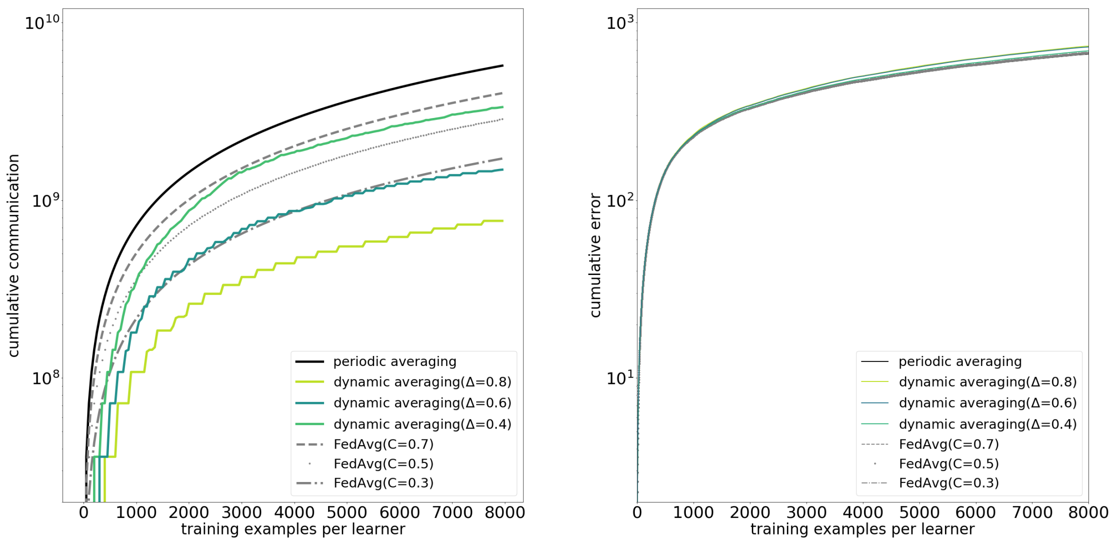
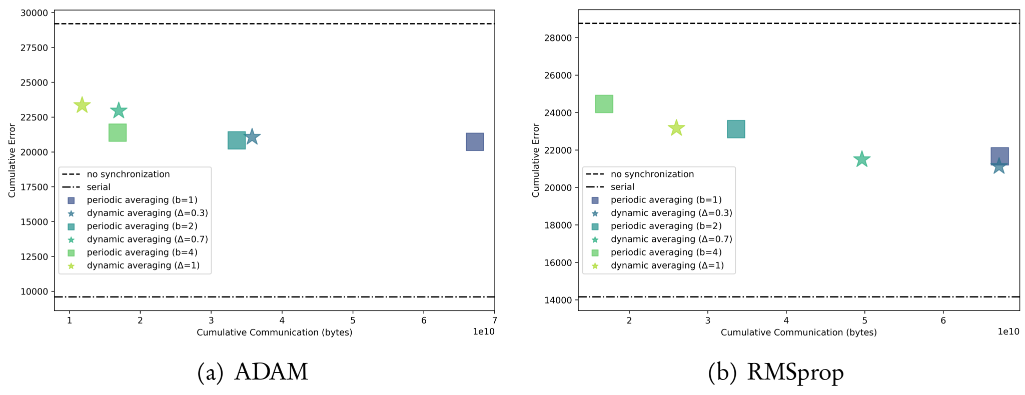


Figure 4.9.: Evolution of (a) total communication and (b) cumulative error during training for different dynamic averaging and FedSGD protocols.

only 8.3%. The difference in terms of classification accuracy is even smaller, dynamic averaging is only worse by 1.9%. Allowing for more communication improves the loss of dynamic averaging to the point where dynamic averaging has virtually the same accuracy as FedAvg with 16.9% less communication.

Black-Box Property



(a) ADAM (b) RMSprop

Figure 4.10.: The averaged loss and the cumulative communication of the synchronization protocols for $m = 10$ learners. Training is performed on MNIST for 2 epochs.

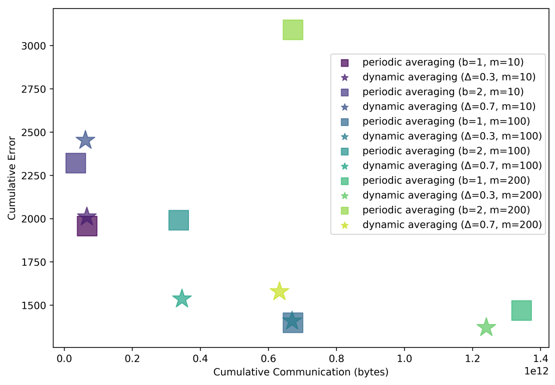


Figure 4.11.: The cumulative loss and the cumulative communication of the same synchronization protocols for a different amount of learners. Training is performed on MNIST for 2, 20 and 40 epochs for $m = 10$, $m = 100$, $m = 200$ setups correspondingly.

In comparison with distributed mini-batch SGD (Chen et al., 2016; Dekel et al., 2012), dynamic averaging allows to treat the optimization algorithm as a black-box. To show this, the MNIST experiments are repeated with the ADAM optimizer (Kingma and Ba, 2014) and RMSprop (Tieleman and Hinton, 2012). The experiments show similar behavior for the same synchronization operators, i.e., $\sigma_{b=1}$, $\sigma_{b=2}$, $\sigma_{b=4}$, $\sigma_{\Delta=0.3}$, $\sigma_{\Delta=0.7}$, $\sigma_{\Delta=1}$ (see Figure 4.10).

Scale-out Experiments

Communication size grows when the amount of the local learners is becoming larger, while it is still bounded for periodic and dynamic synchronization protocols (Section 4.1). In order to see the behavior of periodic and dynamic synchronization while changing the number of learners an experiment that included setup with $m = 10$, $m = 100$ and $m = 200$ was executed. The same synchronization operators were used in all three setups and the same number of examples was presented to each of the learners.

When the amount of examples per learner is fixed the larger number of synchronizing learners leads to a larger training dataset and, as a consequence, to a better performance of all the models. When the models are saturated due to the long training and do not differ significantly enough to trigger the local conditions (see Section 3.2), the advantage of the dynamic protocols over the periodic ones becomes more pronounced. The plot depicted in Figure 4.11 shows the performance of two dynamic and two periodic protocols for the three scaling setups. In order to make the cumulative loss comparable it was divided by the number of learners, i.e., the cumulative loss of $m = 100$ learners is the sum of cumulative losses of all of them that is 10 times more than the sum for $m = 10$ learners—thus the first sum divided by 100 is comparable to the second one divided by 10. The plot shows that in the setup with $m = 10$ learners $\sigma_{\Delta=0.7}$ shows a comparable result to $\sigma_{b=2}$ by the price of the same communication. At the same time with $m = 100$ learners it already reaches much smaller cumulative loss. With $m = 200$ learners $\sigma_{\Delta=0.3}$ requires less communication than $\sigma_{b=1}$ that was not the case for the previous setups.

Case Study on Deep Driving

After having studied dynamic averaging in contrast to periodic approaches and FedAvg on MNIST, it is analyzed how it performs in the realistic application scenario of in-fleet training for autonomous driving. That is, every vehicle continuously trains a local model based on its observations in shadow mode. This requires to infer the correct output locally for training. For that, the label is inferred from a human driver by mimicking her driving behavior using a frontal view camera as input (Bojarski et al., 2016; Fernando et al., 2017; Pomerleau, 1989).

Using only vehicles in a specific region, the data seen by an individual local learner has in good approximation a low variability and heterogeneity, because people driving cars tend to stay close to their base. Thus, data from cars from a similar region can be assumed to be fairly homogeneously distributed.

In order to mimic this scenario for the experiments, human driving behavior is recorded in a simulation for multiple drivers on a single track. However, data of cars from different base locations, collected at different times or in different cars will underlie different (local) approximations of the actual distribution. This actual distribution has in contrast to its local approximations a large variability and heterogeneity, even if one considers only the minor set of tasks solved by machine learning. Thus, for in-fleet training over various regions the data cannot be assumed iid anymore. However, empirical results (McMahan et al., 2017) and recent extensions to the federated learning (Smith et al., 2017) suggest that the approach is capable of handling non-iid data. One of the approaches in autonomous driving is direct steering control of a constantly moving car via a neural network that predicts a steering angle given an input from the front view camera. Since one network fully controls the car this approach is termed **deep driving**. Deep driving neural networks can be trained on a dataset generated by recording human driver control and corresponding frontal view (Bojarski et al., 2016; Fernando et al., 2017; Pomerleau, 1989).

For the experiments, a neural network architecture is used that is suggested for deep driving by Bojarski et al. (2016). The architecture of its layers are given in Table 4.5. The employed loss function is squared error as it is implemented in Keras "mean_squared_error". The network is optimized using mini-batch SGD. The experiments were run with training batch size of $B = 10$ and learning rate $\eta = 0.1$ both for the baselines and local learners. The input to the network is the front camera view from a car driven in a simulator[2]. The output of the network is a steering angle that allows to control the car in the autonomous mode in the simulator when the speed is kept on a constant level.

The learners are evaluated by their driving ability following the qualitative evaluation made by Bojarski et al. (2016) or Pomerleau (1989) as well as techniques used in the automotive industry. For that, a custom loss was developed together with experts for autonomous driving: During the evaluation a trained model has been loaded to drive the car in the autonomous regime in the simulator. The time that the model is able to control the car without going off the road or crashing is measured. The longest time t_{max} is the time for the model that is able to keep going for 2 laps on the track or the maximum time of all the models in one experiment. Also the amount of times the car has touched the sideline of the road is counted together with the time duration while car is still on the sideline t_{line}. Then a frequency c of

[2] `https://github.com/udacity/self-driving-car-sim`

Layer Type	Output Shape	#Weights
Conv2D	(32, 158, 24)	1 824
Conv2D	(14, 77, 36)	21 636
Conv2D	(5, 37, 48)	43 248
Conv2D	(3, 35, 64)	27 712
Conv2D	(1, 33, 64)	36 928
Flatten	(2 112)	0
Dense	(100)	211 300
Dense	(50)	5 050
Dense	(10)	510
Dense	(1)	11
Total		348 219

Table 4.5.: The architecture of the Convolutional Neural Network used for deep driving. Printout of the parameters made via Keras library.

sideline crossings is calculated in a form $\frac{\#crossings}{t}$, where t is the time before going off road or crash, and the maximal frequency among all the models is assigned to c_{max}. The overall formula for the custom loss is:

$$L_{dd} = \lambda\frac{t_{max} - t}{t_{max}} + \mu\frac{c}{c_{max}} + (1 - \mu - \lambda)\frac{t_{line}}{t}$$

where t is the time that the model is able to drive on the road, $\lambda, \mu \in [0; 1]$ are weighting coefficients. For the experiment $\lambda = 0.8$ and $\mu = 0.15$ were used.

The amount of examples shown to each of the $m = 10$ learners is 25000, i.e., the dataset was shown to the serial learner approximately 5 times (the overall size of the dataset is ≈ 48000). The list of the considered communication protocols can be seen from Table 4.6.

Figure 4.12(a) shows the measurements of the custom loss against the cumulative communication. The principal difference from the previous experiments is the evaluation of the resulting models without taking into account cumulative training loss. All the resulting models as well as baseline models were loaded to the simulator and driven with a constant speed. The plot shows that each periodic communication protocol can be outperformed by a dynamic protocol. Examining the evolution of cumulative communication (see Figure 4.12(b)) shows that—similar to the results on MNIST—dynamic averaging invests a large amount of communication in the beginning and then considerably reduces communication. This pattern is most apparent for dynamic averaging with $\Delta = 0.1$.

Similar to our previous experiments, too little communication leads to bad performance, but for deep driving, very high communication ($\sigma_{b=1}$ and $\sigma_{\Delta=0.01}$) results in a bad performance as well. On the other hand, proper setups achieve performance similar to the performance of the serial model (e.g. dynamic averaging with $\Delta = 0.1$ or $\Delta = 0.3$). This raises the question, how much diversity is beneficial in-between averaging steps and how diverse models should be initialized. We discuss this question and other properties of dynamic averaging in the following section.

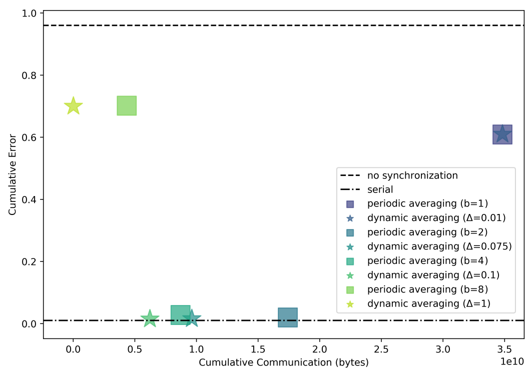
protocol configurations	
type	**parameters**
periodic protocol (σ_b)	$b = 1$
	$b = 2$
	$b = 4$
	$b = 8$
dynamic protocol ($\sigma_{\Delta,b}$)	$b = 1, \Delta = 0.01$
	$b = 1, \Delta = 0.05$
	$b = 1, \Delta = 0.1$
	$b = 1, \Delta = 0.3$

Table 4.6.: Overview of the different communication protocol configurations used for deep driving experiment. Except for protocol parameters Δ and b, all other parameters are kept constant between configurations.

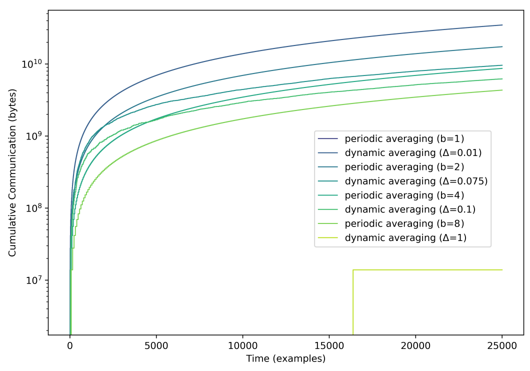
Impact of Model Initialization for Distributed Deep Learning

A general question when using averaging is how local models should be initialized. McMahan et al. (2017) suggest using the same initialization for all local models and report that different initializations deteriorate the learning process when models are averaged only once at the end. Studying the transition from homogeneously initialized and converging model configurations to heterogeneously initialized and failing ones reveals that, surprisingly, for multiple rounds of averaging different initializations can indeed be beneficial. Figure 4.13 shows the performances of dynamic and periodic averaging for different numbers of rounds of averaging and different levels of inhomogeneity in the initializations. The results confirm that for one round of averaging, strongly inhomogeneous initializations deteriorate the learning process, but for more frequent rounds of averaging mild inhomogeneity actually improves training. For large heterogeneities, however, model averaging fails as expected. This raises an interesting question about the regularizing effects of averaging and its potential advantages over serial learning in case of non-convex objectives.

To parameterize the transition from homogeneous to heterogeneous initializations, the setup starts with a homogeneous initialization according to Xavier Glorot (2010) and imposes noise at different scales ϵ on the homogeneous initialization. The noise scale ϵ is measured relative to the scale of the homogeneous initialization. Experiments are conducted with $m = 10$ learners, $B = 10$ and 500 training examples per learner for a grid of ϵ and b combinations. In Fig. 4.13, each point corresponds to the average model performance after running one such experiment. The b dependency of the averaged model performance is shown on the abscissa, noise scale ϵ serves as curve parameter. Fig. 4.13(a) shows the results for periodic averaging, Fig. 4.13 (a) for dynamic model averaging. The averaged model accuracies are not shown as absolute values but relative to the configuration with $\epsilon = 0$ and $b = 1$ which corresponds to homogeneously initialized models which communicate after processing one mini-batch.



(a)



(b)

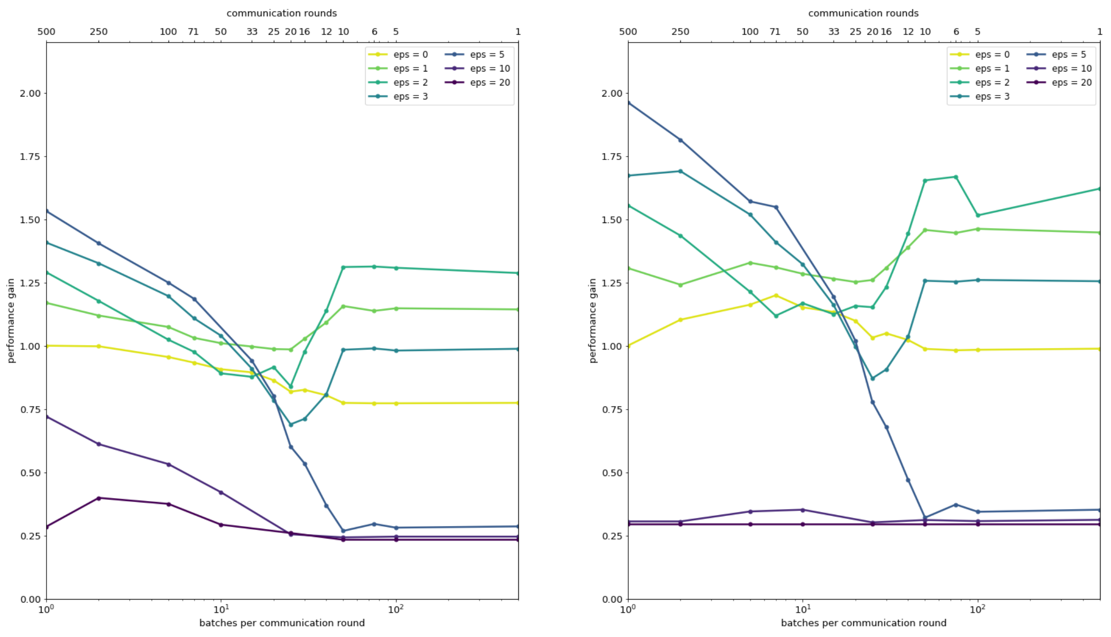
Figure 4.12.: Performance in the terms of the custom loss for the models trained according to a set of communication protocols and baseline models, showing the (a) trade-off between loss and communication and the (b) development of communication.

For homogeneously initialized models, i.e., $\epsilon = 0$, a weak dependence can be found of resulting model performance on the number of local mini-batches. Even configurations with very large numbers of local batches between two subsequent model averagings lead to convergence (see McMahan et al. (2017)). However, this finding can be extended to the heterogeneous case, if the scales of these heterogeneities are at the scale of the underlying homogeneous initialization, e.g., $\epsilon \in \{1, 2, 3\}$. For large heterogeneities, however, model averaging fails, e.g. for $\epsilon = 20$. The transition between these two regimes occurs between $\epsilon = 5$ and $\epsilon = 10$, which show a strong dependency of model convergence on the number of local mini-batches. This critical scale of heterogeneity imposes a constrain on the choice of the dynamic protocol parameter Δ which indirectly determines the average distances between the different weight vectors before model averaging.

4.5. Discussion

In this chapter, dynamic averaging was applied to incremental learning algorithms for batch learning. For learning algorithms that perform regret-proportional convex updates it was shown that dynamic averaging retains the convergence rate of periodic averaging. This includes SGD, mini-batch SGD, GD, and passive aggressive updates for linear and kernel models. Since periodic averaging for SGD and mini-batch SGD has an optimal convergence rate it follows that dynamic averaging for those base learning algorithms has an optimal convergence rate as well. Moreover, the communication bounds from Chapter 3 apply here as well.

A shortcoming of this analysis is that these algorithms do not perform regret-proportional convex updates when applied to neural networks. However, it can be shown that SGD, mini-batch SGD, and GD with neural networks perform updates that are contractions in case the models are in a locally convex environment. For this case, the convergence result has been extended to learning algorithms that perform updates which are contractions. Thus, under



(a) static averaging protocols (b) dynamic averaging protocols

Figure 4.13.: Relative performances of averaged models on MNIST obtained from various heterogeneous model initializations parameterized by ϵ and various $b \in \mathbb{N}$. All averaged model performances are compared to an experiment with homogeneous model initializations ($\epsilon = 0$) and $b = 1$.

the assumption that the region of the loss surface around the neural network models is locally convex, it can be shown that dynamic averaging has optimal convergence for SGD, mini-batch SGD, and GD.

Using this result on the convergence rate, the generalization error of using dynamic averaging for batch learning can be bounded using the ERM model. Assume that each of the $m \in \mathbb{N}$ local learners $i \in [m]$ has a local dataset E^i. Since the model obtained from dynamic averaging converges to the minimizer over all local datasets, the number of examples observed is given by

$$N = \sum_{i=1}^{m} |E^i| \ .$$

If the learning algorithm has a sample complexity of $N_{\mathcal{F}}(\epsilon, \delta)$, then given $\delta \in (0, 1]$ and setting $N = N_{\mathcal{F}}(\epsilon, \delta)$ allows to solve for ϵ. Thus, the risk can be bounded by

$$\mathcal{L}_{\mathcal{D}}(f_T) \leq \min_{f' \in \mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') + \epsilon + \widehat{\epsilon} \ ,$$

where $\widehat{\epsilon} \in \mathbb{R}_+$ denotes the optimization error after $T \in \mathbb{N}$ rounds (see Section 2.1.4 in Chapter 2). Since dynamic averaging retains the convergence rate of several base learning algorithms, the optimization error can be quantified as well: If such a base learning algorithm has a convergence rate in $\mathcal{O}(g(T))$ (e.g., recall that for SGD the convergence rate is in $\mathcal{O}(1/\sqrt{T})$), then the convergence rate of dynamic averaging is in $\mathcal{O}(g(mT))$. For example, for dynamic averaging with SGD it follows that $\widehat{\epsilon} \in \mathcal{O}\left(1/mT\right)$.

When using dynamic averaging with neural networks, a major problem both in theory and in practice is that models are assumed to be in a locally convex environment. As mentioned already in Section 3.2.3 in Chapter 3, averaging models for non-convex objectives can deteriorate the learning process. For example, all local models can be converged to different local minima so that their average performs worse than any local model (see Figure 4.14 for an illustration). McMahan et al. (2017) argue that, in practice, by initializing all local models

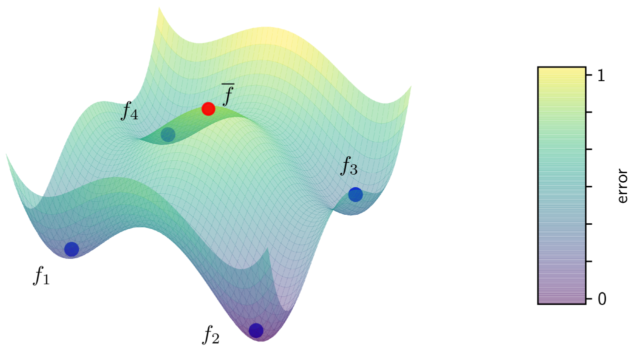


Figure 4.14.: Illustration of the problem of averaging models in non-convex problems: each of the models $f_1, \ldots, f_4$ has reached a local minimum, but their average $\overline{\mathbf{f}}$ has a higher error than each of them.

to the same initial model averaging works for (deep) neural networks. The experiments in Section 4.4.2 confirm this. In addition, they show that mildly inhomogeneous initialization even improves the training. Of course, strongly heterogeneous initialization leads the training to fail. This supports the assumption that models initialized in the same neighborhood with high probability remain in the same locally convex environment. This indicates that the assumption in Theorem 4.4, i.e., in each round local models are in a locally convex environment, is not entirely unrealistic.

This chapter has shown that dynamic averaging is indeed suitable to batch learning with incremental learning algorithms. It retains the convergence rate of several popular learning algorithms, scaling well with the number of learners, and reducing communication substantially compared to periodic scheme, such as periodic averaging and FedAvg. However, it does not achieve polylogarithmic runtime on quasi-polynomially many processing units. Moreover, it is not applicable to non-incremental algorithms.

The next chapter presents a novel aggregation operator based on the Radon point that can be used in an aggregation-at-the-end parallelization. The resulting distributed learning protocol allows to parallelize a broad class of machine learning algorithms—including non-incremental ones–with minimal communication and substantial speedup. At the same time it achieves the same model quality as the serial application of the base learning algorithm.

5. Effective Parallelization Using Radon Points

In the Chapters 3 and 4 I have presented an efficient distributed learning protocol for incremental learning algorithms. Since this protocol requires multiple aggregation steps during the training process, it is not suitable for non-incremental algorithms. For them algorithms, the only viable strategy is to aggregate models at the end of the training process. Such an aggregation-at-the-end protocol runs the non-incremental learning algorithm $\mathcal{A}$ in parallel on m learners with m local datasets $E^1, \dots, E^m$ to obtain m weak models $f^1, \dots f^m$. Here, weak means that the quality of these models is substantially lower than that of a model trained on the union of all local datasets. The goal is to aggregate them into a single model $f = \mathfrak{a}(f^1, \dots f^m)$ with a quality similar to one trained on the union of local datasets. A natural choice for the aggregation operator would be the average. However, no strong guarantees on the model quality can be given for averaging-at-the-end: Shamir and Srebro (2014) have shown that averaging-at-the-end can be arbitrarily bad.

The goal of this chapter is to derive a novel and provably effective parallelization scheme for a broad class of learning algorithms, including non-incremental ones. The significance of this result is to allow the confident application of machine learning algorithms with growing amounts of data. In critical application scenarios, i.e., when errors have almost prohibitively high cost, this confidence is essential (Nouretdinov et al., 2011; Sommer and Paxson, 2010). To this end, the parallelization of an algorithm is considered to be effective if it achieves the same confidence and error bounds as the sequential execution of that algorithm in much shorter time. Indeed, the proposed parallelization scheme can reduce the runtime of learning algorithms from polynomial to polylogarithmic. For that, it consumes more data and is executed on a quasi-polynomial number of processing units. The amount of communication required only depends on the amount of processing units.

Section 5.1 recapitulates the learning setting. Section 5.2 introduces the aggregation operator and the distributed learning protocol, denoted Radon machine, and provides probabilistic error guarantees. Section 5.3 then analyzes the sample complexity and runtime of Radon machine to achieve a given error guarantee, followed by an analysis of its speedup. The approach is empirically evaluated in Section 5.4. The chapter concludes with a discussion of the advantages and limitations of the approach in Section 5.5.

5.1. Non-Incremental Batch Learning Algorithms

In order to analyze the parallelization of a batch learning algorithm, this chapter again uses the empirical risk minimization model (see Section 2.1.3). That is, for a fixed but unknown joint probability distribution $\mathcal{D}$ over the input space $\mathcal{X}$ and the output space $\mathcal{Y}$, a dataset $E \subseteq \mathcal{X} \times \mathcal{Y}$ of size $N \in \mathbb{N}$ drawn iid from $\mathcal{D}$, a convex model space $\mathcal{F}$ of functions $f : \mathcal{X} \to \mathcal{Y}$, a loss function $\ell \colon \mathcal{F} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ that is convex in $\mathcal{F}$, and a convex regularization term $\Omega \colon \mathcal{F} \to \mathbb{R}$, (regularised) empirical risk minimization algorithms solve

$$\mathcal{A}(E) = \operatorname*{argmin}_{f \in \mathcal{F}} \sum_{(x,y) \in E} \ell(f, x, y) + \Omega(f) \ .$$

Recall that the aim is to obtain a model $f \in \mathcal{F}$ with small regret

$$\mathcal{R}(f) = \mathcal{L}_{\mathcal{D}}(f) - \min_{f' \in \mathcal{F}} \mathcal{L}_{\mathcal{D}}(f') \ .$$

Empirical risk minimization algorithms are typically designed to be consistent[1], i.e., there is a function $N_{\mathcal{F}} \colon \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$ such that for all $\epsilon > 0$, $\Delta \in (0,1]$, $N \in \mathbb{N}$ with $N \geq N_{\mathcal{F}}(\epsilon, \Delta)$, and training data $E \sim \mathcal{D}^N$, the probability of generating an ϵ-bad model is smaller than Δ, i.e.,

$$P(\mathcal{R}(\mathcal{A}(E)) > \epsilon) \leq \Delta \ .$$

Such algorithms are **efficient**[2] if the sample complexity $N_{\mathcal{F}}(\epsilon, \Delta)$ is polynomial in $1/\epsilon, \log 1/\Delta$ and the runtime complexity $\mathcal{T}_{\mathcal{A}}$ is polynomial in the sample complexity.

The main theoretical contribution of this chapter is to show that algorithms satisfying the above conditions, e.g., support vector machines, regularized least squares regression, and logistic regression, can be parallelized effectively. We consider a parallelization to be **effective** if the (ϵ, Δ)-guarantees (Equation 2.6) are achieved in time polylogarithmic in $N_{\mathcal{F}}(\epsilon, \Delta)$. To achieve that it is furthermore assumed that data is abundant and that $\mathcal{F}$ can be parameterized in a fixed, finite dimensional Euclidean space $\mathbb{R}^d$ such that the convexity of the empirical risk minimization problem is preserved. The cost for achieving this reduction in runtime comes in the form of an increased data size and through the number of processing units used. For the distributed learning protocol presented in this chapter, this cost can be bound by a quasi-polynomial in $1/\epsilon$ and $\log 1/\Delta$.

[1] The notion of consistency differs from the consistency of a distributed learning protocol: the consistency of an empirical risk minimization algorithm denotes a particular probabilistic error guarantee, whereas a consistent distributed learning protocol retains a particular error guarantee of the base learning algorithm.

[2] Efficiency here means that an empirical risk minimization algorithm achieves a probabilistic error guarantee in polynomial time, whereas a distributed learning protocol is efficient if it retains error guarantees of its base learning algorithm with bounded communication.

The synchronization operator used in this chapter is aggregation at the end

$$\sigma_{\text{end}}(f^1, \ldots, f^m) = \mathfrak{a}(f^1, \ldots, f^m) \ .$$

As mentioned above, averaging is not a suitable aggregation operator, since averaging-at-the-end can be arbitrarily bad (Shamir and Srebro, 2014). Therefore, in this Chapter I propose to use an aggregation operator based on the Radon point (Radon, 1921). Similar to averaging-at-the-end-based parallelizations (Rosenblatt and Nadler, 2016; Zhang et al., 2013; Zinkevich et al., 2010), the resulting distributed learning protocol applies the underlying learning algorithm in parallel to random subsets of the data. Each resulting model is assigned to a leaf of an aggregation tree which is then traversed bottom-up. Each inner node computes a new model that is a **Radon point** (Radon, 1921) of its children's hypotheses. In contrast to aggregation by averaging, the Radon point increases the confidence in the aggregate doubly-exponentially with the height of the aggregation tree.

5.2. From Radon Points to Radon Machines

This section introduces a distributed learning protocol $\mathcal{R} = (\sigma_{\text{end}}, \mathfrak{a}_{\mathfrak{r}})$, denoted Radon machine, that aggregates models at the end using an aggregation operator $\mathfrak{a}_{\mathfrak{r}}$ based on the Radon point (Radon, 1921). The Radon point of a set of points is defined as follows.

Definition 5.1. *A **Radon partition** of a set $S \subset \mathcal{F}$ is a pair $A, B \subset S$ such that $A \cap B = \emptyset$ but $\langle A \rangle \cap \langle B \rangle \neq \emptyset$, where $\langle \cdot \rangle$ denotes the convex hull. The **Radon number** of a space $\mathcal{F}$ is the smallest $r \in \mathbb{N}$ such that for all $S \subset \mathcal{F}$ with $|S| \geq r$ there is a Radon partition; or ∞ if no Radon partition exists. A **Radon point** of a set S with Radon partition A, B is any $\mathfrak{r} \in \langle A \rangle \cap \langle B \rangle$.*

See Figure 5.1 for an illustration of the Radon point in $\mathbb{R}^2$. For the Euclidean space $\mathbb{R}^d$, a simple construction of a system of linear equations can be given with which a Radon point of a set can be determined. In his main theorem, Radon (Radon, 1921) gives the following construction of a Radon point for a set $S = \{s_1, ..., s_r\} \subseteq \mathbb{R}^d$ with $r > d + 1$. Find a non-zero solution $\lambda \in \mathbb{R}^{|S|}$ for the following linear equations

$$\sum_{i=1}^{r} \lambda_i s_i = (0, \ldots, 0) \ , \ \sum_{i=1}^{r} \lambda_i = 0 \ .$$

Such a solution exists, since $|S| > d + 1$ implies that S is linearly dependent. Then, let I, J be index sets such that for all $i \in I : \lambda_i \geq 0$ and for all $j \in J : \lambda_j < 0$. Then a Radon point is defined by

$$\mathfrak{r}(\lambda) = \sum_{i \in I} \frac{\lambda_i}{\Lambda} s_i = \sum_{j \in J} \frac{\lambda_j}{\Lambda} s_j \ , \tag{5.2}$$

where $\Lambda = \sum_{i \in I} \lambda_i = -\sum_{j \in J} \lambda_j$. Any solution to this linear system of equations is a Radon point. The equation system can be solved in time r^3. By setting the first element of λ to one, we obtain a unique solution of the system of linear equations. Using this solution λ, we define the Radon point of a set S as $\mathfrak{r}(S) = \mathfrak{r}(\lambda)$ in order to resolve ambiguity. With this, the distributed learning protocol $\mathcal{R}$, denoted Radon machine, can be defined.

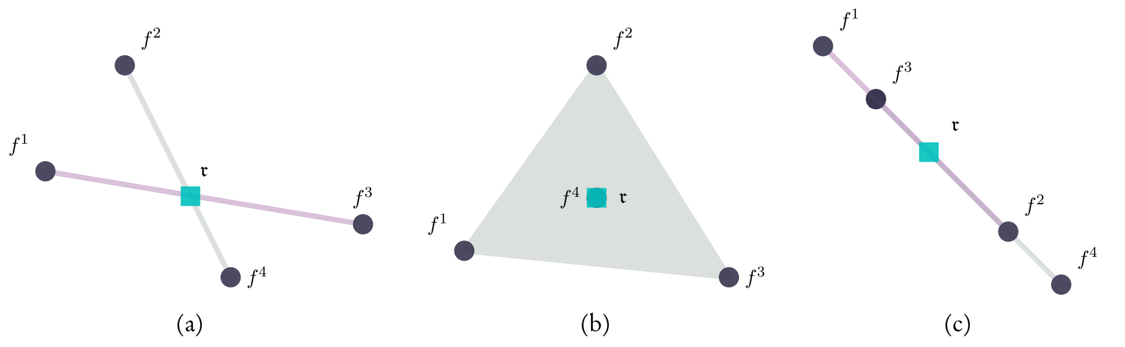


Figure 5.1.: Illustration of a Radon point in $\mathbb{R}^2$ for different locations of points $f^1, \dots, f^4$: points in general position (a) with Radon partitioning $A = \{f^1, f^3\}, B = \{f^2, f^4\}$, one point in the convex hull of the other three (b) with Radon partitioning $A = \{f^1, f^2, f^3\}, B = \{f^4\}$, and collinear points (c) with Radon partition $A = \{f^1, f^2\}, B = \{f^3, f^4\}$. A Radon point is any point in the intersection of the Radon partition A, B. In (a) and (b) the Radon point is unique, in (c) infinitely many points on the line segment from f^3 to f^2 are Radon points, with one arbitrary Radon point depicted in the figure.

The Radon machine, as described in Algorithm 7, first executes the base learning algorithm on random subsets of the data to quickly achieve weak models and then iteratively aggregates them to stronger ones. Both the generation of weak models and the aggregation can be executed in parallel. To aggregate models, we follow along the lines of the iterated Radon point algorithm which was originally devised to approximate the center point of a finite set of points (Clarkson et al., 1996).

Input to the Radon machine is a learning algorithm $\mathcal{A}$ on a model space $\mathcal{F}$, a dataset $E \subseteq \mathcal{X} \times \mathcal{Y}$, the Radon number $r \in \mathbb{N}$ of the model space $\mathcal{F}$, and a parameter $h \in \mathbb{N}$. It divides the dataset into r^h subsets $E_1, \dots, E_{r^h}$ (line 1) and runs the algorithm $\mathcal{A}$ on each subset in parallel (line 2). Then, the set of models (line 3) is iteratively aggregated to form better sets of models (line 4-8). For that the set is partitioned into subsets of size r (line 5) and the Radon point of each subset is calculated in parallel (line 6). The final step of each iteration is to replace the set of models by the set of Radon points (line 7).

The Radon machine requires a model space with a valid notion of convexity and finite Radon number. While other notions of convexity are possible (Kay and Womble, 1971; Rubinov, 2013), this chapter restricts its consideration to Euclidean spaces with the usual notion of convexity. Radon's theorem (Radon, 1921) states that the Euclidean space $\mathbb{R}^d$ has Radon number $r = d + 2$. Radon points can then be obtained by solving a system of linear equations of size $r \times r$ (see Equation 5.2). The next proposition gives a guarantee on the quality of Radon points of models.

Algorithm 7 Radon Machine

```
Require: learning algorithm $\mathcal{A}$, dataset $E \subseteq \mathcal{X} \times \mathcal{Y}$, Radon number $r \in \mathbb{N}$, and parameter $h \in \mathbb{N}$
Ensure: model $f \in \mathcal{F}$
1. divide $E$ into $r^h$ iid subsets $E_i$ of roughly equal size
2. run $\mathcal{A}$ in parallel to obtain $f_i = \mathcal{A}(E_i)$
3. $S \leftarrow \{f_1, \dots, f_{r^h}\}$
4. for $i = h - 1, \dots, 1$ do
5.     partition $S$ into iid subsets $S_1, \dots, S_{r^i}$ of size $r$ each
6.     calculate Radon points $\mathfrak{r}(S_1), \dots, \mathfrak{r}(S_{r^i})$ in parallel        # see Definition 5.1
7.     $S \leftarrow \{\mathfrak{r}(S_1), \dots, \mathfrak{r}(S_{r^i})\}$
8. end for
9. return $\mathfrak{r}(S)$
```

Proposition 5.3. *Given a probability measure P over a model space $\mathcal{F}$ with finite Radon number r, let F denote a random variable with distribution P. Furthermore, let $\mathfrak{r}$ be the random variable obtained by computing the Radon point of r random points drawn according to P^r. Then it holds for the expected regret $\mathcal{R}$ and all $\epsilon \in \mathbb{R}$ that*

$$P\left(\mathcal{R}\left(\mathfrak{r}\right) > \epsilon\right) \leq \left(rP\left(\mathcal{R}\left(F\right) > \epsilon\right)\right)^2 \quad .$$

Proposition 5.3 is proven together with Theorem 5.4. A direct consequence of Proposition 5.3 is a bound on the probability that the output of the Radon machine with parameter h is bad:

Theorem 5.4. *Given a probability measure P over a model space $\mathcal{F}$ with finite Radon number r, let F denote a random variable with distribution P. Denote by $\mathfrak{r}_1$ the random variable obtained by computing the Radon point of r random points drawn iid according to P and by P_1 its distribution. Denote by $\mathfrak{r}_h$ the Radon point of r random points drawn iid from P_{h-1} and by P_h its distribution. Then for any convex function $\mathcal{R} : \mathcal{F} \to \mathbb{R}$ and all $\epsilon \in \mathbb{R}$ it holds that*

$$P\left(\mathcal{R}(\mathfrak{r}_h) > \epsilon\right) \leq \left(rP\left(\mathcal{R}(F) > \epsilon\right)\right)^{2^h} \quad .$$

In order to prove Proposition 5.3 and consecutively Theorem 5.4, the following properties of Radon points and convex functions are required. These properties are proven for the more general case of quasi-convex functions. Since every convex function is also quasi-convex, the results hold for convex functions as well. A quasi-convex function is defined as follows.

Definition 5.5. *A function $\mathcal{R} : \mathcal{F} \to \mathbb{R}$ is called **quasi-convex** if all its sublevel sets are convex, i.e.,*

$$\forall \theta \in \mathbb{R} : \{f \in \mathcal{F} \mid \mathcal{R}\left(f\right) < \theta\} \text{ is convex.}$$

First, a different characterization of quasi-convex functions is given.

Proposition 5.6. *A function $\mathcal{R} : \mathcal{F} \to \mathbb{R}$ is quasi-convex if and only if for all $S \subseteq \mathcal{F}$ and all $s' \in \langle S \rangle$ there exists an $s \in S$ with $\mathcal{R}(s) \geq \mathcal{R}(s')$.*

Proof.

($\Rightarrow$) Suppose this direction does not hold. Then there is a quasi-convex function $\mathcal{R}$, a set $S \subseteq \mathcal{F}$, and an $s' \in \langle S \rangle$ such that for all $s \in S$ it holds that $\mathcal{R}(s) < \mathcal{R}(s')$ (therefore $s' \notin S$). Let $C = \{c \in \mathcal{F} \mid \mathcal{R}(c) < \mathcal{R}(s')\}$. As $S \subseteq C = \langle C \rangle$ we also have that $\langle S \rangle \subseteq \langle C \rangle$ which contradicts $\langle S \rangle \ni s' \notin C$.

($\Leftarrow$) Suppose this direction does not hold. Then there exists an ϵ such that $S = \{s \in \mathcal{F} \mid \mathcal{R}(s) < \epsilon\}$ is not convex and therefore there is an $s' \in \langle S \rangle \setminus S$. By assumption $\exists s \in S : \mathcal{R}(s) \geq \mathcal{R}(s')$. Hence $\mathcal{R}(s') < \epsilon$ and we have a contradiction since this would imply $s' \in S$.

□

The next proposition concerns the value of any convex function at a Radon point.

Proposition 5.7. *For every set S with Radon point $\mathfrak{r}$ and every quasi-convex function $\mathcal{R}$ it holds that $|\{s \in S \mid \mathcal{R}(s) \geq \mathcal{R}(\mathfrak{r})\}| \geq 2$.*

Proof. We show a slightly stronger result: Take any family of pairwise disjoint sets A_i with $\bigcap_i \langle A_i \rangle \neq \emptyset$ and $\mathfrak{r} \in \bigcap_i \langle A_i \rangle$. From proposition 5.6 follows directly the existence of an $a_i \in A_i$ such that $\mathcal{R}(a_i) \geq \mathcal{R}(\mathfrak{r})$. The desired result follows then from $a_i \neq a_j \Leftarrow i \neq j$. □

Using this property, Proposition 5.3 and Theorem 5.4 can be proven.

Proof of Proposition 5.3 and Theorem 5.4. By proposition 5.7, for any Radon point $\mathfrak{r}$ of a set S there must be two points $a, b \in S$ with $\mathcal{R}(a), \mathcal{R}(b) \geq \mathcal{R}(\mathfrak{r})$. Henceforth, the probability of $\mathcal{R}(\mathfrak{r}) > \epsilon$ is smaller or equal than the probability of the pair a, b having $\mathcal{R}(a), \mathcal{R}(b) > \epsilon$. Proposition 5.3 follows by an application of the union bound on all pairs from S. Repeated application of the proposition proves Theorem 5.4. □

Note that this proof also shows the robustness of the Radon point compared to the average: if only one of r points is ϵ-bad, the Radon point is still ϵ-good, while the average may or may not be; indeed, in a linear space with any set of ϵ-good models and any $\epsilon' \geq \epsilon$, one can always find a single ϵ'-bad model such that the average of all these models is ϵ'-bad.

For the Radon machine with parameter h, Theorem 5.4 shows that the probability of obtaining an ϵ-bad model is doubly exponentially reduced: with a bound δ on this probability for the base learning algorithm, the bound Δ on this probability for the Radon machine is

$$\Delta = (r\delta)^{2^h} \quad . \tag{5.8}$$

The next section will use this relation between Δ and δ to compare the Radon machine to a sequential application of the base learning algorithm for the same ϵ and Δ.

5.3. Sample and Runtime Complexity

This section first derives the sample and runtime complexity of the Radon machine $\mathcal{R}$ from the sample and runtime complexity of the base learning algorithm $\mathcal{A}$. It then relates the runtime complexity of the Radon machine to a sequential application of the base learning algorithm when both achieve the same (ϵ, Δ)-guarantee. For that, consistent and efficient base learning algorithms are considered with a sample complexity of the form $N_0^{\mathcal{A}}(\epsilon, \delta) = (\alpha_\epsilon + \beta_\epsilon \log_2 {}^1\!/\!_\delta)^k$, for some[3] $\alpha_\epsilon, \beta_\epsilon \in \mathbb{R}$, and $k \in \mathbb{N}$. From now on, it is also assumed that $\delta \leq {}^1\!/\!_{2r}$ for the base learning algorithm.

The Radon machine creates r^h base models and, with Δ as in Equation 5.8, has sample complexity

$$N_0^{\mathcal{R}}(\epsilon, \Delta) \;=\; r^h N_0^{\mathcal{A}}(\epsilon, \delta) \;=\; r^h \cdot \left(\alpha_\epsilon \;+\; \beta_\epsilon \log_2 \frac{1}{\delta}\right)^k \quad . \tag{5.9}$$

Theorem 5.4 then implies that the Radon machine with base learning algorithm $\mathcal{A}$ is consistent: with $N \geq N_0^{\mathcal{R}}(\epsilon, \Delta)$ samples it achieves an (ϵ, Δ)-guarantee.

To achieve the same guarantee as the Radon machine, the application of the base learning algorithm $\mathcal{A}$ itself (sequentially) would require $M \geq N_0^{\mathcal{A}}(\epsilon, \Delta)$ samples, where

$$N_0^{\mathcal{A}}(\epsilon, \Delta) \;=\; N_0^{\mathcal{A}}\left(\epsilon, (r\delta)^{2^h}\right) \;=\; \left(\alpha_\epsilon \;+\; 2^h \cdot \beta_\epsilon \log_2 \frac{1}{r\delta}\right)^k \quad . \tag{5.10}$$

For base learning algorithms $\mathcal{A}$ with runtime $\mathcal{T}_{\mathcal{A}}(n)$ polynomial in the data size $n \in \mathbb{N}$, i.e., $\mathcal{T}_{\mathcal{A}}(n) \in \mathcal{O}(n^\kappa)$ with $\kappa \in \mathbb{N}$, we now determine the runtime $\mathcal{T}_{\mathcal{R},h}(N)$ of the Radon machine with h iterations and $c = r^h$ processing units on $N \in \mathbb{N}$ samples. In this case all base learning algorithms can be executed in parallel. In practical applications fewer physical processors can be used to simulate r^h processing units—we discuss this case in Section 5.5.

The runtime of the Radon machine can be decomposed into the runtime of the base learning algorithm and the runtime for the aggregation. The base learning algorithm requires $n \geq N_0^{\mathcal{A}}(\epsilon, \delta)$ samples and can be executed on r^h processors in parallel in time $\mathcal{T}_{\mathcal{A}}(n)$. The Radon point in each of the h iterations can then be calculated in parallel in time r^3. Thus, the runtime of the Radon machine with $N = r^h n$ samples is

$$\mathcal{T}_{\mathcal{R},h}(N) = \mathcal{T}_{\mathcal{A}}(n) + hr^3 \quad . \tag{5.11}$$

In contrast, the runtime of the base learning algorithm for achieving the same guarantee is $\mathcal{T}_{\mathcal{A}}(M)$ with $M \geq N_0^{\mathcal{A}}(\epsilon, \Delta)$. Ignoring logarithmic and constant terms, $N_0^{\mathcal{A}}(\epsilon, \Delta)$ behaves as $2^h N_0^{\mathcal{A}}(\epsilon, \delta)$. To obtain polylogarithmic runtime of $\mathcal{R}$ compared to $\mathcal{T}_{\mathcal{A}}(M)$, we choose the parameter $h \approx \log_2 M - \log_2 \log_2 M$ such that $n \approx {}^M\!/\!_{2^h} = \log_2 M$. Thus, the runtime of the Radon machine is in $\mathcal{O}\left(\log_2^\kappa M + r^3 \log_2 M\right)$. This result is formally summarized in Theorem 5.12.

Theorem 5.12. *The Radon machine with a consistent and efficient regularized risk minimization algorithm on a model space with finite Radon number has polylogarithmic runtime on quasi-polynomially many processing units if the Radon number is upper bounded by a function polylogarithmic in the sample complexity of the efficient regularized risk minimization algorithm.*

[3] We derive $\alpha_\epsilon, \beta_\epsilon$ for model spaces with finite VC (Vapnik and Chervonenkis, 1971) and Rademacher (Bartlett and Mendelson, 2003) complexity in Appendix B.3.

Proof. We assume the base learning algorithm $\mathcal{A}$ to be a consistent and efficient regularized risk minimization algorithm on a model space with finite Radon number. Let $r \in \mathbb{N}$ be the Radon number of the model space and

$$N_0^{\mathcal{A}}(\epsilon, \delta) = \left(\alpha_\epsilon + \beta_\epsilon \log_2 \frac{1}{\delta}\right)^k$$

be its sample complexity with $\alpha_\epsilon, \beta_\epsilon \geq 0$. In the following, we want to compare the runtime of the Radon machine for achieving an (ϵ, Δ)-guarantee to the runtime of the sequential application of the base learning algorithm for achieving the same (ϵ, Δ)-guarantee.

To achieve an (ϵ, Δ)-guarantee, the Radon machine with parameter $h \in \mathbb{N}$ requires $N = nr^h$ examples (i.e., with r^h processing units), where n denotes the size of the data subset available to each parallel instance of the base learning algorithm. Since $\Delta = (r\delta)^{2^h}$, each base learning algorithm needs to achieve an (ϵ, δ)-guarantee and thus requires

$$n = \left\lceil N_0^{\mathcal{A}}(\epsilon, \delta) \right\rceil \leq \left(\alpha_\epsilon + \beta_\epsilon \log_2 \frac{1}{\delta}\right)^k + 1 \tag{5.13}$$

examples. The sequential application of the base learning algorithm requires at least (cf. Equation 5.10)

$$M = \left\lceil N_0^{\mathcal{A}}(\epsilon, \Delta) \right\rceil = \left\lceil \left(\alpha_\epsilon \quad + \quad 2^h \cdot \beta_\epsilon \log_2 \frac{1}{r\delta}\right)^k \right\rceil = \left\lceil \left(\alpha_\epsilon \quad + \quad 2^h \left(\beta_\epsilon \log_2 \frac{1}{\delta} - \beta_\epsilon \log_2 r\right)\right)^k \right\rceil \ . \tag{5.14}$$

Solving Equation 5.13 for $\beta_\epsilon \log_2 1/\delta$ yields

$$\beta_\epsilon \log_2 \frac{1}{\delta} \leq (n-1)^{\frac{1}{k}} - \alpha_\epsilon \ .$$

By inserting this into Equation 5.14 we obtain

$$M \geq \left\lceil \left(\alpha_\epsilon + 2^h \left((n-1)^{\frac{1}{k}} - \alpha_\epsilon - \beta_\epsilon \log_2 r\right)\right)^k \right\rceil \in \mathcal{O}\left(2^h \left(n - \log_2 r\right)\right) \ . \tag{5.15}$$

In the following, we show that for the choice of

$$h = \left\lceil \frac{1}{k} \left(\log_2 M - \log_2 \log_2 M\right) \right\rceil \ , \tag{5.16}$$

the runtime of the Radon machine is polylogarithmic in M, i.e., polylogarithmic in the number of examples the sequential application of the base learner requires to achieve an (ϵ, Δ)-guarantee. For that, the Radon machine requires quasi-polynomially many processors in M. Note that the Radon machine processes $N \geq M$ many samples to achieve that (ϵ, Δ)-guarantee, which is more than the sequential application of the base learner requires with $N \in \mathcal{O}\left(r^h/2^{hk} M\right)$.

Thus, we need to express the runtime of the Radon machine, which is

$$\mathcal{T}_{\mathcal{R},h}(N) = \mathcal{T}_{\mathcal{A}}\left(\frac{N}{r^h}\right) + r^3 \log_r r^h = \mathcal{T}_{\mathcal{A}}(n) + r^3 \log_r r^h \ ,$$

in terms of M instead of N. First, we express n in terms of M, by solving Equation 5.15 for n which yields

$$n \leq \left(\left(\alpha_\epsilon\left(1-\frac{1}{2^h}\right)+\beta_\epsilon \log_2 r+\frac{1}{2^h}M^{\frac{1}{k}}\right)^k+1\right) \in \mathcal{O}\left(\log_2^k r+\frac{1}{2^{hk}}M\right) \quad . \tag{5.17}$$

Since $\mathcal{A}$ is efficient, $\mathcal{T}_\mathcal{A}(n) \in \mathcal{O}(n^\kappa)$ and thus the runtime of the Radon machine in terms of M, denoted $\mathcal{T}_\mathcal{R}^M$, is

$$\mathcal{T}_\mathcal{R}^M = \mathcal{T}_\mathcal{A}(n) + r^3 \log_r r^h \in \mathcal{O}\left(\left(\log_2^k r+\frac{1}{2^{hk}}M\right)^\kappa + r^3 \log_2 r^h\right) \quad .$$

Inserting h as in Equation 5.16 yields

$$\begin{aligned}\left(\log_2^k r+\frac{1}{2^{hk}}M\right)^\kappa + r^3\log_2\frac{M}{\log_2 M} &= \left(\log_2^k r+\frac{M}{2^{k\frac{1}{k}\log_2\frac{M}{\log_2 M}}}\right)^\kappa + r^3\log_2\frac{M}{\log_2 M}\\ &= \left(\log_2^k r+\frac{M}{\frac{M}{\log_2 M}}\right)^\kappa + r^3\log_2\frac{M}{\log_2 M}\\ &= \left(\log_2^k r+\log_2 M\right)^\kappa + r^3\log_2\frac{M}{\log_2 M} \quad .\end{aligned}$$

This shows that

$$\mathcal{T}_\mathcal{R}^M \in \mathcal{O}\left(\log_2^\kappa M + \log_2^{k\kappa} r + r^3 \log_2 M\right) \quad .$$

Thus, the runtime of the Radon machine to achieve an (ϵ,Δ)-guarantee in terms of M (i.e., the number of samples required by the sequential application of the base learning algorithm) is in $\mathcal{O}\left(\log_2^\kappa M + \log_2^{k\kappa} r + r^3 \log_2 M\right)$ and thus polylogarithmic in M.

We now determine the number of processing units $c = r^h$ in terms of M. For that, observe that h as in Equation 5.16 can be expressed as

$$h = \left\lceil \frac{1}{k}\left(\log_2 M - \log_2\log_2 M\right)\right\rceil = \left\lceil \frac{1}{k}\left(\log_2 \frac{M}{\log_2 M}\right)\right\rceil = \left\lceil \frac{\log_2 r}{k}\log_r\frac{M}{\log_2 M}\right\rceil \quad ,$$

and thus the number of processing units is

$$c = r^h \in \mathcal{O}\left(M^{\log_2 r}\right) \quad .$$

□

As mentioned above, for the Radon machine to achieve an (ϵ,Δ)-guarantee the base learning algorithm has to achieve $\delta \leq {}^1\!/_{2r}$. Thus, the sample size with respect to M has to be large enough so that each base learner achieves this minimum confidence. The base learning algorithm achieves the minimum confidence for $M \geq 2^{k\beta_\epsilon(\alpha_\epsilon+1)}$: Equation 5.14 implies that for each instance of the base learning algorithm to achieve $\delta \leq {}^1\!/_{2r}$ it is required that

$$M \geq \left(\alpha_\epsilon + 2^h\cdot\beta_\epsilon\log_2\frac{1}{r^{\frac{1}{2r}}}\right)^k = \left(\alpha_\epsilon+2^h\beta_\epsilon\right)^k = \left(\alpha_\epsilon + \left(\frac{M}{\log_2 M}\right)^{\frac{1}{k}}\beta_\epsilon\right)^k \quad . \tag{5.18}$$

With $M \geq 2^{k\beta_\epsilon(\alpha_\epsilon+1)} \geq 2^{k\beta_\epsilon}$ we have

$$\left(\frac{M}{\log_2 M}\right)^{\frac{1}{k}} \beta_\epsilon \geq 1 \ ,$$

and thus we can upper bound the right hand side of Equation 5.18 by

$$\begin{aligned}\left(\alpha_\epsilon + \left(\frac{M}{\log_2 M}\right)^{\frac{1}{k}} \beta_\epsilon\right)^k &= \left(\left(\frac{M}{\log_2 M}\right)^{\frac{1}{k}} \beta_\epsilon \left(\frac{\alpha_\epsilon}{\left(\frac{M}{\log_2 M}\right)^{\frac{1}{k}} \beta_\epsilon} + 1\right)\right)^k \\ &\leq \frac{M}{\log_2 M} \beta_\epsilon^k (\alpha_\epsilon + 1)^k \underbrace{\leq}_{\log_2 M \geq \beta_\epsilon^k (\alpha_\epsilon+1)^k} M \ .\end{aligned}$$

The proof of Theorem 5.12 relates to Nick's Class (Arora and Barak, 2009): A decision problem can be solved efficiently in parallel in the sense of Nick's Class, if it can be decided by an algorithm in polylogarithmic time on polynomially many processors (assuming, e.g., PRAM model). For the class of decision problems that are the hardest in P, i.e., for P-complete problems, it is believed that there is no efficient parallel algorithm for solving them in this sense. Theorem 5.12 provides a step towards finding efficient parallelisations of regularised risk minimisers and towards answering the open question: is consistent regularised risk minimisation possible in polylogarithmic time on polynomially many processors. While Nick's Class as a notion of efficiency has been criticized (Kruskal et al., 1990), it is the only notion of efficiency that forms a proper complexity class in the sense of Blum (1967). To overcome the weakness of using only this notion, Kruskal et al. (1990) suggested to consider also the inefficiency of simulating the parallel algorithm on a single processing unit.

The following section analyses the inefficiency and speedup of the Radon machine.

5.3.1. Speedup and Inefficiency

This section determines the speedup of the Radon machine over the sequential execution of the base learning algorithm when both achieve the same (ϵ, Δ)-guarantee. For that, recall that the sample complexity of the base learning algorithm for a given $\epsilon > 0$, $0 < \Delta < 1$ is

$$N_0^{\mathcal{A}}(\epsilon, \Delta) = \left(\alpha_\epsilon + \beta_\epsilon \log_2 \frac{1}{\Delta}\right)^k \ .$$

Assuming that $\alpha_\epsilon \in \Theta(\epsilon^{-1})$ and $\beta_\epsilon \in \Theta(\epsilon^{-1})$ (see for example Lemma A.4 and Lemma A.5), and following the notion of Hanneke (2016) the sample complexity can be expressed as

$$N_0^{\mathcal{A}}(\epsilon, \Delta) \in \Theta\left(\left(\frac{1}{\epsilon} + \frac{1}{\epsilon}\log_2 \frac{1}{\Delta}\right)^k\right) = \Theta\left(\left(\frac{1}{\epsilon}\log_2 \frac{1}{\Delta}\right)^k\right) \ . \tag{5.19}$$

Similar to Kruskal et al. (1990), we assume the base algorithm to have a runtime polynomial in N, i.e.,

$$\mathcal{T}_{\mathcal{A}} \in \Theta\left(N^\kappa\right) = \Theta\left(\left(\frac{1}{\epsilon}\log_2 \frac{1}{\Delta}\right)^{k\kappa}\right) \ . \tag{5.20}$$

The Radon machine runs $\mathcal{A}$ in parallel on c processors to obtain r^h weak models with (ϵ,δ)-guarantee. It then combines the obtained solutions h times—level-wise in parallel—calculating the Radon point (which takes time r^3). In this paper we assume the number of available processors to be abundant and thus set $c = r^h$. With this, the runtime of the Radon machine is

$$\mathcal{T}_{\mathcal{R}} \in \Theta\left(\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa} + hr^3\right) \ . \tag{5.21}$$

We now provide an analysis on the speed-up for $c = r^h$ and arbitrary $h \in \mathbb{N}$.

Proposition 5.22. *Given a polynomial time consistent regularized risk minimization algorithm $\mathcal{A}$ using a model space with finite Radon number $r \in \mathbb{N}$ and runtime as in Equation 5.20, the Radon machine run with parameter $h \in \mathbb{N}$ on r^h processors. Then, the ratio of the runtime of the base learner over the runtime of the Radon machine, denoted the speed-up (Kruskal et al., 1990)*

$$\frac{\mathcal{T}_{\mathcal{A}}}{\mathcal{T}_{\mathcal{R}}} \ ,$$

is in

$$\Theta\left(\frac{2^{hk\kappa}}{1+\frac{hr^3}{\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}}}\right) \ .$$

Proof. In order to achieve an (ϵ,Δ)-guarantee, the Radon machine runs r^h parallel instances of the the base learning algorithm on $n = \lceil N_0^{\mathcal{A}}(\delta)\rceil$ examples with $\delta \leq 1/2r$ so that $\Delta = (r\delta)^{2^h}$. To achieve the same (ϵ,Δ)-guarantee, the sequential execution of the base learning algorithm requires

$$M = \lceil N_0^{\mathcal{A}}(\Delta)\rceil = \left\lceil\left(2^h\cdot\frac{1}{\epsilon}\log_2\frac{1}{r\delta}\right)^k\right\rceil \in \Theta\left(\left(2^h\frac{1}{\epsilon}\log_2\frac{1}{r\delta}\right)^k\right) = \Theta\left(\left(2^h\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^k\right) \ .$$

The last step follows from the fact that, since $\delta \leq 1/2r$, we have $1/r\delta \geq 2r/r \geq r$ and thus

$$\begin{aligned}\log_2\frac{1}{r\delta} \leq \log_2\frac{1}{r\delta} + \log_2 r = \log_2\frac{1}{\delta} \leq 2\log_2\frac{1}{r\delta}\\ \Rightarrow \log_2\frac{1}{\delta} \in \Theta\left(\log_2\frac{1}{r\delta}\right) \Leftrightarrow \log_2\frac{1}{r\delta} \in \Theta\left(\log_2\frac{1}{\delta}\right) \ ,\end{aligned}$$

To achieve the (ϵ,Δ)-guarantee, the base learning algorithm has a runtime of

$$\mathcal{T}_{\mathcal{A}} \ \in \ \Theta\left(M^\kappa\right) \ = \ \Theta\left(\left(2^h\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}\right) \ .$$

Using $\mathcal{T}_{\mathcal{R}}$ from Equation 5.21, we get that

$$\frac{\mathcal{T}_{\mathcal{R}}}{\mathcal{T}_{\mathcal{A}}} \in\Theta\left(\frac{\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa} + hr^3}{\left(2^h\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}}\right) = \Theta\left(\frac{1}{2^{hk\kappa}}\left(1+\frac{hr^3}{\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}}\right)\right)$$

The speed-up then is

$$\frac{\mathcal{T}_\mathcal{A}}{\mathcal{T}_\mathcal{R}} \in \Theta\left(\frac{2^{hk\kappa}}{1+\frac{hr^3}{\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}}}\right) \ .$$

□

Note that the runtime of the Radon machine for the case that $1 \leq c \leq r^h$ is given by

$$\mathcal{T}_\mathcal{R} \in \Theta\left(\frac{r^h}{c}\left(\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^{k\kappa}\right) + r^3\sum_{i=1}^{h}\left\lceil\frac{r^i}{c}\right\rceil\right) \ .$$

In this case, the speed-up is lower by a factor of r^h/c.

In the following, we analyse the inefficiency (Kruskal et al., 1990) of the Radon machine, i.e., the ratio between the total number of operations executed by all processors, and the work of the sequential algorithm.

Proposition 5.23. *The Radon machine with a consistent and efficient regularized risk minimization algorithm on a model space with finite Radon number has quasi-polynomial inefficiency if the Radon number is upper bounded by a function polylogarithmic in the sample complexity of the efficient regularised risk minimisation algorithm.*

Proof. Let $\mathcal{A}$ be a consistent and efficient regularised risk minimisation algorithm on a model space with finite Radon number $r \in \mathbb{N}$. Since $\mathcal{A}$ is efficient, its runtime $\mathcal{T}_\mathcal{A}(M)$ is in $\mathcal{O}(M^\kappa)$. From the proof of Theorem 5.12 follows that, when choosing $h = \left\lceil\frac{1}{k}\left(\log_2 M - \log_2\log_2 M\right)\right\rceil$ the Radon machine has a runtime in $\mathcal{O}\left(\log_2^\kappa M + \log_2^{k\kappa} r + r^3\log_2 M\right)$ using $\mathcal{O}\left(M^{\log_2 r}\right)$ processing units. The inefficiency of the Radon machine then is in

$$\mathcal{O}\left(\frac{M^{\log_2 r}\left(\log_2^\kappa M + \log_2^{k\kappa} r + r^3\log_2 M\right)}{M^\kappa}\right) \in \mathcal{O}\left(M^{(\log_2 r)-\kappa}\log_2^\kappa M\right) = \mathcal{O}\left(M^{\log_2 r}\right) \ .$$

Thus, the inefficiency of the Radon machine is quasi-polynomially bounded or, for short, it has quasi-polynomial inefficiency. □

In order to achieve the same (ϵ, Δ)-guarantee as the base learning algorithm, the Radon machine requires more data. In the following, we analyze the data inefficiency, i.e., the ratio of the data required by the Radon machine over the data required by the base learning algorithm $N_\mathcal{R}(\epsilon,\Delta)/N_\mathcal{A}(\epsilon,\Delta)$.

Proposition 5.24. *The Radon machine with a consistent and efficient regularized risk minimization algorithm $\mathcal{A}$ with sample complexity $N_\mathcal{A}(\epsilon,\Delta)$ on a model space with finite Radon number $r \in \mathbb{N}$ has a data inefficiency in*

$$\Theta\left(\left(\frac{M}{\log_2 M}\right)^{\frac{\log_2 r}{k}}\right) \ ,$$

where $M = \lceil N_\mathcal{A}(\epsilon,\Delta)\rceil$.

Proof. We assume the sample complexity can be expressed as in Equation 5.19. For $\Delta = (r\delta)^{2^h}$ we have that

$$\begin{aligned}
N_{\mathcal{R}}(\epsilon,\Delta) =& r^h N_{\mathcal{A}}(\epsilon,\delta) \in \Theta\left(r^h\left(\frac{1}{\epsilon}\log_2\frac{1}{\delta}\right)^k\right)\\
=&\Theta\left(r^h\left(\frac{1}{2^h}\frac{1}{\epsilon}\log_2\frac{1}{\Delta}\right)^k\right) = \Theta\left(\frac{r^h}{2^{hk}}\left(\frac{1}{\epsilon}\log_2\frac{1}{\Delta}\right)^k\right)\\
=&\Theta\left(\frac{r^h}{2^{hk}}N_{\mathcal{A}}(\epsilon,\Delta)\right) = \Theta\left(\frac{r^h}{2^{hk}}M\right) \enspace .
\end{aligned}$$

Thus, the data inefficiency is in

$$\Theta\left(\frac{r^h}{2^{hk}}\right) \enspace .$$

Choosing $h = \lceil k^{-1}(\log_2 M - \log_2\log_2 M)\rceil$ as in the proof of Theorem 5.12, this is in

$$\Theta\left(\frac{r^{\frac{1}{k}\log_2 r \log_r \frac{M}{\log_2 M}}}{2^{k\frac{1}{k}\log_2\frac{M}{\log_2 M}}}\right) = \Theta\left(\frac{\left(\frac{M}{\log_2 M}\right)^{\frac{\log_2 r}{k}}}{\frac{M}{\log_2 M}}\right) = \Theta\left(\left(\frac{M}{\log_2 M}\right)^{\frac{\log_2 r}{k}}\right)$$

□

After having determined the speed-up for achieving the same theoretical guarantees, the following section empirically studies the predictive quality and speedup of the Radon machine in scenarios with a fixed, finite dataset in realistic scenarios.

5.4. Empirical Evaluation

This empirical study compares the Radon machine to state-of-the-art parallel machine learning algorithms from the Spark machine learning library (Meng et al., 2016), as well as the natural baseline of averaging models instead of calculating their Radon point, denoted averaging-at-the-end (**Avg**). In this study, we use base learning algorithms from WEKA (Witten et al., 2016) and scikit-learn (Pedregosa et al., 2011). We compare the Radon machine to the base learning algorithms on moderately sized datasets, due to scalability limitations of the base learners, and reserve larger datasets for the comparison with parallel learners. The experiments are executed on a Spark cluster (5 worker nodes, 25 processors per node)[4]. In this study, we apply the Radon machine with parameter $h = 1$ and the maximal parameter h (denoted $h = max$) such that each instance of the base learning algorithm is executed on a subset of size at least 100. Averaging-at-the-end uses the same parameter h and executes the base learning algorithm on r^h subsets, i.e., the same number as the Radon machine with that parameter.

[4] The source code implementation in Spark can be found in the bitbucket repository `https://bitbucket.org/Michael_Kamp/radonmachine`.

What is the speed-up of our scheme in practice? In Figure 5.2(a), we compare the Radon machine to its base learners on moderately sized datasets (details on the datasets are provided in Table 5.1). There, the Radon machine is between 80 and around 700-times faster than the base learner using 150 processors. The speed-up is detailed in Figure 5.3. On the SUSY dataset (with 5 000 000 instances and 18 features), the Radon machine on 150 processors with $h = 3$ is 721 times faster than its base learning algorithms. At the same time, their practical performances, measured by the area under the ROC curve (AUC) on an independent test dataset, are comparable.

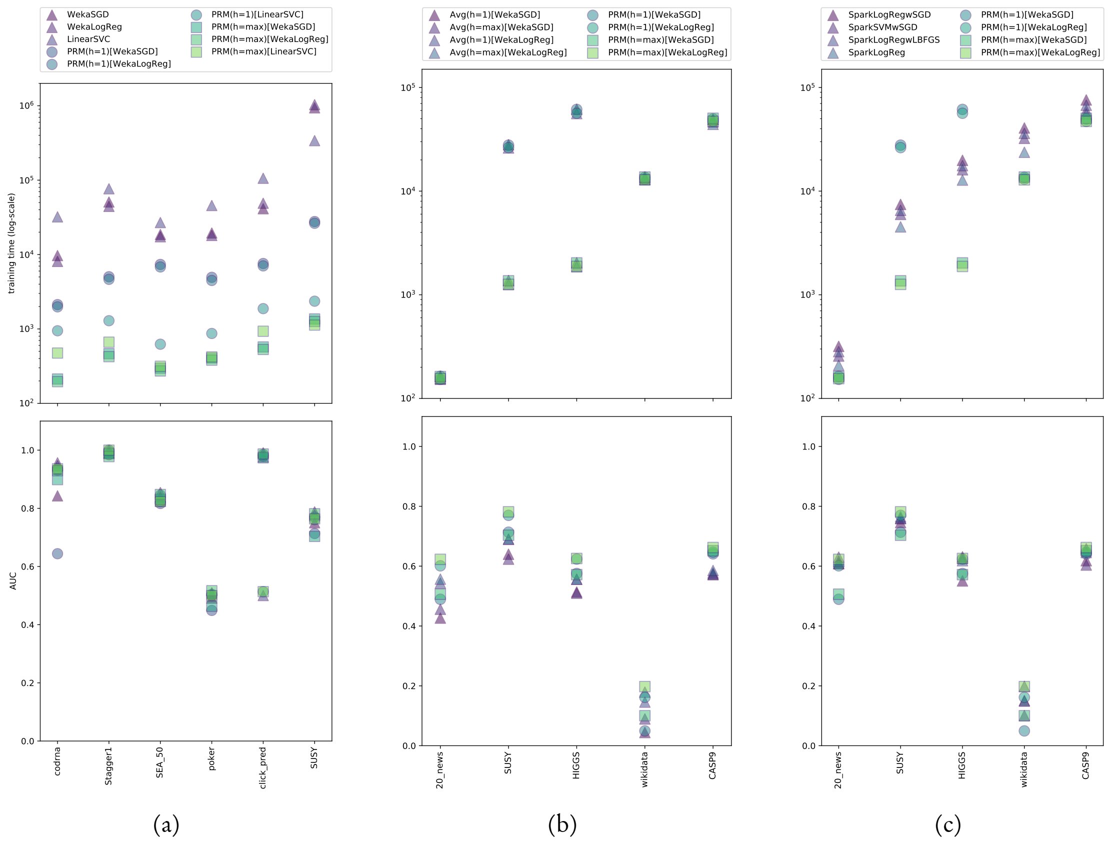


Figure 5.2.: (a) Runtime (log-scale) and AUC of base learners and their parallelisation using the Radon machine (PRM) for 6 datasets with $N \in [488\,565, 5\,000\,000]$, $d \in [3, 18]$. Each point represents the average runtime (upper part) and AUC (lower part) over 10 folds of a learner—or its parallelization—on one datasets. (b) Runtime and AUC of the Radon machine compared to the averaging-at-the-end baseline (Avg) on 5 datasets with $N \in [5\,000\,000, 32\,000\,000]$, $d \in [18, 2\,331]$. (c) Runtime and AUC of several Spark machine learning library algorithms and the Radon machine using base learners that are comparable to the Spark algorithms on the same datasets as in Figure 5.2(b).

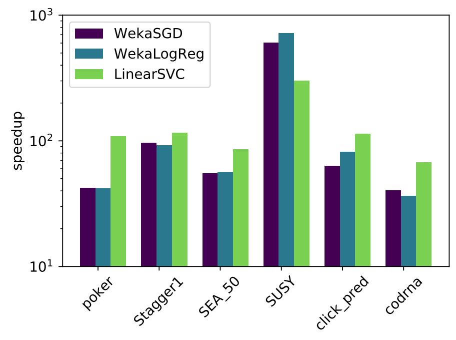


Figure 5.3.: Speed-up (log-scale) of the Radon machine over its base learners per dataset from the same experiment as in Figure 5.2(a).

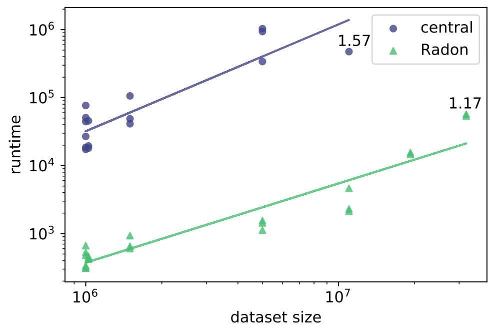


Figure 5.4.: Dependence of the runtime on the dataset size for of the Radon machine compared to its base learners.

How does the scheme compare to averaging-at-the-end? In Figure 5.2(b) we compare the runtime and AUC of the parallelisation scheme against the averaging-at-the-end baseline (Avg). Since averaging is less computationally expensive than calculating the Radon point, the runtimes of the averaging-at-the-end baselines are slightly lower than the ones of the Radon machine. However, compared to the computational complexity of executing the base learner, this advantage becomes negligible. In terms of AUC, the Radon machine outperforms the averaging-at-the-end baseline on all datasets by at least 10%.

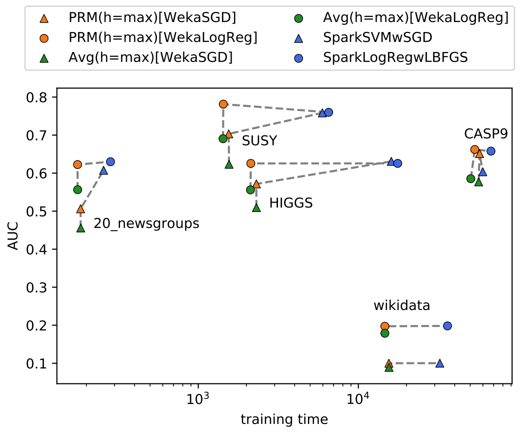


Figure 5.5.: Representation of the results in Figure 5.2(b) and 5.2(c) in terms of the trade-off between runtime and AUC for the Radon machine (PRM) and averaging-at-the-end (Avg), both with parameter $h = max$, and parallel machine learning algorithms in Spark.

How does our scheme compare to state-of-the-art Spark machine learning algorithms? We compare the Radon machine to various Spark machine learning algorithms on 5 large datasets. The results in Figure 5.2(c) indicate that the proposed parallelization scheme with $h = max$ has a significantly smaller runtime than the Spark algorithms on all datasets. On the SUSY and HIGGS dataset, the Radon machine is one order of magnitude faster than the Spark implementations—here the comparatively small number of features allows for a high level of parallelism. On the CASP9 dataset, the Radon machine is 15% faster than the fastest Spark algorithm. The performance in terms of AUC of the Radon machine is similar to the Spark algorithms. In particular, when using WekaLogReg with $h = max$, the Radon machine outperforms the Spark algorithms in terms of AUC and runtime on the datasets SUSY, wikidata, and CASP9. Details are given in the following Section 5.4.1. A summarizing comparison of the parallel approaches in terms of their trade-off between runtime and predictive performance is depicted in Figure 5.5. Here, results are shown for the Radon machine and averaging-at-the-end with parameter $h = max$ and for the two Spark algorithms most similar to the base learning algorithms. Note that it is unclear, what caused the consistently weak performance of all algorithms on wikidata. Nonetheless, the results show that on all datasets the Radon machine has comparable predictive performance to the Spark algorithms and substantially higher predictive performance than averaging-at-the-end. At the same time, the Radon machine has a runtime comparable to averaging-at-the-end on all datasets, both are substantially faster than the Spark algorithms.

How does the runtime depend on the dataset size in a real-world system? In Figure 5.4 we compare the runtimes of all base learning algorithms per dataset size to the Radon machines. Results indicate that, while the runtimes of the base learning algorithms depends on the dataset size with an average exponent of 1.57, the runtime of the Radon machine depends on the dataset size with an exponent of only 1.17. This is plausible because with enough processors the generation of weak models can be done completely in parallel. Moreover, the time for aggregating the models does not depend on the number of instances in the dataset, but only on the number of iterations and the dimension of the model space.

How generally applicable is the scheme? As an indication of the general applicability in practice, we apply the scheme to a Scikit-learn implementation of regularized least squares regression (Pedregosa et al., 2011). On the dataset *YearPredictionMSD*, regularized least squares regression achieves an RMSE of 12.57, whereas the Radon machine achieved an RMSE of 13.64. At the same time, the Radon machine is 197-times faster. We also compare the Radon machine on a multi-class prediction problem using conditional maximum entropy models. We use the implementation described in Mcdonald et al. (2009), who also propose to use averaging-at-the-end for distributed training. We compare the Radon machine to averaging-at-the-end with conditional maximum entropy models on two large multi-class datasets (*drift* and *spoken-arabic-digit*). On average, our scheme performs 4% better with only 0.2% longer runtime. The minimal difference in runtime can be explained—similar to the results in Figure 5.2(b)—by the smaller complexity of calculating the average instead of the Radon point.

5.4.1. Additional Details

This section provides additional details on the experiments conducted. All experiments are performed on a Spark cluster with a master node, 5 worker nodes, 25 processors and 64GB of RAM per node. The Radon machine is applied with parameter $h = 1$ and with the maximal h for a given dataset. Recall, that the number of iterations h is limited by the dataset size (i.e., number of instances) and the Radon number of the model space, since the dataset is partitioned into r^h parts of size n. Thus, given a data set of size N, the maximal h is given by

$$h_{\max} = \left\lfloor \log_r \frac{N}{n_{\min}} \right\rfloor ,$$

where $n_{\min}$ denotes the minimum size of the local subset of data that each instance of the base learner is executed on. The experiments have been carried out with $n_{\min} = 100$. If r^h is larger than the actual number of processing units, some instances of the base learner are executed sequentially.

As base learning algorithms we use the WEKA (Witten et al., 2016) implementation of Stochastic Gradient Descent (*WekaSGD*), and Logistic Regression (*WekaLogReg*), as well as a the Scikit-learn implementation of the linear support vector machine (*LinearSVM*) with pyspark. The parallelizations of a base learner using the Radon machine is denoted *PRM(h=?)[<base learner>]*.

We compare the Radon machine to the natural baseline of aggregating models by calculating their average, denoted averaging-at-the-end (*Avg(h=?)[<base learner>]*). Given a parameter $h \in \mathbb{N}$, averaging-at-the-end executes the base learning algorithm on r^h subsets of the data, i.e., on the same number of subsets as the Radon machine. Accordingly, the runtime for obtaining the set of models is similar, but the time for aggregating the models is shorter, since averaging is less computationally expensive than calculating the Radon point.

Name	**Instances**	**Dimensions**	**Output**
click_prediction	1 496 391	11	$\mathcal{Y} = \{-1, 1\}$
poker	1 025 010	10	$\mathcal{Y} = \{-1, 1\}$
SUSY	5 000 000	18	$\mathcal{Y} = \{-1, 1\}$
Stagger1	1 000 000	9	$\mathcal{Y} = \{-1, 1\}$
HIGGS	11 000 000	28	$\mathcal{Y} = \{-1, 1\}$
SEA_50	1 000 000	3	$\mathcal{Y} = \{-1, 1\}$
codrna	488 565	8	$\mathcal{Y} = \{-1, 1\}$
CASP9	31 993 555	631	$\mathcal{Y} = \{-1, 1\}$
wikidata	19 254 100	2331	$\mathcal{Y} = \{-1, 1\}$
20_newsgroups	399 940	1002	$\mathcal{Y} = \{-1, 1\}$
YearPredictionMSD	515 345	90	$\mathcal{Y} \subseteq \mathbb{R}$
drift	13 991	90	$\mathcal{Y} = \{1, \dots, 89\}$
spoken-arabic-digit	263 256	15	$\mathcal{Y} = \{1, \dots, 10\}$

Table 5.1.: Description of the datasets used in our experiments.

We compare the Radon machine to parallel machine learning algorithms from the Spark machine learning library, as well. That is, we compare it to SparkMLLibLogisticRegressionWithLBFGS (*SparkLogRegwLBFGS*), SparkMLLibLogisticRegressionWithSGD (*SparkLogRegwSGD*), SparkMLLibSVMWithSGD (*SparkSVMwSGD*), and SparkMLLogisticRegression (*SparkLogReg*).

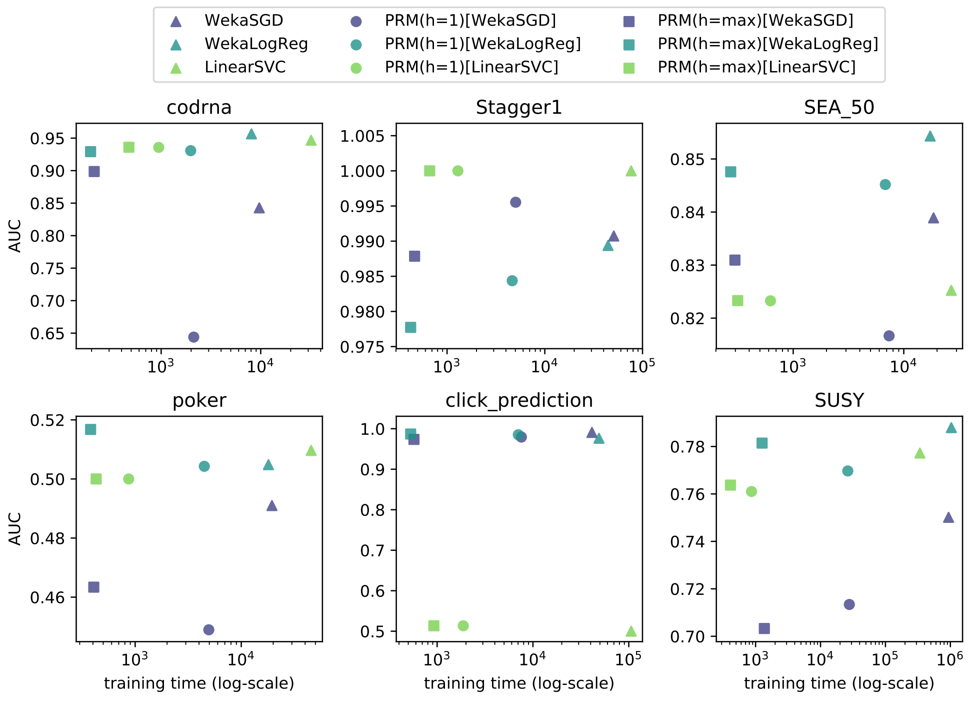


Figure 5.6.: AUC vs. training time for base learning algorithms and their parallelisation with the Radon machine per dataset from the same experiment as in Figure 5.2(a).

The properties of the datasets used in the empirical evaluation are presented in Table 5.1. Datasets have been acquired from OpenML (Vanschoren et al., 2013), the UCI machine learning repository (Lichman, 2013), and Big Data competition of the ECDBL'14 workshop[5]. Experiments on moderately sized datasets—on which we compared the Radon machine to the base learning algorithms executed on the entire dataset have been conducted on the datasets click_prediction, poker, SUSY, Stagger1, SEA_50, and codrna. The comparison of Radon machine and Spark ML learners has been executed on the datasets CASP9, HIGGS, wikidata, 20_newsgroups, and SUSY. The regression experiment was conducted using the YearPredictionMSD dataset, multiclass-prediction experiments using the drift, and spoken-arabic-digit datasets.

In the following, we provide more details on the experiments presented in Figures 5.2(a), 5.2(b), and 5.2(c). In particular, we analysis the trade-off between training time and AUC per dataset.

Figure 5.6 shows the trade-off between training time and AUC for base learning algorithms and their parallelization using the Radon machine. It confirms that the training time for the Radon machine is orders of magnitude smaller than the base learning algorithms on all datasets. Moreover, the training time is substantially smaller for the Radon machine with maximal height ($h = max$), compared to a height of 1. In terms of AUC, the performance of

[5] Big Data Competition 2014: `http://cruncher.ncl.ac.uk/bdcomp/`

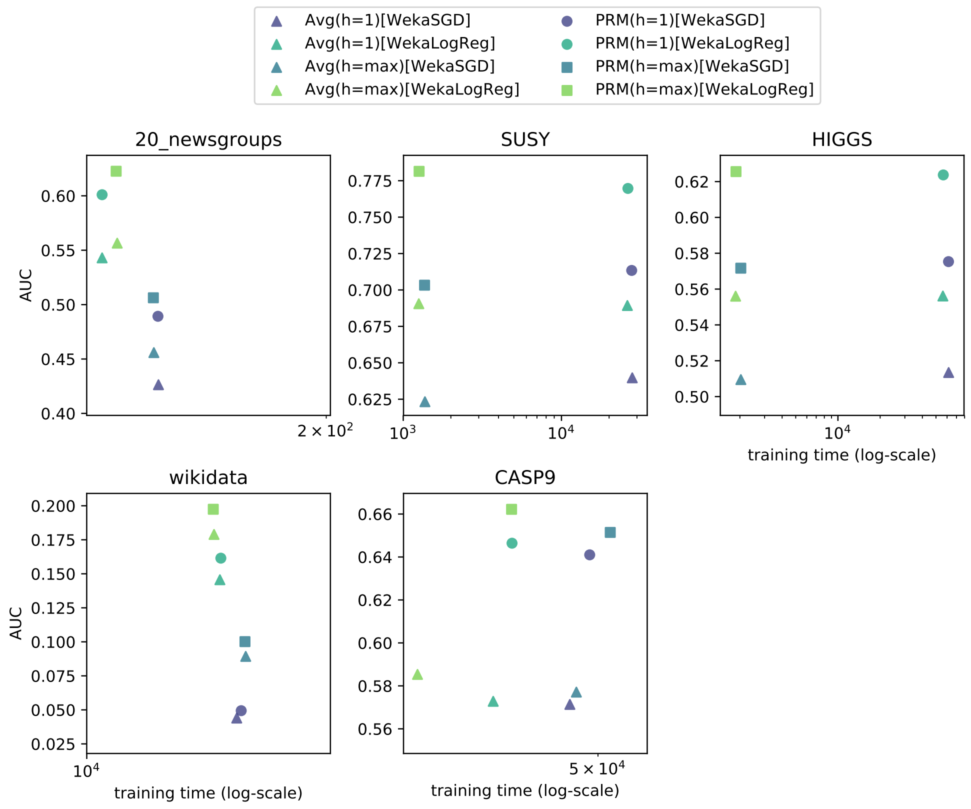


Figure 5.7.: AUC vs. training time for the parallelisation of base learning algorithms using the averaging-at-the-end baseline (*Avg*) and the Radon machine per dataset from the same experiment as in Figure 5.2(b).

the parallelization is comparable to the base learner for WekaLogReg and LinearSVC on all datasets. For the base learner WekaSGD, its parallelization with the Radon machine only has substantially lower AUC on the dataset codrna with parameter $h = 1$.

Figure 5.7 presents the trade-off for the Radon machine compared to the averging-at-the-end baseline. It confirms that the Radon machine has substantially higher AUC than a parallelisation of the same base learning algorithm using the averaging-at-the-end baseline. The schemes only differ in the aggregation operation at the end, so that the difference in training time follows from the faster computation of the average, compared to the iterated Radon point computation.

In Figure 5.8, the trade-off between training time and AUC of the Radon machine compared to the Spark learners is plotted. While it confirms that the Radon machine is always favorable in terms of training time, in terms of AUC the results are mixed. For the base learner WekaLogReg, its parallelization is always among the best in terms of AUC. The parallelization of WekaSGD, however, has worse performance than the Spark learners on 2 out of 5 datasets. It also confirms that for the datasets SUSY and HIGGS, the runtime of the Radon machine with $h = 1$ is substantially larger than for $h = max$. Thus, for the best performance in terms of runtime and AUC, the height should be maximal.

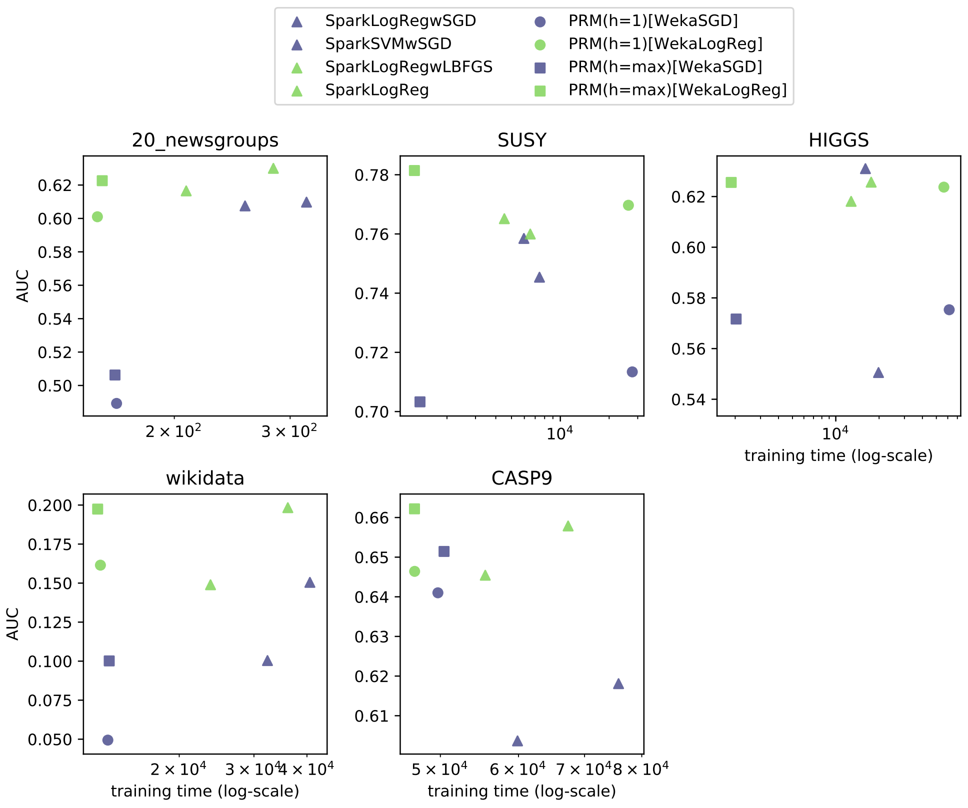
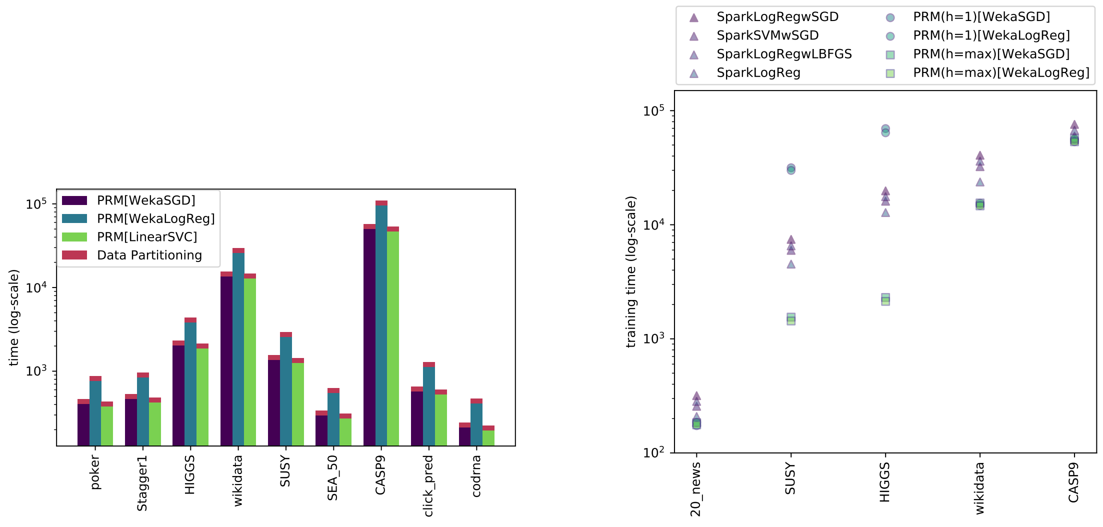


Figure 5.8.: AUC vs. training time for Spark learners and parallelisations of comparable base learning algorithms with the Radon machine per dataset from the same experiment as in Figure 5.2(c).

In order to investigate the results depicted in Figure 5.8 more closely, we provide the training times and AUCs in detail in Table 5.2. As mentioned above, the Radon machine using WekaLogReg as base learner has better runtime than all Spark algorithms. At the same time, this version of the Radon machine outperforms the Spark algorithms in terms of AUC on all datasets but 20_newsgroups—there it is 2.2% worse than the best Spark algorithm. In particular, on the largest dataset in the experiments—the CASP9 dataset with 32 million instances and 631 features—the Radon machine is 15% faster and 2.6% better in terms of AUC than the best Spark algorithm.

Note that for HIGGS and SUSY, the Radon machine with $h = 1$ is an order of magnitude slower than with $h = max$ as well as the Spark algorithms. This follows from the low degree of parallelization, since for $h = 1$ only 20 (for SUSY), respectively 30 (for HIGGS) models



(a) (b)

Figure 5.9.: (a) Runtime of the Radon machine together with the time required for repartitioning the data to fit the parallelisation scheme. (b) Runtime and AUC of several Spark machine learning library algorithms and the Radon machine including the time required for repartitioning the data before training.

have to be generated. Thus, only 20, or 30 of the 150 available processors are used in parallel. At the same time, the amount of data each processor has to process is orders of magnitude larger than for $h = max$.

For the above experiments we assume that the data is already distributed over the nodes in the cluster so that it can directly be processed by the Radon machine. When loading data in Spark, this data is distributed over the worker nodes in subsets, but not necessarily in r^h subsets. In Spark, distributed data is organized in partitions, where each partition corresponds to the subset of data available to one instance of the base learning algorithm. In order to apply the Radon machine to a dataset within the Spark framework, the data needs to re-distributed and partitioned into r^h partitions which is achieved by a method called repartition. In the experiments, it is assumed that the data is already partitioned to make a fair comparison to the Spark learning algorithms which do not require repartitioning. Figure 5.9(a) illustrates the time required for repartitioning a dataset in contrast to the runtime of the Radon machine. Unfortunately, repartitioning in Spark always includes a complete shuffling of the data, requiring communication to redistribute the dataset. This is rather inefficient in our context. Nonetheless, the time required for repartitioning is small compared to the overall runtime—in the worst case it takes 14% of the runtime of the Radon machine. Still, taking into account the time for repartitioning the data shrinks the runtime advantage of the proposed scheme over the Spark algorithms. Figure 5.9(b) shows the runtimes of the Spark algorithms

Dataset	SparkLogReg wSGD	SparkSVM wSGD	PRM(h = 1) [WekaSGD]	PRM(h = max) [WekaSGD]	SparkLogReg wLBFGS	SparkLogReg	PRM(h = 1) [WekaLogReg]	PRM(h = max) [WekaLogReg]
				Runtime				
20_newsgroups	317.7	256.2	163.4	162.5	282.9	208.5	**152.8**	155.4
SUSY	7 439.5	5 961.8	27 781.6	1 363.7	6 526.3	4 516.8	26 299.6	**1259.7**
HIGGS	19 815.1	16 071.9	61 429.5	2 029.7	17 617.4	12 783.6	56 394.2	**1876.2**
wikidata	40 645.8	32 288.5	13 575.7	13 677.3	36 060.1	23 702.0	13 039.5	**12845.5**
CASP9	75 782.4	59 864.7	49 711.5	50 430.6	67 367.3	55 523.5	47 085.1	**47070.1**
				AUC				
20_newsgroups	0.6098	0.6075	0.4893	0.5063	**0.63**	0.6165	0.601	0.6226
SUSY	0.7454	0.7585	0.7134	0.7033	0.76	0.7652	0.7697	**0.7814**
HIGGS	0.5506	**0.631**	0.5753	0.5717	0.6257	0.6181	0.6237	0.6256
wikidata	0.1505	0.1004	0.0494	0.1002	0.1983	0.1489	0.1615	**0.1974**
CASP9	0.6181	0.6037	0.641	0.6514	0.6579	0.6454	0.6464	**0.6622**

Table 5.2.: Runtime and AUC of Spark machine learning library algorithms and the Radon machine using WekaSGD and WekaLogReg as base learning algorithms. The results, reported for each dataset, are the average over all folds in a 10-fold cross-validation. These results correspond to the ones presented in Figure 5.2(c).

compared to the Radon machine—similar to Figure 5.2(c)—but with the time required for repartitioning the data added to the runtime of the *Radon machines*. The Radon machine with $h = max$ remains superior to the Spark algorithms in terms of runtime.

5.5. Discussion

In this chapter we provided a step towards answering an open problem: *Is parallel machine learning possible in polylogarithmic time using a polynomial number of processors only?* This question has been posed for half-spaces by Long and Servedio (2013) and called "a fundamental open problem about the abilities and limitations of efficient parallel learning algorithms". It relates machine learning to Nick's Class of parallelizable decision problems and its variants (Greenlaw et al., 1995). Early theoretical treatments of parallel learning with respect to NC considered *probably approximately correct* (PAC) (Blumer et al., 1989; Valiant, 1984) concept learning. Vitter and Lin (1992) introduced the notion of *NC-learnable* for concept classes for which there is an algorithm that outputs a probably approximately correct model in polylogarithmic time using a polynomial number of processors. In this setting, they proved positive and negative learnability results for a number of concept classes that were previously known to be PAC-learnable in polynomial time. More recently, the special case of learning half spaces in parallel was considered by Long and Servedio (2013) who gave an algorithm for this case that runs on polynomially many processors in time that depends polylogarithmically on the size of the instances but is inversely proportional to a parameter of the learning problem.

Some parallelization schemes also train learning algorithms on small chunks of data and average the found models. While this approach has advantages (Freund et al., 2001; Rosenblatt and Nadler, 2016), current error bounds do not allow a derivation of polylogarithmic runtime (Lin et al., 2017; Shamir et al., 2014; Zhang et al., 2013) and it has been doubted to have any benefit over learning on a single chunk (Shamir and Srebro, 2014). Another popular class of parallel learning algorithms is based on stochastic gradient descent, targeting expected risk minimization directly (Shamir and Srebro, 2014, and references therein). The best so far known algorithm in this class (Shamir and Srebro, 2014) is the distributed mini-batch algorithm (Dekel et al., 2012). This algorithm still runs for a number of rounds inversely proportional to the desired optimization error, hence not in polylogarithmic time. A more traditional approach is to minimize the empirical risk, i.e., an empirical sample-based approximation of the expected risk, using any, deterministic or randomized, optimization algorithm. This approach relies on generalization guarantees relating the expected and empirical risk minimization as well as a guarantee on the optimization error introduced by the optimization algorithm. The approach is readily parallelizable by employing available parallel optimization algorithms (e.g., Boyd et al., 2011). It is worth noting that these algorithms solve a harder than necessary optimization problem and often come with prohibitively high communication cost in distributed settings (Shamir and Srebro, 2014). Recent results improve over these (Ma et al., 2017) but cannot achieve polylogarithmic time as the number of iterations depends linearly on the number of processors.

In the following we discuss properties and limitations of the proposed parallelization scheme.

In the experiments we considered datasets where the number of dimensions is much smaller than the number of instances. **What about high-dimensional models?** The basic version of the parallelization scheme presented in this paper cannot directly be applied to cases in which the size of the dataset is not at least a multiple of the Radon number of the model space. For various types of data such as text, this might cause concerns. However, random projections (Johnson and Lindenstrauss, 1984) or low-rank approximations (Balcan et al., 2016; Oglic and Gärtner, 2017b) can alleviate this problem and are already frequently employed in machine learning. An alternative might be to combine our parallelization scheme with block coordinate descent (Sra et al., 2012). In this case, the scheme can be applied iteratively to subsets of the features.

In the experiments we considered only linear models. **What about non-linear models?** Learning non-linear models causes similar problems to learning high-dimensional ones. In non-parametric methods like kernel methods, for instance, the dimensionality of the optimization problem is equal to the number of instances, thus prohibiting the application of our parallelization scheme. However, similar low-rank approximation techniques as described above have been applied with non-linear kernels (Fine and Scheinberg, 2002). Furthermore, novel methods for speeding up the learning process for non-linear models rely on explicitly constructing an embedding in which a linear model can be learned (Rahimi and Recht, 2007). Using explicitly constructed feature spaces, Radon machines can directly be applied to non-linear models.

We have theoretically analyzed our parallelization scheme for the case that there are enough processors available to find each weak model on a separate processor. **What if there are less than r^h processors?** The parallelization scheme can quite naturally be de-parallelized and partially executed in sequence. For the runtime this implies an additional factor of $\max\{1, r^h/c\}$. Thus, the Radon machine can be applied with any number of processors.

The scheme improves the confidence Δ doubly exponentially in its parameter h but for that it requires the weak models to already achieve a base confidence of $1 - \delta > 1 - 1/2r$. **Is the scheme only applicable in high-confidence domains?** Many application scenarios require high-confidence error bounds, e.g., in the medical domain (Nouretdinov et al., 2011) or in intrusion detection (Sommer and Paxson, 2010). In practice our scheme achieves similar predictive quality much faster than its base learner.

Besides runtime, communication plays an essential role in parallel learning. **What is the communication complexity of the scheme?** As for all aggregation at the end strategies, the overall amount of communication is low compared to periodically communicating schemes. For the parallel aggregation of models, the scheme requires $\mathcal{O}(r^{h+1})$ messages each containing a single model of size $\mathcal{O}(d)$. Furthermore, only a fraction of the data has to be transferred to each processor. Our scheme is ideally suited for inherently distributed data.

Since in a lot of applications data is no longer available as a batch but in the form of data streams, as future work it would be interesting to investigate how the scheme can be applied to distributed data streams. A promising approach is to aggregate models periodically using the Radon machine, similar to the federated learning approach proposed by McMahan et al. (2017). In order to use it as an aggregation operator with dynamic averaging, it has to be shown that the Radon point is central with respect to a distance measure. However, it is an open question which distance measure fulfills this property. Another direction for future work is to apply the scheme to general randomized convex optimization algorithms with unobservable target function.

6. Conclusion

This chapter summarizes the main results on a high-level and gives a final discussion of their character and value. The chapter concludes with an outlook for potential follow-up research.

6.1. Summary

The goal of this thesis is to provide communication-efficient parallelization schemes for a broad class of machine learning algorithms that scale well with the number of employed processing units. This is motivated by the increasing number of distributed, loosely connected data-generating devices and the observation that centralizing all their data is infeasible. As specified in the introductory chapter, these parallelizations should be theoretically sound, i.e., with guaranteed model quality, bounded communication, and high speedup per employed processor (see requirements (R1), (R2), and (R3) in Section 1). To that end, Chapter 2 introduced black-box parallelization as a general framework to craft such schemes by defining an aggregation and synchronization operator that together form a distributed learning protocol. The Chapters 3-5 then presented two protocols that fulfill these requirements, one applicable to incremental learning algorithms, the other also applicable to non-incremental ones.

As seen in Chapters 3 and 4, the approach for incremental learning algorithms allows to achieve optimal model quality, as measured by the regret in case of online learning and by the convergence rate (and subsequently generalization bounds) in case of batch learning. At the same time, by dynamically scheduling the aggregation, the amount of communication required to achieve this quality is bounded linearly in the number of learners and the hardness of the learning problem. This is a non-trivial property, since it requires to jointly monitor a measure of that hardness over all learners without communicating. Using divergence as a proxy for the hardness, local conditions can be designed that allow monitoring it in a communication-efficient way. This is applicable to a broad class of learning algorithms that perform (approximately) regret-proportional convex updates, including SGD, mini-batch SGD, and PA updates, using linear and kernel models, as well as neural networks. For SGD and mini-batch SGD, it could be shown that the speedup is nearly linear in the number of processing units $c \in \mathbb{N}$. That is, the speedup is in $\Theta(c/\log c)$. However, the number of processing units can not be arbitrarily increased, i.e., it should be sublinear in the number of rounds T. Thus, this approach cannot achieve polylogarithmic runtime.

Chapter 5 presents an approach for parallelizing batch learning that is applicable to all empirical risk minimization algorithms with models that have a finite-dimensional representation in the Euclidean space. This scheme also achieves optimal model quality in terms of generalization bounds, using only one round of communication at the end. Since averaging model parameters once at the end cannot achieve this, a novel aggregation operator was introduced, based on iteratively replacing sets of models by their Radon point. This operator allows to improve generalization bounds doubly exponentially in the number of iterations. The speedup scales with $\Theta(c^{\kappa/\log d})$ for a base learning algorithm with runtime of $\Theta(N^{\kappa})$ in the sample size $N \in \mathbb{N}$. Here, $d \in \mathbb{N}$ is the dimension of the model space. For learning algorithms with polynomial runtime, this parallelization achieves polylogarithmic runtime on quasi-polynomially many processing units, thus fulfilling requirements formulated above.

6.2. Discussion

The contributions of this thesis are primarily of a theoretical nature, i.e., they provide tight bounds for the model quality, communication, and speedup of the presented approaches. These contributions are relevant for understanding whether machine learning algorithms can be efficiently parallelized in the sense of Nick's class and its learning-related variants (Greenlaw et al., 1995). The results in this thesis indicate that it is indeed possible to parallelize certain classes of polynomial-time learning algorithms efficiently, i.e., with polylogarithmic runtime on polynomially many processors. The Radon machine achieves polylogarithmic runtime on quasi-polynomially many processors. That is, the number of processors is in $\mathcal{O}(N^{\log d})$, where $N \in \mathbb{N}$ is the sample size and $d \in \mathbb{N}$ is the dimension of the model space. Since the dimension of the model space often depends on the dataset (e.g., for linear models the dimension of the model space is equal to that of the data, for neural networks the number of input neurons equals the data dimension, and for kernel models the dimension is equal to the dataset size), it depends on the input size and thus cannot be treated as a constant. Fixed-dimensional models can be achieved, e.g., by kernel approximations, such as random Fourier features (Rahimi and Recht, 2007) or Nyström approximations (Oglic and Gärtner, 2017a). However, for the approximations to be accurate—and thus the error guarantees to hold—their dimension has to be chosen again with respect to the dataset size. Thus, also in this case the Radon machine requires quasi-polynomially many processing units. It remains an open question whether a generic parallelization for learning algorithms exists that achieves polylogarithmic runtime on polynomially many processing units.

The proposed framework of distributed learning protocols consisting of aggregation and synchronization operators emphasizes the generality of the approach. The presented protocols, together with the existing parallelization approaches that fit this framework (e.g., periodic averaging (Li et al., 2014), federated averaging (McMahan et al., 2017)) indicate that the right choice of aggregation operator allows to achieve strong guarantees on the model quality. The right choice of synchronization operator then allows for communication efficiency. It has yet to be seen how useful this framework is and if it constitutes a novel paradigm for parallelization, i.e., whether aggregation and synchronization operators are indeed a rich source for novel parallelization schemes.

The analysis of the presented protocols relies on geometrical properties of the models and the loss surface, such as convexity. This poses a major challenge for non-convex optimization objectives, such as training neural networks. The analysis in Chapter 4 provides quality guarantees under the assumption that local models remain in a locally convex environment. Recent studies on the error surface of neural networks suggest that in many cases, the error surface consists of larger, locally convex regions (e.g., wide local minima) (Dinh et al., 2017; Nguyen and Hein, 2017). Together with the empirical analysis of McMahan et al. (2017) and the experiments in this thesis, this suggests that by careful initialization neural networks remain within locally convex regions for most parts of the training process. The theoretical findings on the loss surface also indicate that it becomes more benign for very deep and wide network architectures (Choromanska et al., 2015; Nguyen and Hein, 2017). This makes it challenging to apply the Radon machine to deep learning, since for a network with $d \in \mathbb{N}$ parameters it requires $(d+2)^h$ learners with $h \geq 1$ being the number of iterations. For example, parallelizing the training of ResNet-50 (He et al., 2016) with $25.6M$ parameters requires roughly $25.6M$ processors for $h = 1$ to apply the Radon machine which is already infeasible in most cases, and $655 \cdot 10^{12}$ processors for $h = 2$. Thus, even though the error surface of deep neural networks might be benign, the Radon machine is not suitable (in its vanilla form) for parallelizing their training.

In addition to the theoretical contributions, the results of this thesis have practical significance as well. The empirical evaluation has shown that the proposed approaches are relevant, e.g., for maintaining high in-place performance of real-time services, and training of neural networks. A challenge in practice is that the approaches are only applicable in scenarios where training data—including the label—can be inferred locally. This can be a limiting factor, e.g., for speech recognition on mobile phones, where users rarely provide a transcript of their voice recording, so that no label is locally available. Similarly, in object detection for autonomous driving, it is difficult to label objects on the camera images within the car. However, in application scenarios where the label is revealed shortly after a prediction, or it can be inferred, e.g., from user behavior, the approaches are applicable and, as indicated by the experiments, outperform state-of-the-art approaches.

6.3. Outlook

After having discussed the contributions of this thesis, this section highlights potential research topics that might emerge from them. Note that several open research questions that specifically relate to certain parts of this thesis are discussed at the end of the respective chapters. Here, I want to conclude with two major directions for future research.

As discussed in the previous section, the training of neural network poses several challenges to black-box parallelizations: the non-convex loss surface makes it hard to provide guarantees on the model quality and the large dimension of the model space complicates geometric operations. Further studying the nature of the loss surface can lead to a more profound understanding, how and when the aggregation of neural networks improves their quality. In that regard, one interesting finding is that the loss surface contains connected local minima with a non-increasing path towards a global optimum (Draxler et al., 2018; Fukumizu and Amari, 2000; Wessels and Barnard, 1992). It is speculated that the randomness in SGD helps escaping such local minima (Draxler et al., 2018). A natural question is whether black-box parallelization reinforces that effect, since local models spread across the local minimum. Moreover, adding noise to the local models can further strengthen this effect (Adilova et al., 2018). Instead of analyzing the loss surface, Tishby and Zaslavsky (2015) propose to analyze the development of mutual information of neural network layers with the data instances and with the label, conjecturing that a well-trained network filters all information not related to the label through the layers so that the last layers represent a minimal sufficient statistics of the instances. Applying this approach to black-box parallelizations, a natural question is how much mutual information about the label is preserved by aggregation. This might lead to the design of novel aggregation operators that maximize the preserved mutual information. Since mutual information is measured level-wise, this might also lead to an understanding, to what extend a layer-wise aggregation is possible, e.g., by computing the Radon point of each layer individually, thereby alleviating the problem of the required number of learners discussed in the previous section.

The goal of parallelizations is to achieve a good model quality with substantial speedup. To that end, a parallelization of a polynomial time algorithm is considered efficient in the sense of Nick's class $\mathcal{NC}$ and its variants (Greenlaw et al., 1995), if it has polylogarithmic runtime on polynomially many processing units. For decision problems, it was conjectured that $P \neq \mathcal{NC}$ (Arora and Barak, 2009), more precisely, there are decision problems in P that cannot be parallelized efficiently. Through black-box parallelization it might be possible to show that a broad class of learning algorithms can be efficiently parallelized. To that end, powerful aggregation operators have to be developed. The aggregation operators used so far replace local models by a point in their convex hull (e.g., the average (Zinkevich et al., 2010), the Radon point (see Chapter 5), or the geometric median (Minsker et al., 2015)). Gilad-Bachrach et al. (2004) provide a motivation for that: they show for linear models that the Bayes point (Herbrich et al., 2001) has a generalization error close to the Bayes optimal classifier and that furthermore the Bayes point is almost identical to the Tukey median (Tukey, 1975) which in turn is a center point of the models. The aggregation of the Radon machine is based on the iterated Radon point algorithm (Clarkson et al., 1996) that approximates the center point. A natural first question is whether computing the Tukey median directly is a better aggregation operator than that used by the Radon machine. The Tukey median of $N \in \mathbb{N}$ points in $\mathbb{R}^d$ can be computed in time $\mathcal{O}(N^{d-1} + N \log N)$ (Chan, 2004) which does not allow for polylogarithmic runtime. It remains an open problem whether the exact Tukey median can be efficiently computed in parallel. A middle ground between the Radon point and computing the Tukey median exactly is the Tverberg point (Tverberg, 1981) which is closer to the Tukey median than the Radon point but is more computationally expensive. Further investigating the connection between these geometric properties and machine learning might lead to a deeper understanding of the capabilities of black-box parallelization.

Bibliography

Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

Linara Adilova, Nathalie Paul, and Peter Schlicht. Introducing noise in decentralized training of neural networks. In *Communications in Computer and Information Science*, 2018.

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

Hande Alemdar and Cem Ersoy. Wireless sensor networks for healthcare: A survey. *Computer networks*, 54(15):2688–2710, 2010.

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

Apache Software Foundation. Storm, 2017. URL `https://storm.apache.org`.

Sanjeev Arora and Boaz Barak. *Computational complexity: A modern approach*. Cambridge University Press, 2009.

Peter Auer, Mark Herbster, and Manfred K Warmuth. Exponentially many local minima for single neurons. In *Advances in Neural Information Processing Systems*, pages 316–322, 1996.

Maria Florina Balcan, Avrim Blum, Shai Fine, and Yishay Mansour. Distributed learning, communication complexity and privacy. In *Proceedings of the 25th Annual Conference on Learning Theory*, volume 23, pages 26.1–26.22. PMLR, 2012.

Maria Florina Balcan, Yingyu Liang, Le Song, David Woodruff, and Bo Xie. Communication efficient distributed kernel principal component analysis. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 725–734, 2016.

Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3:463–482, 2003.

Christian Berger and Michael Dukaczewski. Comparison of architectural design decisions for resource-constrained self-driving cars-a multiple case-study. *Informatik 2014*, 2014.

Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM (JACM)*, 14(2):322–336, 1967.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)*, 36(4):929–965, 1989.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

Nader H. Bshouty and Philip M. Long. Linear classifiers are nearly optimal when hidden variables have diverse effects. *Machine Learning*, 86(2):209–231, 2012.

Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2): 1153–1176, 2016.

Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4): 1547–1621, 2015.

Richard Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

Volkan Cevher, Steffen Becker, and Martin Schmidt. Convex optimization for big data: Scalable, randomized, and parallel algorithms for big data analytics. *Signal Processing Magazine, IEEE*, 31(5):32–43, 2014.

Phillip K Chan, Salvatore J Stolfo, et al. Toward parallel and distributed learning by meta-learning. In *AAAI workshop in Knowledge Discovery in Databases*, pages 227–240, 1993.

Timothy M Chan. An optimal randomized algorithm for maximum tukey depth. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 430–436. Society for Industrial and Applied Mathematics, 2004.

Kamalika Chaudhuri, Claire Monteleoni, and Anand D Sarwate. Differentially private empirical risk minimization. *Journal of Machine Learning Research*, 12(Mar):1069–1109, 2011.

Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 2722–2730. IEEE, 2015.

Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. In *International Conference on Learning Representations Workshop Track*, 2016.

Weizhu Chen, Zhenghao Wang, and Jingren Zhou. Large-scale l-bfgs using mapreduce. In *Advances in Neural Information Processing Systems 27*, pages 1332–1340. Curran Associates, Inc., 2014.

Anna Choromanska, MIkael Henaff, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38, pages 192–204. PMLR, 2015.

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, volume 6, pages 281–288. Vancouver, BC, 2006.

Kenneth L Clarkson, David Eppstein, Gary L Miller, Carl Sturtivant, and Shang-Hua Teng. Approximating center points with iterative Radon points. *International Journal of Computational Geometry & Applications*, 6(03):357–377, 1996.

Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 1337–1345. PMLR, 2013.

Thomas M Cover. Behavior of sequential predictors of binary sequences. In *Proceedings of the 4th Prague Conference on Information Theory, Statistical Decision Functions, Random Processes*, pages 263–272. Publishing House of the Czechoslovak Academy of Sciences, Prague, 1965.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.

Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(1):165–202, 2012.

James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1): A206–A239, 2012.

Don Dennis, Chirag Pabbaraju, Harsha Vardhan Simhadri, and Prateek Jain. Multiple instance learning for efficient sequential data classification on resource-constrained devices. In *Advances in Neural Information Processing Systems*, volume 31, pages 10976–10987. Curran Associates, Inc., 2018.

Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In *International Conference on Machine Learning*, pages 1019–1028, 2017.

Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. Essentially no barriers in neural network energy landscape. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1309–1318. PMLR, 2018.

Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

Jiashi Feng, Huan Xu, and Shie Mannor. Outlier robust online learning. *CoRR*, abs/1701.00251, 2017.

Tharindu Fernando, Simon Denman, Sridha Sridharan, and Clinton Fookes. Going deeper: Autonomous steering with neural memory networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 214–221, 2017.

Shai Fine and Katya Scheinberg. Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264, 2002.

Yoav Freund, Yishay Mansour, and Robert E. Schapire. Why averaging classifiers can protect against overfitting. In *Proceedings of the 8th International Workshop on Artificial Intelligence and Statistics*, 2001.

Kenji Fukumizu and Shun-ichi Amari. Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural networks*, 13(3):317–327, 2000.

Moshe Gabel, Daniel Keren, and Assaf Schuster. Communication-efficient distributed variance monitoring and outlier detection for multivariate time series. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 37–47. IEEE, 2014.

Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. Prediction-based geometric monitoring over distributed data streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 265–276. ACM, 2012.

Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. Bayes and tukey meet at the center point. In *Learning Theory*, pages 549–563. Springer, 2004.

Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Inc., 1995.

Steve Hanneke. The optimal sample complexity of PAC learning. *Journal of Machine Learning Research*, 17(38):1319–1333, 2016.

Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. Distributed deep learning on edge-devices: Feasibility via adaptive compression. In *16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Ralf Herbrich, Thore Graepel, and Colin Campbell. Bayes point machines. *Journal of Machine Learning Research*, 1(Aug):245–279, 2001.

Mark Herbster and Manfred K Warmuth. Tracking the best linear predictor. *Journal of Machine Learning Research*, 1:281–309, 2001.

Daniel Hsu and Sivan Sabato. Loss minimization and parameter estimation with heavy tails. *Journal of Machine Learning Research*, 17(18):1–40, 2016.

Zhanhong Jiang, Aditya Balu, Chinmay Hegde, and Soumik Sarkar. Collaborative deep learning in fixed topology networks. In *Advances in Neural Information Processing Systems*, pages 5904–5914, 2017.

William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

Pooria Joulani, Andras Gyorgy, and Csaba Szepesvári. Online learning under delayed feedback. In *International Conference on Machine Learning*, pages 1453–1461, 2013.

Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Differentially private multi-party computation. In *2016 Annual Conference on Information Science and Systems (CISS)*, pages 128–132, 2016.

Michael Kamp, Mario Boley, and Thomas Gärtner. Beating human analysts in nowcasting corporate earnings by using publicly available stock price and correlation features. In *2013 IEEE 13th International Conference on Data Mining Workshops*, pages 384–390. IEEE, 2013.

David Kay and Eugene W Womble. Axiomatic convexity theory and relationships between the Carathéodory, Helly, and Radon numbers. *Pacific Journal of Mathematics*, 38(2):471–485, 1971.

Michael Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.

Michael Kearns and Yuriy Nevmyvaka. Machine learning for market microstructure and high frequency trading. *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.

David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD*, pages 289–300, 2006.

Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. Shape sensitive geometric monitoring. *Transactions on Knowledge and Data Engineering*, 24(8):1520–1535, 2012.

Daniel Keren, Guy Sagy, Amir Abboud, David Ben-David, Assaf Schuster, Izchak Sharfman, and Antonios Deligiannakis. Geometric monitoring of heterogeneous streams. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1890–1903, 2014.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2014.

Jyrki Kivinen, Alexander J. Smola, and Robert C. Williamson. Online learning with kernels. *Transactions on Signal Processing*, 52(8):2165–2176, 2004.

Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

Aleksandar Lazarevic and Zoran Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed and Parallel Databases*, 11(2):203–229, 2002.

Arnon Lazerson, Izchak Sharfman, Daniel Keren, Assaf Schuster, Minos Garofalakis, and Vasilis Samoladas. Monitoring distributed streams using convex decompositions. *Proceedings of the VLDB Endowment*, 8(5):545–556, 2015.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, 2013.

Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–670. ACM, 2014.

M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

Shao-Bo Lin, Xin Guo, and Ding-Xuan Zhou. Distributed learning with regularized least squares. *Journal of Machine Learning Research*, 18(92):1–31, 2017.

Yehida Lindell. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*, pages 1005–1009. IGI Global, 2005.

Qiang Liu and Alexander T Ihler. Distributed estimation, information loss and exponential families. In *Advances in Neural Information Processing Systems*, pages 1098–1106, 2014.

Phil Long and Rocco Servedio. Algorithms and hardness results for parallel large margin learning. In *Advances in Neural Information Processing Systems*, pages 1314–1322, 2011.

Philip M. Long and Rocco A. Servedio. Algorithms and hardness results for parallel large margin learning. *Journal of Machine Learning Research*, 14:3105–3128, 2013.

Chenxin Ma, Jakub Konečný, Martin Jaggi, Virginia Smith, Michael I. Jordan, Peter Richtárik, and Martin Takáč. Distributed optimization with arbitrary local solvers. *Optimization Methods and Software*, 32(4):813–848, 2017.

Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems*, pages 1231–1239, 2009.

Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 456–464. Association for Computational Linguistics, 2010.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. In *International Conference on Learning Representations*, 2018.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

Stanislav Minsker et al. Geometric median and robust estimation in banach spaces. *Bernoulli*, 21(4):2308–2335, 2015.

Nitinder Mohan and Jussi Kangasharju. Edge-fog cloud: A distributed cloud for internet of things computations. In *Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2016.

Cleve Moler. Matrix computation on distributed memory multiprocessors. *Hypercube Multiprocessors*, 86(181-195):31, 1986.

Cleve Moler. Another look at amdahl's law. Technical report, Technical Report TN-02-0587-0288, Intel Scientific Computers, 1987.

Mahesh Chandra Mukkamala and Matthias Hein. Variants of rmsprop and adagrad with logarithmic regret bounds. In *International Conference on Machine Learning*, pages 2545–2553, 2017.

S Muthukrishnan. Ad exchanges: Research issues. In *International Workshop on Internet and Network Economics*, pages 1–12. Springer, 2009.

Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87. Kluwer Academic Publisher, 2003.

Quynh Nguyen and Matthias Hein. The loss surface of deep and wide neural networks. In *International Conference on Machine Learning*, pages 2603–2612. PMLR, 2017.

Frank Nielsen and Richard Nock. Sided and symmetrized Bregman centroids. *IEEE Transactions on Information Theory*, 55(6):2882–2904, 2009.

Ilia Nouretdinov, Sergi G. Costafreda, Alexander Gammerman, Alexey Chervonenkis, Vladimir Vovk, Vladimir Vapnik, and Cynthia H.Y. Fu. Machine learning classification with confidence: application of transductive conformal predictors to MRI-based diagnostic and prognostic markers in depression. *Neuroimage*, 56(2):809–813, 2011.

Dino Oglic and Thomas Gärtner. Nyström method with kernel k-means++ samples as landmarks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2652–2660. PMLR, 2017a.

Dino Oglic and Thomas Gärtner. Nyström method with kernel k-means++ samples as landmarks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2652–2660, 06–11 Aug 2017b.

Francesco Orabona, Joseph Keshet, and Barbara Caputo. Bounded kernel-based online learning. *Journal of Machine Learning Research*, 10:2643–2666, 2009.

Francesco Orabona, Koby Crammer, and Nicolo Cesa-Bianchi. A generalized online mirror descent with applications to classification and regression. *Machine Learning*, 99(3):411–435, 2015.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, pages 305–313, 1989.

J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

Johann Radon. Mengen konvexer Körper, die einen gemeinsamen Punkt enthalten. *Mathematische Annalen*, 83(1):113–115, 1921.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.

Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

Jonathan D Rosenblatt and Boaz Nadler. On the optimality of averaging in distributed statistical learning. *Information and Inference*, 5(4):379–404, 2016.

Alexander M Rubinov. *Abstract convexity and global optimization*, volume 44. Springer Science & Business Media, 2013.

Sujay Sanghavi, Rachel Ward, and Chris D White. The local convexity of solving systems of quadratic equations. *Results in Mathematics*, 71(3-4):569–608, 2017.

Bernhard Scholkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond.* MIT press, 2001.

Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. A generalized representer theorem. In *Computational learning theory*, pages 416–426. Springer, 2001.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1):3–30, 2011.

Ohad Shamir. Without-replacement sampling for stochastic gradient methods. In *Advances in Neural Information Processing Systems*, pages 46–54, 2016.

Ohad Shamir and Nathan Srebro. Distributed stochastic optimization and learning. In *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*, pages 850–857. IEEE, 2014.

Ohad Shamir, Nati Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate newton-type method. In *International conference on machine learning*, pages 1000–1008, 2014.

Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *Transactions on Database Systems*, 32(4), 2007.

Izchak Sharfman, Assaf Schuster, and Daniel Keren. Shape sensitive geometric monitoring. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 301–310. ACM, 2008.

Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated multi-task learning. In *Advances in Neural Information Processing Systems*, pages 4424–4434, 2017.

Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Symposium on Security and Privacy*, pages 305–316, 2010.

Evan R Sparks, Ameet Talwalkar, Valton Smith, Jey Kottalam, Xinghao Pan, Jose Gonzalez, Michael J Franklin, Michael Jordan, Tim Kraska, et al. MLI: An API for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1187–1192. IEEE, 2013.

Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *Information Theory Workshop (ITW)*, pages 1–5. IEEE, 2015.

John W Tukey. Mathematics and the picturing of data. In *Proceedings of the International Congress of Mathematicians*, volume 2, pages 523–531, 1975.

Helge Tverberg. A generalization of Radon's theorem II. *Bulletin of the Australian Mathematical Society*, 24(03):321–325, 1981.

Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

Vladimir Naumovich Vapnik and Alexey Yakovlevich Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

Jeffrey Scott Vitter and Jyh-Han Lin. Learning in parallel. *Information and Computation*, 96 (2):179–202, 1992.

Ulrike Von Luxburg and Bernhard Schölkopf. Statistical learning theory: Models, concepts, and results. In *Handbook for the History of Logic*, volume 10, pages 751–706. Cambridge University Press, 2009.

Grace Wahba. *Spline models for observational data*, volume 59. Siam, 1990.

Weiran Wang and Nathan Srebro. Stochastic nonconvex optimization with large minibatches. *CoRR*, abs/1709.08728, 2017.

Zhuang Wang and Slobodan Vucetic. Online passive-aggressive algorithms on a budget. In *International Conference on Artificial Intelligence and Statistics*, pages 908–915, 2010.

Lodewyk FA Wessels and Etienne Barnard. Avoiding false local minima by proper initialization of connections. *IEEE Transactions on Neural Networks*, 3(6):899–905, 1992.

Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

Yoshua Bengio Xavier Glorot. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2010.

Lin Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11:2543–2596, 2010.

Feng Yan, Shreyas Sundaram, SVN Vishwanathan, and Yuan Qi. Distributed autonomous online learning: Regrets and intrinsic privacy-preserving properties. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2483–2493, 2013.

Shuai Yuan, Jun Wang, and Xiaoxue Zhao. Real-time bidding for online advertising: measurement and analysis. In *Proceedings of the Seventh International Workshop on Data Mining for Online Advertising*, page 3. ACM, 2013.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over Hadoop data with Spark. *USENIX; login*, 37(4):45–51, 2012.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *Proceedings of the International Conference on Learning Representations*, 2017.

Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the international conference on machine learning (ICML)*, page 116. ACM, 2004.

Yuchen Zhang, Martin J Wainwright, and John C Duchi. Communication-efficient algorithms for statistical optimization. In *Advances in Neural Information Processing Systems*, pages 1502–1510, 2012.

Yuchen Zhang, John C. Duchi, and Martin J. Wainwright. Communication-efficient algorithms for statistical optimization. *Journal of Machine Learning Research*, 14(1):3321–3363, 2013.

Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.

Notation

$\mathcal{X}$	–	The input space, i.e., the space of instances of a machine learning problem.
x	–	An instance from the input space $\mathcal{X}$.
$\mathcal{Y}$	–	The output space, i.e., the space of labels assigned to each instance.
y	–	A label from the output space $\mathcal{Y}$.
$\mathcal{D}$	–	A target distribution over the input and output space.
E	–	A training dataset $E = \{(x_1, y_1), \dots, (x_N, y_N)\} \subseteq \mathcal{X} \times \mathcal{Y}$
$\mathcal{F}$	–	A model space.
f	–	A model from the model space $\mathcal{F}$.
w	–	A parameterization of a model f_w from the Euclidean space.
$\mathbf{f}$	–	A set of models from the model space, i.e., $\mathbf{f} \subset \mathcal{F}$.
$\overline{\mathbf{f}}$	–	The average of a set of models.
f^*	–	The optimal model from $\mathcal{F}$ for a given learning problem.
l	–	A general loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$
ℓ	–	A convex loss function.
$\mathcal{L}$	–	A risk functional.
$\mathcal{R}$	–	The regret of a model $f \in \mathcal{F}$.
$\mathcal{H}$	–	A reproducing kernel Hilbert space.
k	–	A kernel function.
S	–	The set of support vectors of the dual representation of a model from $\mathcal{H}$.
α	–	The coefficients corresponding to the support vectors S of a model from $\mathcal{H}$.
P	–	A probability distribution.
$\mathcal{A}$	–	A machine learning algorithm.
m	–	The number of learners.
i	–	One of the m learners.
N	–	The overall size of the training data set E.
$N_{\mathcal{F}}$	–	The sample complexity for the model space $\mathcal{F}$.
n	–	The size of the training dataset available to each parallel instance of $\mathcal{A}$.
$\mathcal{T}$	–	The runtime of an algorithm, depending on the input size.
c	–	The number of processing units available for parallel computation of $\mathcal{A}$.
ϵ	–	An error bound in a probabilistic guarantee on hypothesis quality.
Δ	–	The confidence for the error bound to hold in a probabilistic guarantee.
δ	–	The confidence achieved by each parallel instance of $\mathcal{A}$.
$\mathfrak{r}$	–	A Radon point.
$\mathfrak{r}_h$	–	A Radon point obtained by the iterated Radon point algorithm with h iterations.
T	–	The total number of rounds processed by an online learning algorithm.
t	–	A specific round processed by an online learning algorithm.
$[n]$	–	For a natural number $n \in \mathbb{N}$, $[n]$ abbreviates the set $\{1, \dots, n\}$.

List of Figures

List of Tables

A. Appendix

A. Implementations of the Protocols

This section presents two implementations of the protocols presented in this thesis.

A.1. Dynamic Averaging: An Open-Source Learning Framework in STORM

The dynamic averaging approach developed in this thesis has been implemented as part of an distributed online learning framework based on Apache STORM. The learning framework was developed in the EU project FERARI as part of an open-source big-data in-stream complex event processing (CEP) architecture. The architecture can be obtained under Apache License 2.0 at `https://bitbucket.org/sbothe-iais/ferari`.

The architecture allows to flexibly set up a communication-efficient, distributed complex event processing architecture, including distributed online learners. These learners generate predictions based on complex events detected in the system which can either be send to an external site or used as further input for the event processing engine. Given a configuration of the CEP—including online learning—an optimizer component derives the optimal distribution of components over physically distributed sites, starts the respective components and re-optimizes its plan according to incoming statistics from the sites. These statistics are estimated using the variant of distributed kernel density estimation proposed in this thesis.

In addition to the implementation within the FERARI architecture, a version suitable for execution on a single cluster or cloud system as a single STORM topology has been implemented. This version is derived from the original FERARI version but stripped from the components required for complex event processing, query optimization and multiple STORM topology support. It is available as open source under Apache License 2.0 at `https://bitbucket.org/Michael_Kamp/distributedonlinelearning`.

The framework implements the dynamic averaging approach, periodic averaging, and mini-batching, for linear and kernel models. It supports classification, regression, outlier detection and kernel density estimation.

A.2. Radon Machine: An Open Source Implementation in SPARK

The parallel Radon machine is implemented both in python and Scala. Both implementations build on the distributed computation framework Apache Spark. , while the Scala variant natively integrates with the Spark framework. The implementation is open source under the Apache License 2.0 at `https://bitbucket.org/Michael_Kamp/radonmachine`.

A Python Implementation

The Radon machine has been implemented as a python class that parallelizes machine learning algorithms using a wrapper to allow integrating novel algorithms. It uses the PySpark interfaces to interact with the Spark framework. For a quick start with the parallel Radon machine, wrappers for machine learning algorithms from the popular scikit-learn machine learning library (Pedregosa et al., 2011) are provided. Example experiments allow to quickly get familiar with the implementation, as well as offer reproducibility of empirical results.

Python allows for rapid prototyping and easy implementation of experiments and new learning algorithms. A major drawback of the python implementation is that it is slow. This stems on the one hand from the innate low performance of python itself due to the fact that it is an interpreted language. On the other hand, data has to be transfered from the Spark framework, which is implemented in Java, to the python learners. Spark transfers this data from its Java classes to the PySpark interfaces using sockets. That is, the data is first serialized, then sent over sockets, and finally de-serialized. This process is very time consuming and seems to be the main reason for the low performance of the python implementation. In order to overcome this drawback, I implemented the Radon machine in Scala.

A Scala Implementation

The Scala implementation of the Radon machine is similar to the python implementation in that it is applicable to all machine learning algorithms that implement a Learner trait (in Scala, a trait is similar to an interface). The implementation provides wrappers for a large set of machine learning algorithms from the Apache SPARK machine learning library (Meng et al., 2016), as well as algorithms from the WEKA library (Witten et al., 2016).

B. Additional Theoretical Results

B.1. Properties of Convex Projections

Proposition A.1. *Let $\Pi(\cdot)$ be a projection on a non-empty closed convex set and $v,w\in\mathcal{F}$ for a Hilbert space $\mathcal{F}$. Then it holds that*

$$\|\Pi(v)-\Pi(w)\|^2\leq\|v-w\|^2-\|v-\Pi(v)-w+\Pi(w)\|^2$$

Proof. We have that

$$\begin{aligned}\|v-w\|^2=&\|\Pi(v)-\Pi(w)+v-\Pi(v)-w+\Pi(w)\|^2\\=&\|\Pi(v)-\Pi(w)\|^2+\|v-\Pi(v)-w+\Pi(w)\|^2\\&+2\langle\Pi(v)-\Pi(w),v-\Pi(v)-w+\Pi(w)\rangle\quad.\end{aligned}$$

We can rewrite the inner product as

$$\langle\Pi(v)-\Pi(w),v-\Pi(v)-w+\Pi(w)\rangle=\langle\Pi(v)-\Pi(w),v-\Pi(v)\rangle+\langle\Pi(w)-\Pi(v),w-\Pi(w)\rangle\quad.$$

From lemma 3.1.4 in Nesterov (2003) it follows that

$$\langle \Pi(v) - \Pi(w)\,, v - \Pi(v)\rangle \geq 0 \quad \text{and} \quad \langle \Pi(w) - \Pi(v)\,, w - \Pi(w)\rangle \geq 0 \ .$$

Thus we have that

$$\|v - w\|^2 \geq \|\Pi(v) - \Pi(w)\,\|^2 + \|v - \Pi(v) - w + \Pi(w)\,\|^2 \ .$$

□

Proposition A.2. *Let $\mathcal{F}$ be a convex Hilbert space, $\Pi(\cdot)$ be a projection on a non-empty closed convex set Γ and $d, s, d', s' \in \mathcal{F}$. Then it holds that*

$$\|d' - s'\|^2 - \|d' - \Pi(d) - s' + \Pi(s)\,\|^2 \leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\,\|^2 \ ,$$

Proof. The proof is technical and relies on the fact that the projections on the left side are not on d' and s' but on d and s. Using the triangle inequality yields

$$\begin{aligned}
&\|d' - s'\|^2 - \|d' - \Pi(d) - s' + \Pi(s)\,\|^2 \leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\,\|^2 \\
\Leftrightarrow\ &\|d' - s'\|^2 - \|d - s\|^2 \leq \|d' - \Pi(d) - s' + \Pi(s)\,\|^2 - \|d - \Pi(d) - s + \Pi(s)\,\|^2 \\
&= (\underbrace{\|d' - s' + \Pi(s) - \Pi(d)\,\|}_{\leq \|d'-s'\| + \|\Pi(s)-\Pi(d)\|})^2 - (\underbrace{\|d - s + \Pi(s) - \Pi(d)\,\|}_{\geq |\|d-s\| - \|\Pi(s)-\Pi(d)\||})^2 \\
&\leq (\|d' - s'\|^2 + \|\Pi(s) - \Pi(d)\,\|^2 + 2\|d' - s'\|\|\Pi(s) - \Pi(d)\,\|) \\
&\quad - (\|d - s\|^2 + \|\Pi(s) - \Pi(d)\,\|^2 - 2\|d - s\|\|\Pi(s) - \Pi(d)\,\|) \\
&= \|d' - s'\|^2 - \|d - s\|^2 \\
&\quad + 2\|d' - s'\|\|\Pi(s) - \Pi(d)\,\| + 2\|d - s\|\|\Pi(s) - \Pi(d)\,\| \\
\Leftrightarrow\ 0 &\leq 2\|d' - s'\|\|\Pi(s) - \Pi(d)\,\| + 2\|d - s\|\|\Pi(s) - \Pi(d)\,\| \\
&= 2\|\Pi(s) - \Pi(d)\,\|\left(\|d' - s'\| + \|d - s\|\right)
\end{aligned}$$

By definition, $\|\cdot\| \geq 0$ ands the above inequality holds. Thus it holds that

$$\|d' - s'\|^2 - \|d' - \Pi(d) - s' + \Pi(s)\,\|^2 \leq \|d - s\|^2 - \|d - \Pi(d) - s + \Pi(s)\,\|^2 \ ,$$

□

B.2. Proof of Lemma 4.1

This section proofs the extension of the update lemma to mini-batches of data. For convenience, the lemma is re-stated:

Lemma A.3 (re-stating Lemma 4.1). *Let the updates of an incremental learning algorithm $\mathcal{A}$ be regret-proportional convex updates with $\gamma > 0$. Then for all models $d, s \in \mathcal{F}$ and all datasets $E \subset \mathcal{X} \times \mathcal{Y}$ it holds that*

$$\|\mathcal{A}(E, d) - \mathcal{A}(E, s)\|^2 \leq \|d - s\|^2 - \gamma^2 \sum_{(x,y)\in E} (\ell(d, x, y) - \ell(s, x, y))^2 \,.$$

Proof. From Equation 3.18 from Lemma 3.14 in Chapter 3 it is known that

$$\|d' - s'\|^2 \leq \|d - s\|^2 - \tau_E \left(\|d - \Pi(d)\| - \|s - \Pi(s)\|\right)^2 \quad ,$$

i.e., that the distance between models is reduced by an update proportional to the update magnitude. It remains to relate the update magnitude to the loss over the dataset E. For that, observe that it follows from condition (i) of the definition of regret-proportionality that

$$\|f - \Pi(f)\| = \frac{1}{\tau_E}\|f - (f + \tau_E(\Pi(f) - f))\| = \frac{\|f - f'\|}{\tau_E} \geq \frac{\gamma}{\tau_E} \sum_{(x,y)\in E} \ell(f, x, y) \quad .$$

Since $\tau_E \in (0, 1]$ this yields

$$\|f - \Pi(f)\| \geq \gamma \sum_{(x,y)\in E} \ell(f, x, y) \quad .$$

Finally, inserting this into Equation 3.18 yields the claim

$$\|d' - s'\|^2 \leq \|d - s\|^2 - \gamma^2 \sum_{(x,y)\in E} \left(\ell(d, x, y) - \ell(s, x, y)\right)^2 \quad .$$

□

B.3. Consistency Results for Empirical Risk Minimisation

In this section we provide some technical results on the consistency of empirical risk minimisation algorithms.

Lemma A.4. *For consistent empirical risk minimisers with a model space of finite Vapnik-Chervonenkis (VC) dimension the sample size required to achieve an* (ϵ, Δ)*-guarantee is given by* $N(\Delta) = (\alpha_\epsilon + \beta_\epsilon \log_2 1/\Delta)^k$ *with* $\alpha_\epsilon = 4\ln 2 1/\epsilon^2, \beta_\epsilon = 4/\epsilon^2 \log_2 e$ *and* $k = 2$.

Proof. For consistent empirical risk minimisers with finite VC-dimension, the confidence $1 - \Delta$ for a given N and ϵ is $\Delta = 2\mathcal{N}(\mathcal{F}, N)\exp(-N\epsilon^2/4)$ (Von Luxburg and Schölkopf, 2009), where the shattering coefficient $\mathcal{N}(\mathcal{F}, N)$ is a polynomial in N for finite VC-dimension. Solving for N yields that the algorithm run with

$$N \geq \frac{1}{\epsilon^2}\left(\ln 2 + 4\frac{1}{\log_2(e)} \log_2 \frac{1}{\Delta}\right)$$

achieves a confidence larger or equal to the desired $1 - \Delta$. □

Lemma A.5. *For consistent empirical risk minimisers with a model space of finite Rademacher complexity the sample size required to achieve an* (ϵ, Δ)*-guarantee is given by* $N(\Delta) = (\alpha_\epsilon + \beta_\epsilon \log_2 1/\Delta)^k$ *with* $\alpha_\epsilon = 0, \beta_\epsilon = 1/2(\epsilon + 2\rho)^2$ *and* $k = 1$*, where* ρ *denotes the Rademacher complexity.*

Proof. For consistent empirical risk minimisers with a model space of finite Rademacher complexity ρ, a given Δ and N the error bound is given by $\epsilon = 2\rho + \sqrt{\log_2 1/\delta / 2N}$ (Von Luxburg and Schölkopf, 2009). Solving for N yields the above result. □

B.4. A Result on Strong Convexity

In this section we present a result that extends the definition of μ-strong convexity to an arbitrary number of points. For that, we first recapitulate the definition of strong convexity.

Definition A.6. *A function $\mathcal{R}\colon \mathbb{R}^d \to \mathbb{R}$ is μ-strongly convex with $\mu \in \mathbb{R}$ if for all $\tau \in [0,1]$ and all $f, g \in \mathbb{R}^d$ it holds that*

$$\mathcal{R}(\tau f + (1-\tau)g) \leq \tau\mathcal{R}(f) + (1-\tau)\mathcal{R}(g) - \frac{\mu}{2}\tau(1-\tau)\|f-g\|_2^2 \ .$$

We extend this result to a set of $m \in \mathbb{N}$ points $f_1, \ldots, f_m$ and their weighted average $\overline{f}_w$.

Proposition A.7. *For a μ-strongly convex function $\mathcal{R}\colon \mathbb{R}^d \to \mathbb{R}$, $m \in \mathbb{N}$ points $f_1, \ldots, f_m \in \mathbb{R}^d$ and arbitrary weights $w_1, \ldots, w_m \in \mathbb{R}_+$ with $\sum_{i\in[m]} w_i = 1$ it holds that*

$$\mathcal{R}\left(\overline{f}_w\right) \leq \sum_{i\in[m]} w_i\mathcal{R}(f_i) - \frac{\mu}{2}\max_{j\in[m]}\frac{w_j}{1-w_j}\left\|\overline{f}_w - f_j\right\|_2^2 \ ,$$

with $\overline{f}_w = \sum_{i\in[m]} w_i f_i$.

Proof. Given a set of $m \in \mathbb{N}$ points $f_1, \ldots, f_m$, it holds for all permutations of the set that

$$\mathcal{R}\left(\overline{f}\right) = \mathcal{R}\left(\sum_{i=1}^{m} w_i f_i\right) = \mathcal{R}\left(\sum_{i=1}^{m-1} w_i f_i + w_m f_m\right) = \mathcal{R}\left((1-w_m)\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i + w_m f_m\right) \ .$$

From the μ-strong convexity follows that

$$\begin{aligned}\mathcal{R}\left(\overline{f}_w\right) \leq& (1-w_m)\mathcal{R}\left(\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i\right) + w_m\mathcal{R}(f_m)\\ &- \frac{\mu}{2}w_m(1-w_m)\left\|\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i - f_m\right\|_2^2\\ =& (1-w_m)\mathcal{R}\left(\frac{\sum_{i=1}^{m-1} w_i f_i}{\sum_{i=1}^{m-1} w_i}\right) + w_m\mathcal{R}(f_m)\\ &- \frac{\mu}{2}w_m(1-w_m)\left\|\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i - f_m\right\|_2^2 \ .\end{aligned} \tag{A.8}$$

Applying Jensen's inequality, we can bound

$$\mathcal{R}\left(\frac{\sum_{i=1}^{m-1} w_i f_i}{\sum_{i=1}^{m-1} w_i}\right) \leq \frac{\sum_{i=1}^{m-1} w_i\mathcal{R}(f_i)}{\sum_{i=1}^{m-1} w_i} = \frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i\mathcal{R}(f_i)$$

which yields together with Equation A.8

$$\begin{aligned}
\mathcal{R}\left(\overline{f}_w\right) \leq& (1-w_m)\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i\mathcal{R}\left(f_i\right) + w_m\mathcal{R}(f_m)\\
&-\frac{\mu}{2}w_m(1-w_m)\left\|\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i - f_m\right\|_2^2\\
=&\sum_{i=1}^{m} w_i\mathcal{R}\left(f_i\right) - \frac{\mu}{2}w_m(1-w_m)\left\|\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i - f_m\right\|_2^2\\
=&\sum_{i=1}^{m} w_i\mathcal{R}\left(f_i\right) - \frac{\mu}{2}w_m(1-w_m)\left\|\frac{1}{1-w_m}\sum_{i=1}^{m-1} w_i f_i - \frac{1-w_m}{1-w_m}f_m\right\|_2^2\\
=&\sum_{i=1}^{m} w_i\mathcal{R}\left(f_i\right) - \frac{\mu}{2}w_m(1-w_m)\frac{1}{(1-w_m)^2}\left\|\sum_{i=1}^{m-1} w_i f_i - (1-w_m)f_m\right\|_2^2\\
=&\sum_{i=1}^{m} w_i\mathcal{R}\left(f_i\right) - \frac{\mu}{2}\frac{w_m}{(1-w_m)}\left\|\sum_{i=1}^{m-1} w_i f_i + w_m f_m - f_m\right\|_2^2\\
=&\sum_{i=1}^{m} w_i\mathcal{R}\left(f_i\right) - \frac{\mu}{2}\frac{w_m}{(1-w_m)}\left\|\sum_{i=1}^{m} w_i f_i - f_m\right\|_2^2 \quad .
\end{aligned}$$

Since this result holds for all permutations of $f_1,\dots,f_m$ it holds in particular for

$$j = \arg\max_{j\in[m]}\frac{w_j}{(1-w_j)}\left\|\sum_{i=1}^{m} w_i f_i - f_j\right\|_2^2$$

From the fact that $\overline{f}_w = \sum_{i=1}^{m} w_i f_i$ follows the result. □

For $m = 2$, this result is equal to the definition of μ-strong convexity. From this directly follows a result on the standard average $\overline{f}$.

Lemma A.9. *For a μ-strongly convex function $\mathcal{R}\colon\mathbb{R}^d\to\mathbb{R}$ and $m\in\mathbb{N}$ points $f_1,\dots,f_m\in\mathbb{R}^d$ it holds that*

$$\mathcal{R}(\overline{f}) \leq \frac{1}{m}\sum_{i=1}^{m} f_i - \frac{\mu}{2}\max_{j\in[m]}\frac{1}{m-1}\left\|\overline{f} - f_j\right\|_2^2 \quad ,$$

with $\overline{f} = {}^1\!/\!_m\sum_{i\in[m]} f_i$.

✉ *info@michaelkamp.org*
michaelkamp.org
**04 September, 1984*

Publications

Journals and Conferences

[1] Michael Kamp, Linara Adilova, Joachim Sicking, Fabian Hüger, Peter Schlicht, Tim Wirtz, Stefan Wrobel. Efficient Decentralized Deep Learning by Dynamic Model Averaging. *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, 2018.

[2] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Kamp, and Michael Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217-236, 2017.

[3] Michael Kamp, Mario Boley, Olana Missura, and Thomas Gärtner. Effective parallelisation for machine learning. In *Advances in Neural Information Processing Systems - NIPS*, pages 6459-6470, 2017.

[4] Katrin Ullrich, Michael Kamp, Thomas Gärtner, Martin Vogt, and Stefan Wrobel. Co-regularised support vector regression. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, 2017.

[5] Michael Kamp, Sebastian Bothe, Mario Boley, and Michael Mock. Communication-efficient distributed online learning with kernels. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, pages 805-819, 2016.

[6] Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-efficient distributed online prediction by dynamic model synchronization. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, pages 623-639, 2014.

[7] Michael Kamp, Mario Boley, and Thomas Gärtner. Beating human analysts in nowcasting corporate earnings by using publicly available stock price and correlation features. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 641-649, 2014.

[8] Michael Kamp, Christine Kopp, Michael Mock, Mario Boley, and Michael May. Privacy-preserving mobility monitoring using sketches of stationary sensor readings. In *Machine Learning and Knowledge Discovery in Databases - ECML PKDD*, pages 370-386, 2013.

Workshops

[9] Sven Giesselbach, Katrin Ullrich, Michael Kamp, Daniel Paurat, Thomas Gärtner. Designing Visualisation Enhancements for SIEM Systems. In *NeurIPS Workshop on Machine Learning for Health (ML4H)*, 2018.

[10] Phong H Nguyen, Siming Chen, Natalia Andrienko, Michael Kamp, Linara Adilova, Gennady Andrienko, Olivier Thonnard, Alysson Bessani, Cagatay Turkay. Designing Visualisation Enhancements for SIEM Systems. In *15th IEEE Symposium on Visualization for Cyber Security âĂŞ VizSec*, 2018.

[11] Katrin Ullrich, Michael Kamp, Thomas Gärtner, Martin Vogt, and Stefan Wrobel. Ligand-Based Virtual Screening with Co-regularised Support Vector Regression. In *16th IEEE International Conference on Data Mining (ICDM) Workshops*, 2016.

[12] Michael Kamp, Mario Boley, and Thomas Gärtner. Parallelizing randomized convex optimization. In *8th NIPS Workshop on Optimization for Machine Learning (OPT)*, 2015.

[13] Michael Kamp, Mario Boley, Michael Mock, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Adaptive communication bounds for distributed online learning. In *7th NIPS Workshop on Optimization for Machine Learning (OPT)*, 2014.

[14] Mario Boley, Michael Kamp, Daniel Keren, Assaf Schuster, and Izchak Sharfman. Communication-Efficient Distributed Online Prediction using Dynamic Model Synchronizations. In *First Internation Workshop on Big Dynamic Distributed Data (BD3) at VLDB*, 2013.

[15] Michael Kamp, Mario Boley, and Thomas Gärtner. Beating Human Analysts in Nowcasting Corporate Earnings by Using Publicly Available Stock Price and Correlation Features. In *13th IEEE International Conference on Data Mining (ICDM) Workshops*, 2013.

[16] Michael Kamp and Andrei Manea. Stones: Stochastic technique for generating songs. In *NIPS Workshop on Constructive Machine Learning (CML)*, 2013.