
Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Theoretische Informatik

# Federated Learning

Florian Hartmann
Matrikelnummer: 4775495
florian.hartmann@fu-berlin.de

Betreuer: Prof. Dr. Wolfgang Mulzer
Zweitkorrektor: Prof. Dr. Dr. (h.c.) habil. Raúl Rojas

Berlin, 20.8.2018

**Abstract**

Over the past few years, machine learning has revolutionized fields such as computer vision, natural language processing, and speech recognition. Much of this success is based on collecting vast amounts of data, often in privacy-invasive ways. Federated Learning is a new subfield of machine learning that allows training models without collecting the data itself. Instead of sharing data, users collaboratively train a model by only sending weight updates to a server. While this better respects privacy and is more flexible in some situations, it does come at a cost. Naively implementing the concept scales poorly when applied to models with millions of parameters. To make Federated Learning feasible, this thesis proposes changes to the optimization process and explains how dedicated compression methods can be employed. With the use of Differential Privacy techniques, it can be ensured that sending weight updates does not leak significant information about individuals. Furthermore, strategies for additionally personalizing models locally are proposed. To empirically evaluate Federated Learning, a large-scale system was implemented for Mozilla Firefox. 360,000 users helped to train and evaluate a model that aims to improve search results in the Firefox URL bar.

## Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 20. August 2018

(Florian Hartmann)

## Acknowledgements

Many people have supported me while writing this thesis. First of all, I would like to thank Wolfgang Mulzer for advising me on this thesis and for providing so much helpful feedback, even when I was not in Berlin.

A major part of this thesis was developed during my time at Mozilla in Mountain View. I am very thankful for the research-friendly environment that Mozilla provided and greatly appreciate how open Mozilla is in its work.

Sunah Suh made it possible to launch my code to Firefox Beta and helped to resolve many problems along the way. For this support and for always having an open ear to my problems, I would like to wholeheartedly thank her. I also want to express my gratitude to Arkadiusz Komarzewski for being so interested in the project and for porting my Python server-side implementation to Scala.

Furthermore, I want to extend my thanks to Drew Willcoxon and Rob Helmer, who helped me navigate the Firefox specific parts of the project. I also want to thank Katie Parlante for making it possible to work on my thesis at Mozilla and for being so incredibly supportive of my ideas.

I would like to thank Rishi Bommasani for reading several drafts of the thesis, for all his feedback and for the nice time we shared in California. I also want to thank Anthony Miyaguchi for all the interesting conversations we had in the Mountain View office, some of which revolved around this thesis.

Finally, I would like to thank my parents for always encouraging me in all my endeavors. I deeply appreciate all the support I received over the years.

# Contents

## Notation

- At a given time:
  - There are $k$ users in total
  - They have $n$ data points in total
  - $n_i$ data points are from the $i$-th user
- In each iteration:
  - $K$ users sampled
  - These users have $N$ data points
  - $H_i$ is the update proposed by the $i$-th user
- Statistics:
  - $\eta$ is the learning rate
  - $\theta$ are the weights of the model
  - $m$ is the number of weights
  - $f$ is the prediction function of a model
  - $L$ is the total loss of the model over a dataset
  - $\tilde{X}$ is an estimator of a value $X$
- Data:
  - $x_i$ and $y_i$ respectively are the $i$-th data point and label in total
  - $x_{ij}$ and $y_{ij}$ are the $j$-th data point and label of the $i$-th user
- $\nabla$ is the gradient operator, or, more generally, a function that computes a model improvement based on the loss

# 1 Introduction

## 1.1 Motivation

Many problems in computer science are tremendously difficult to solve by handcrafting algorithms. Speech recognition systems are a prominent example of this. The audio recordings that need to be analyzed consist of huge amounts of high-dimensional data. Understanding this data well by manual inspection is virtually impossible.

Machine learning provides an alternative approach to solving such problems. It is often referred to as the field of learning from example data [12, 30]. By collecting data and then applying statistical techniques, patterns can be found in an automated way. In the example of speech recognition, one would collect a large amount of audio recordings and their respective transcriptions. Machine learning algorithms can then find patterns in the audio data that help with interpreting the audio signals [40].

In the past few years, this idea has been successfully applied to many different areas [54]. Speech recognition and recommender systems are mostly based on machine learning nowadays [36, 40, 77]. The areas of computer vision and natural language processing also increasingly rely on data-driven approaches. For example, most state-of-the-art solutions in object detection and machine translation are based on learning from data [52, 6].

Many learning algorithms that are now extremely successful have been known for many years. For instance, the backpropagation algorithm, which most of deep learning is based on, was described as early as in 1970 [57, 37]. To explain why these ideas are only now being used successfully, three reasons are generally given. First, there have been some fundamental improvements to the algorithms that had a great effect. To give just a single example, Adam has greatly reduced the amount of tuning that is required to make gradient descent work well [49].

A second reason for the recent success has been the increase in computational resources that are available. Processing power has repeatedly doubled over the years [59] and special hardware for linear algebra and machine learning has been released [48]. However, the third reason, and often considered the most central one, is that more data to train on is available. Having more example data means that the algorithms get more information to decide what patterns are truly important. Consequently, it is less likely that example data points are just memorized or that random patterns are detected as a useful signal.

An impressive demonstration of the value of having more data was given by Facebook in May 2018 [93]. By training an object detection model using 3.5 billion Instagram images [21], they outperformed all other models on ImageNet, the standard benchmark for object recognition [24]. Even though the methods used to analyze the data were not completely new, the

huge amount of data helped them to build the best object detection model to date. The fact that having a lot of data is extremely important in building good machine learning models has been widely described and discussed [38].

The predominant way of using machine learning nowadays involves collecting all this data in a data center. The model is then trained on powerful servers. However, this data collection process is often privacy-invasive. Many users do not want to share private data with companies, making it difficult to use machine learning in some situations, e.g. when medical data is involved. Even when privacy is not a concern, having to collect data can be infeasible. For example, self-driving cars generate too much data to be able to send all of it to a server [101].

*Federated Learning* [63, 50] is an alternative approach to machine learning where data is not collected. In a nutshell, the parts of the algorithms that touch the data are moved to the users' computers. Users collaboratively help to train a model by using their locally available data to compute model improvements. Instead of sharing their data, users then send only these abstract improvements back to the server.

This approach is much more privacy-friendly and flexible. Applications on mobile phones provide examples where this is especially evident. Users generate vast amounts of data through interaction with the device. This data is often deeply private in nature and should not be shared completely with a server. Federated Learning still allows training a common model using all this data, without necessarily sacrificing computational power or missing out on smarter algorithms. Here, Federated Learning approaches can even lead to better models than conventional techniques since more data is available.

## 1.2 Structure of the Thesis

It is the goal of this thesis to give an introduction to Federated Learning and to provide an overview over solving various related problems. Since the field is still fairly new as of 2018, the aim is to provide a complete guide on developing a Federated Learning system. For aspects that were previously suggested as future research areas, such as personalizing models [64], we propose new approaches. Additionally, the goal is to demonstrate empirically that Federated Learning does not only work in simulations but can also work in complex software projects.

The remainder of this thesis is structured as follows: The field of Federated Learning is defined and formalized in Section 1.3. The fundamental properties and the protocol introduced in that section serve as a basis for all other chapters. The necessary background knowledge to understand this thesis is given in Section 2.

The next two sections focus on making Federated Learning more efficient. Section 3 deals with various aspects related to the optimization

process. Compression techniques for reducing the communication effort are introduced in Section 4.

In Section 5, privacy-specific aspects of Federated Learning are discussed. Using Differential Privacy [26], it can be ensured that it is virtually impossible to make conclusions about what data individuals used to help train the model. Allowing personalized models, while still collaboratively helping to train a central model, is the topic of Section 6.

An implementation of a Federated Learning system for Mozilla Firefox is discussed in Section 7. Finally, Section 8 gives an overview over all the topics covered and provides a short conclusion.

## 1.3 Federated Learning

In supervised learning, $n$ example inputs and the corresponding outputs are given:

$$\begin{aligned} \text{Example inputs} \quad & x_1, \ldots, x_n \\ \text{outputs} \quad & y_1, \ldots, y_n \end{aligned}$$

This example data is sampled from some underlying distribution that is not known to us. The goal is to find a function $f$ that maps from example inputs to outputs sampled from this distribution. To do this, we decide on the form of the function and then optimize the variable parts of it. For example, this could mean fixing the function to be linear and then finding the best possible coefficients. Because only some example data is available, the goal is to find a function that maps well from the example inputs to the outputs under the constraint that it also needs to work as well for new data sampled from the same distribution.

In other areas like unsupervised machine learning, the example outputs might be missing and the success criteria for $f$ are defined differently [33]. The variable parts of the function which are optimized during the learning process are called *parameters* or *weights*. Values that need to be set before the training begins are called *hyperparameters*. Generally, these are much more difficult to optimize [35].

In order to use conventional machine learning techniques, the $n$ data points are collected and the training process is performed on a server. This is in contrast to Federated Learning, where the explicit requirement is that users do not have to share their data. Instead, the $n$ data points are partitioned across the computers of $k$ users. Users can have varying numbers of data points, so $n_i$ denotes how many examples the $i$-th client has access to.

Most machine learning algorithms work in an iterative process where they repeatedly look at example data. They start off with an initial solution and then continually try to improve it. In each iteration, the model is evaluated using the training data and then updated. Each update is meant to improve the model's performance on the training data a little bit.

To avoid collecting the training data, a distributed way of running the learning algorithm is required. The parts of the algorithm that directly make use of the data need to be executed on the users' computers. These correspond to the sections of the algorithm that compute the previously mentioned updates. In Federated Learning, users compute updates based on their locally available training data and send them to a server. These updates are much harder to interpret than pure data, so this is a major improvement for privacy. For some applications with huge amounts of data, it might also be cheaper to communicate the updates compared to directly sending the data.

While the computers of users generally have much less computational power than servers in a data center, there is also less data on them. By having a lot of users, the computations that need to be performed are vastly distributed. There is not much work to do on the computer of each individual.

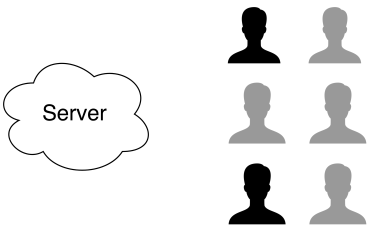
Conventional machine learning can be seen as a centralized system where all the work is performed on one server. In the process described so far, responsibilities are moved from the server to the clients. This is not a fully decentralized system because the server still runs a part of the algorithm. Instead, it is a *federated* system: A federation of clients takes over a significant amount of work but there is still one central entity, a server, coordinating everything.

Before the server starts off the distributed learning process, it needs to initialize the model. Theoretically, this can be done randomly. In practice, it makes sense so smartly initialize the model with sensible default values. If some data is already available on the server, it can be used to pretrain the model. In other cases, there might be a known configuration of model parameters that already leads to acceptable results. Having a good first model gives the training process a headstart and can reduce the time it takes until convergence.

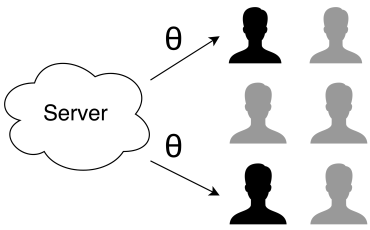
After the model has been initialized, the iterative training process is kicked off. A visualization of the steps performed in each iteration is shown in Figure 12. At the beginning of an iteration, a subset of $K$ clients are randomly selected by the server. They receive a copy of the current model parameters $\theta$ and use their locally available training data to compute an update. The update of the $i$-th client is denoted by $H_i$. The updates are then sent back to the server.

In this thesis, we generally assume $\theta$ and $H_i$ to be vectors in $\mathbb{R}^m$. However, the same concepts transfer directly to any sequence of vectors since they can be concatenated into one long vector.

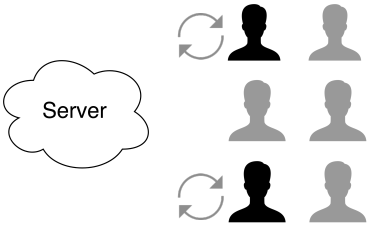
The server waits until it has received all updates and then combines them into one final update. This is usually done by computing an average of all updates, weighted by how many training examples the respective clients



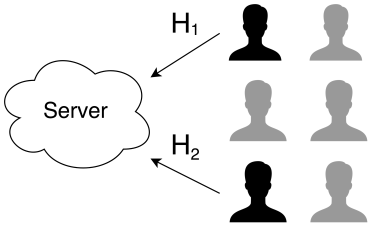

(a) The server selects $K$ users




(b) They receive the current model




(c) and compute updates using their data




(d) Updates are shared with the server

Figure 1: One communication round in a Federated Learning system

used. The update is then applied to the model using

$$\theta \leftarrow \theta - \sum_{i=1}^{K} \frac{n_i}{N} H_i \tag{1}$$

where $N = \sum_{i=1}^{K} n_i$ is the total number of data points used in this iteration. A new iteration begins after every model update.

In each iteration only $K$ users are queried for updates. While requesting updates from all users would lead to more stable model improvements, it would also be extremely expensive to do because there can be millions of users. Only querying a subset of them makes it more feasible to efficiently run many iterations.

This training process is then repeated until the model parameters converge, as determined by an appropriate criterion. In some situations, it can also make sense to keep the training running indefinitely. In case user preferences change, the model will automatically adapt correspondingly. Independently of the length of the training process, new models should be distributed to all clients from time to time. This ensures that they have a good model to use locally.

Federated Learning might seem similar to distributed machine learning in a data center. Parts of the problem setting are indeed comparable. Data is distributed across compute nodes, or users. Algorithms are used

in a MapReduce [23] kind of a way where updates are calculated on many computers at the same time and then combined afterwards [19].

However, Federated Learning is a vastly more distributed way of collaboratively training machine learning models. It can be distinguished by several key properties. These also describe the some of the challenges in Federated Learning:

1. **A huge number of users**: In a data center, there might be thousands of compute nodes. Popular consumer software has several orders of magnitude more users than that. All of these users should be able to participate in training the model at some point, so Federated Learning needs to scale to millions of users

2. **Unbalanced number of data points**: It is easy to guarantee that compute nodes have a similar number of data points in a data center. In Federated Learning, there is no control over the location of data at all. It is likely that some users generate vastly more data than others

3. **Different data distributions**: Even worse, no assumptions about the data distributions themselves can be made. While some users probably generate similar data, two randomly picked users are likely to compute very different updates. This is also unfortunate from a theoretical standpoint because no iid [25] assumptions can be made, i.e. random variables are generally not independently and identically distributed

4. **Slow communication**: Since compute nodes in Federated Learning correspond to users' computers, the network connections are often bad [106]. This is especially the case if the training happens on mobile phones [105]. Updates for complex models can be large, so this is problematic when training more sophisticated models

5. **Unstable communication**: Some clients might not even be connected to the internet at all when the server asks them to send back model updates. In a data center, it is much easier to guarantee that compute nodes stay online

In a nutshell, Federated Learning is a massively distributed way of training machine learning models where very little control over the compute nodes and the distribution of data can be exercised. The properties listed above motivate several of the next chapters.

## 1.4 Applications

The protocol introduced so far is fairly abstract and it remains to be discussed what exactly can be implemented with it. In general, it is possible

to use Federated Learning with any type of models for which some notion of updates can be defined. It turns out that most popular learning algorithms can be described in that way.

A large part of machine learning is based on *gradient descent*, a popular optimization algorithm that is described in more detail in Section 2.2. Gradient descent naturally transfers well to Federated Learning. The updates are partial derivatives that define in what direction the model parameters should be moved. Linear regression, logistic regression and neural networks are generally all optimized using gradient descent [35, 12]. Linear support vector machines and matrix factorization based methods, like collaborative filtering, can also be optimized with this method [72, 32, 51].

Federated Learning is not just limited to supervised learning. It can also be used in other areas, such as unsupervised and reinforcement learning. $k$-means is an unsupervised learning algorithm that by design works well in a Federated Learning setting: Updates specify how the centroids are moved [12, 56].

Even algorithms that are not usually described in terms of updates can be used with Federated Learning. For example, principal component analysis is based on computing eigenvectors and -values. These can also be calculated using the power-iteration method, which transfers well to distributed computation [56]. The model consists of eigenvector estimates which are continually improved by the clients.

Some algorithms, however, can not be reformulated for Federated Learning. For example, $k$-NN requires memorizing the data points themselves [12], which is not possible here. Non-parametric models in general can be problematic since their configurations often heavily depend on the exact data that was used to train them.

Independently of the model type, the kind of data that is available is another criteria which can be used to decide if it is reasonable to use Federated Learning. Of course, whenever data is too private to be transferred to a server, Federated Learning is a good choice. This is often the case in situations where users implicitly generate a lot of data, just by interacting with their device. In the best case, they also label the data in this process.

One example for such an application is showing users suggestions for the next word they might want to type on a mobile phone, a functionality offered by most virtual keyboards. Recurrent neural networks are generally used to implement this [65, 64, 84]. They try to predict the next word that is going to be typed by analyzing the previously typed words. Such a model could be trained on any language corpus, for example on text from Wikipedia. However, the language used on Wikipedia differs from the one used by people in daily life.

For this reason, directly training on the text typed by users would lead to better results. Conventional methods cannot be used to do this since the data is extremely private. It should not be sent to a server and should not

even be stored unnecessarily for a longer time. Federated Learning can still be used to train a model on this data [64].

This is an extremely elegant application of Federated Learning: People generate data points by typing on their devices and label these themselves as soon as they type the next word. The model can improve itself on the fly. While the user is typing, the model tries to predict the next word. As soon as the user finished typing the word, the correct label is available and the model can use this information to compute an update to improve itself. The data used to generate the update is directly discarded afterwards. This way, a high-quality model can still be trained, without making any sacrifice on privacy.

# 2 Background

## 2.1 Estimators

Often it is not possible or simply impractical to compute certain values exactly. This might be because it is too expensive computationally or because not enough information is available. Instead, these values can be estimated. The quality of estimates varies. In statistics, this concept is formalized in estimation theory [25, 41].

An *estimator* is a function that estimates a value based on other observations. This process can involve randomness. Reasons for this can for example be that the function itself is random or that there is random noise in the observations it uses. One measure for the quality of an estimator $\tilde{X}$ is its *bias* or how far off its estimate is on average from the true value $X$:

$$\text{bias}[\tilde{X}] = \mathbb{E}[\tilde{X}] - X$$

where the expected value is over the randomness involved in computing estimates.

If the bias of an estimator is 0, it is called an *unbiased estimator* [88]. This is generally a desirable property to have because it means that the estimator is correct on average. If one samples for long enough from the estimator, the average converges to the true value $X$. This is due to the law of large numbers [25].

**Theorem 1.** *If $k$ estimators $\tilde{X}_1, \ldots, \tilde{X}_k$ all produce unbiased estimates of $X$, then any weighted average of them is also an unbiased estimator. The new estimator is given by*

$$\tilde{X} = w_1 * \tilde{X}_1 + \ldots + w_k * \tilde{X}_k$$

*where the sum of weights $\sum_{i=1}^{k} w_i$ needs to be normalized to* 1.

*Proof.* The unbiasedness is due to the linearity of expectation:

$$\begin{aligned}
\mathbb{E}[\tilde{X}] &= \mathbb{E}[w_1 * \tilde{X}_1 + \ldots + w_k * \tilde{X}_k] \\
&= w_1 * \mathbb{E}[\tilde{X}_1] + \ldots + w_k * \mathbb{E}[\tilde{X}_k] \\
&= w_1 * X + \ldots + w_k * X \\
&= X
\end{aligned}$$

□

If an estimator is unbiased, its individual estimates can still be far off from the true value. While the mean of many sampled estimates eventually converges to the true expected value, this can take a long time, meaning the estimator is inefficient. To quantify how consistently an estimator is close

to the true value, another statistic is required. Commonly, the *variance* of the estimator is considered here. It is defined as the mean squared distance between the estimate and the value to be estimated:

$$\mathrm{Var}[\tilde{X}] = \mathbb{E}[(\tilde{X} - X)^2]$$

Many different things can be analyzed using estimators. For example, statistical models can be seen as estimators. They use observations, or data, to make predictions. These predictions are generally not perfect because randomness is involved and only a limited amount of information is available. Thus, it makes sense to analyze statistical models in terms of bias and variance.

A central problem when building models is balancing underfitting and overfitting. If the training data is just memorized, the model does not generalize well to new data. This is a case of overfitting. The opposite issue, only barely matching the pattern in the training data, is called underfitting.

This problem is also known as the *bias-variance tradeoff* [30, 12, 81]. If the model has a high bias, its predictions are off, which corresponds to underfitting. If overfitting occurred, i.e. the data is matched too well, the estimates have a high variance. By resampling the data that the model was built on, totally different estimates are generated. This is because the model is now based on different random noise.

Generally, it is not possible to perfectly optimize both, bias and variance, for statistical models, so they need to be balanced here. On the other hand, the focus in some of the following sections is purely on keeping estimators unbiased. Depending on how the estimators are used, different qualities are important.

## 2.2 Gradient Descent

Machine learning can be considered an optimization problem. In supervised learning, a *loss function* quantifies how close a prediction $f(x_i)$ of a model is to the correct answer $y_i$. The parameters of the model should be chosen to minimize the loss.

Generally this is done by optimizing them for the training data while also validating the model on otherwise unused data. The parameters with the best performance on the validation data are selected in the end. The full loss for a prediction function $f$ is given by:

$$L = \frac{1}{n} \sum_{i=1}^{n} \mathrm{loss}(f(x_i), y_i)$$

A popular choice for the loss function is the squared error:

$$\mathrm{loss}(p, y) = (p - y)^2$$

There are many algorithms for minimizing $L$. Perhaps surprisingly, a huge part of machine learning is based on a conceptually simple method called *gradient descent* [35, 12]. It is an iterative algorithm where the model parameters are repeatedly moved into a direction where they work a little bit better. To start off, the model is initialized with an arbitrary set of parameters.

Afterwards, the iterative optimization process begins. In each iteration, the partial derivatives of the loss with respect to the model parameters are computed. To be able to do this, gradient descent requires the prediction function $f$ and the loss function to be differentiable.

An important property of the vector of partial derivatives is that it points into the direction of steepest ascent. Because the loss should be minimized, the parameters are updated into the opposite direction. Since the vector of partial derivatives only points to the next optimum but does tell us how far to go, a scaling factor $\eta$, called the learning rate, is also applied.

Each parameter $\theta_i$ is then repeatedly updated using this idea:

$$\theta_i \leftarrow \theta_i - \eta * \frac{\partial L}{\partial \theta_i}$$

In each iteration, all parameters are updated at the same time using this formula. Note that this is conceptually very similar to Equation 1. The parameters are repeatedly improved, moving them closer and closer to a local optimum. Once a local optimum is reached, the gradient becomes 0 and no more updates are performed. In practice, the learning rate needs to be tuned well to make sure that the updates do not jump over an optimum. Figure 2 shows several gradient descent iterations using a contour plot of the loss and two weights.

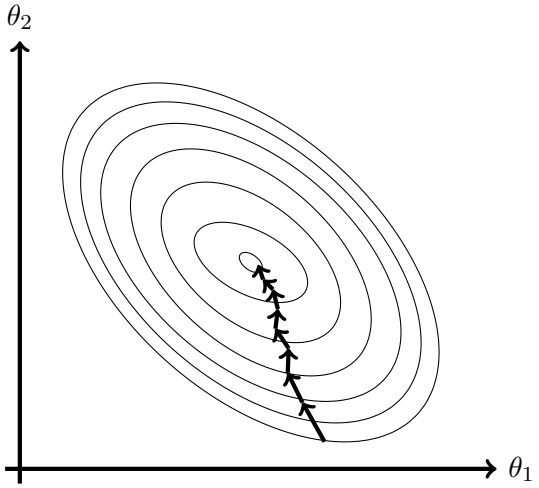



Figure 2: A two-dimensional illustration of gradient descent [107]

To simplify the notation when working with many partial derivatives, the gradient operator $\nabla$ is introduced. The gradient $\nabla L$ is the vector of all partial derivatives:

$$\nabla_i L = \frac{\partial L}{\partial \theta_i}$$

To calculate the gradient $\nabla L$ of the loss, we can make use of the linearity of the gradient operator:

$$\begin{aligned}\nabla L &= \nabla \frac{1}{n} \sum_{i=1}^{n} \text{loss}(f(x_i), y_i) \\ &= \frac{1}{n} \sum_{i=1}^{n} \nabla \text{loss}(f(x_i), y_i)\end{aligned}$$

In many cases, $n$ is a very large value and computing the full update $\nabla L$ is expensive. This is especially true if there is duplicated training data. If the training set consists of 10 copies of a different dataset, then the evaluation of the formula above is also unnecessarily expensive. Every required calculation is repeated 10 times. While this is an extreme example, it does happen in practice that much of the training data is similar. To save time, it often makes sense to only use a part of the data to estimate the gradient.

In *stochastic gradient descent* (SGD), a single data point $x$ and label $y$ are sampled uniformly from the training set. The true gradient $\nabla L$ is then estimated using

$$\nabla \tilde{L} = \nabla \text{loss}(f(x), y)$$

**Theorem 2.** *The SGD estimate $\nabla \tilde{L}$ is an unbiased estimator of $\nabla L$.*

*Proof.*

$$\begin{aligned}\mathbb{E}[\nabla \tilde{L}] &= \sum_{i=1}^{n} \frac{1}{n} \nabla \text{loss}(f(x_i), y_i) \\ &= \frac{1}{n} \nabla \sum_{i=1}^{n} \text{loss}(f(x_i), y_i) \\ &= \nabla L\end{aligned}$$

□

The computations for SGD can be performed very quickly but still give us an unbiased estimate of the true gradient. This property is the reason why optima can be found using this algorithm. While individual estimates are off, the randomness averages out over iterations and the parameters still move into a sensible direction overall. Since iterations are much cheaper, more of them can be performed.

These individual estimates can have a large variance however, leading to noisy and jumpy updates. A further improvement over this method is *mini-batch gradient descent*. Instead of just sampling one data point, we sample a small batch of $k$ examples. The estimated gradient is an average of all $k$ single estimates. By Theorem 2 each of these individual estimators is unbiased. Thus, their average also has to be an unbiased estimator, due to Theorem 1. In contrast to SGD however, there is much less variance, because more data is used to compute the estimate.

Most gradient computations can be formulated using linear algebra operations. These calculations can be parallelized very well on GPUs [68]. So with appropriate hardware there is no significant performance penalty for using $1 < k \ll n$ data points to compute the estimate. Because the variance is much smaller compared to SGD, mini-batch gradient descent typically has the best convergence rate in practice.

Finally, it is worth mentioning that gradient descent can only guarantee the convergence to local optima. This is typically not a problem in real-world applications because there are generally very few local optima in high-dimensional spaces. The likelihood that the loss does increase in all directions at the same time is extremely low. However, gradient descent can also get stuck in saddle points and plateaus, which occur more often [22]. Plateaus commonly happen when the gradient is dominated by one dimension, in which case updates mostly slide along this one dimension. If there is little improvement of the loss in this dimension, then the algorithm gets stuck on a plateau. A saddle point has less requirements than an optimum. While the gradient is 0 in all directions, it does not mean that there is no better point in the close neighborhood.

## 2.3 Computational Graphs

Using gradient descent requires being able to compute partial derivatives. For most well established models, it is known how these derivatives look like. The gradients for linear and logistic regression, for example, are easy to derive by hand and can be found in the literature [79, 82]. As learning algorithms become more complex, deriving the gradients gets harder. If one wants to test out new architectures, the process of manually figuring out the derivatives is time-consuming and error-prone.

The major innovation behind deep learning libraries are *computational graphs* [2, 74]. They are a smart abstraction that allow for an efficient and easy way of implementing automatic differentiation. Instead of viewing a function as one huge symbolic term, it is broken up into smaller pieces. These smaller pieces correspond to elementary functions like addition, multiplication or the exponential function. Derivatives for these functions are directly implemented.

To build up a large function, like a neural network, these small building

blocks are then composed and connected in a graph. Nodes in the graph represent functions. Edges show how data flows between those functions.

For example, the function

$$f(a, b, c) = (a + \exp(b)) * (exp(b) + c)$$

can be represented using the graph shown in Figure 3. This example also shows that sending data through the computational graph can be more efficient than naively evaluating a symbolic expression. The term $\exp(b)$ is only computed once when executing the computational graph for $f$. These graphs can be automatically created when describing the function in code. The respective libraries only need to be told which values represent variables and which ones should be treated as constants.

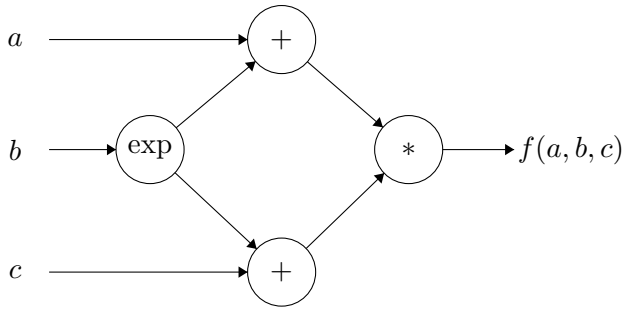



Figure 3: A simple example for a computational graph

After the graph has been constructed, backpropagation can be used to compute partial derivatives [35, 80]. The derivatives for elementary functions are already known, so they only need to be combined with each other and with the original functions themselves. This is done by applying the chain rule for function composition and other well-known rules for differentiation, e.g. for handling function addition.

In the same way that computation was saved when sending data through the graph, backpropagation is an extremely efficient way of computing derivatives. It is a dynamic programming approach to differentiation: By smartly computing the derivatives step by step and by reusing results, duplicated calculations are avoided. Another advantage of computational graphs is that the decomposed representation of functions makes it easier to efficiently schedule and parallelize the execution of individual computations.

Computational graphs as a software engineering abstraction make it much easier to quickly test out new ideas. They can be utilized for all models where the training process involves computing gradients. The following sections make use of them whenever architectures are adapted. This makes it much easier to apply the changes. No assumptions about the rest of the model architecture have to be made to be able to describe how the derivatives can be computed.

# 3 Federated Optimization

Machine learning algorithms require many iterations until the model parameters converge. A part of the field of optimization is concerned with developing methods for reducing this number of iterations. In Federated Learning this is particularly important because individual iterations take much more time. Since the server is idle until users respond back with updates, it is plausible that much more time is spent waiting rather than actually computing something. This is especially the case if the server needs to wait until users generate new data.

This chapter shows how the optimization parts of Federated Learning can be improved. The first subsection begins by analyzing some fundamental properties of training using distributed data. Based on this, the remaining subsections focus on speeding up convergence. This can be done by selecting users in a smarter way, by improving the quality of their proposed updates and by stopping the training process at a good time.

## 3.1 Distributed Training Data

The Federated Learning protocol described earlier implements an adapted form of mini-batch gradient descent. To estimate the full update more efficiently, only some data points are used in each iteration. But since the data points are partitioned across users, they are not sampled completely independently of each other anymore.

To start off analyzing what this means, consider a simplified protocol where in each iteration only a single user is sampled. The $i$-th user is selected with a probability of $\frac{n_i}{n}$. Their $j$-th data point and label are denoted by $x_{ij}$ and $y_{ij}$ respectively. The full update proposed by the $i$-th user is then given by

$$H_i = \sum_{j=1}^{n_i} \frac{1}{n_i} \nabla \operatorname{loss}(f(x_{ij}), y_{ij})$$

The remainder of this thesis generally assumes updates to be gradients. However, the same analysis transfers over to any mechanism for computing updates that is linear like the gradient operator.

To show that the update estimate of a single user is unbiased, we need to iterate over all users, denoted by $k$. For each one, their proposed update and the probability that they were sampled is considered:

$$\begin{aligned} \mathbb{E}[\nabla \tilde{L}] &= \sum_{i=1}^{k} \frac{n_i}{n} H_i \\ &= \sum_{i=1}^{k} \frac{n_i}{n} \sum_{j=1}^{n_i} \frac{1}{n_i} \nabla \operatorname{loss}(f(x_{ij}), y_{ij}) \end{aligned}$$

$$= \frac{1}{n} \sum_{i=1}^{k} \sum_{j=1}^{n_i} \nabla \text{loss}(f(x_{ij}), y_{ij}) \tag{2}$$
$$= \frac{1}{n} \sum_{i=1}^{n} \nabla \text{loss}(f(x_i), y_i)$$
$$= \nabla L$$

where after Equation 2 we switch from a nested summation over all users and their data points to a direct summation over all data points. This is possible because $\sum_{i=1}^{k} n_i = n$.

Even though the data is arbitrarily distributed among users, an unbiased estimate can still be computed. This also holds when $K$ users are sampled at the same time. The final estimate in that case is an average of the individual estimates, weighted by the number of data points that the clients used. By Theorem 1, this is an unbiased estimator again.

While the distribution of the training data has no influence on the bias of the estimator, it can have a huge effect on the variance. If it can be controlled where the data is located, it is easy to ensure that all users produce similar estimates. In that case, the variance of the estimator is low and convergence can be reached quickly. In Federated Learning, such control over the data is not possible and the variance can be large.

As an extreme example, consider a situation where users only have two kinds of data points, which are extremely different from each other. If the location of training data can be changed, like in a data center, each compute node can have a similar number of both data points. In Federated Learning, it is possible that users either only have the first kind of data points or only the second. In that case, the two kinds of users will produce very different gradient estimates, leading to a larger variance.

## 3.2 Sampling Techniques

To improve convergence, it is helpful to reduce the variance of the update estimates. Variance reduction is a well-understood topic in statistics, with common strategies that can be implemented [15, 25]. Some of these methods based on smarter sampling are introduced in this section.

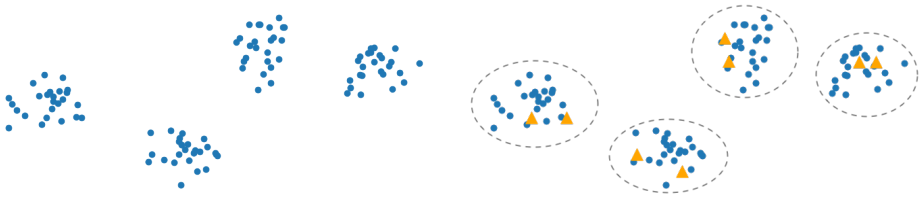
The technique used so far is based on *simple sampling*. Users are chosen independently of each other, weighted by their number of data points. As a result, all data points have the same probability of being used in an iteration. This can lead to bad results because the set of selected users is not necessarily representative of the entire population.

The field of statistics has developed alternative sampling strategies that explicitly aim to sample a diverse set of people. This has applications in many areas. For example, when performing surveys for elections or when

testing new drugs, it is important to base the experiment on a representative subpopulation. Rather than choosing people completely at random, it should be ensured that individuals of different groups are represented.

Two common sampling strategies to do this are *stratified sampling* and *cluster sampling* [25]. Both try to encourage sampling a diverse set of people by assigning them to groups. The two methods differ in how these groups and the relationship between individuals in the groups are defined.

In stratified sampling, each user is assigned to a *stratum*. Strata are meant to group similar users together, forming collections that are homogeneous internally. To sample users, a simple sampling is performed in each stratum. The number of users sampled in a group depends on its size. A visualization of this idea is shown in Figure 4.



(a) Users are represented by two features (b) Strata and the sampled users

Figure 4: Stratified sampling: Random sampling in each stratum

To assign people to strata, their personal features are used. Political surveys commonly choose strata based on demographic factors. In Federated Learning, this is not done by using the data stored on the users' computer, which is meant to be used by the learning algorithm. Instead, some metadata about users is often already known on the server and can be used for stratified sampling.

For example, users could be grouped based on the region in which they are located. This is especially useful if the data of users highly differs depending on their region, e.g. when training on language data. A simple sampling is unlikely to select a representative group of users since only a few users of the regions where it is currently night are online. Stratified sampling, on the other hand, ensures that regardless of the time, users from different regions are always represented in the same way.

Cluster sampling is a related strategy that also assigns users to groups, called *clusters*. In contrast to strata, clusters represent heterogeneous groups of people. Each group by itself is meant to be representative of the entire population.

This technique is commonly used whenever a natural way of dividing the population into similar groups exists. For example, when a new drug is to be tested in a country, clusters could represent villages. A simple sampling of individuals across the entire country would lead to high costs for the study.

Instead, cluster sampling can be used to select people from a handful of villages. This way, a study can be performed on a representative group of people in a much cheaper way. Cluster sampling generally selects all people that are assigned to the sampled clusters.

How exactly stratified and cluster sampling can be implemented for Federated Learning heavily depends on the application and on the meta data that is available. Generally, it is useful to do whenever a sensible way of grouping users on the server exists.

## 3.3 Improving Individual Update Quality

The previous section purely focused on carefully selecting the users that participate in an iteration. After this set of users is fixed, the quality of their proposed updates should be improved as much as possible. Generally this works by increasing the amount of work performed on each user's computer.

In the case of gradient descent, this idea can be implemented by performing multiple update steps locally before sending anything to the server [63]. By running several iterations, the update that is computed can be of much higher quality. This is often the case because it adds degrees of freedom to the update. only taking one step, the update has to be alongside the gradient of that iteration. By allowing several iterations, the update can move in different ways and has more chances of properly going into an optimum.

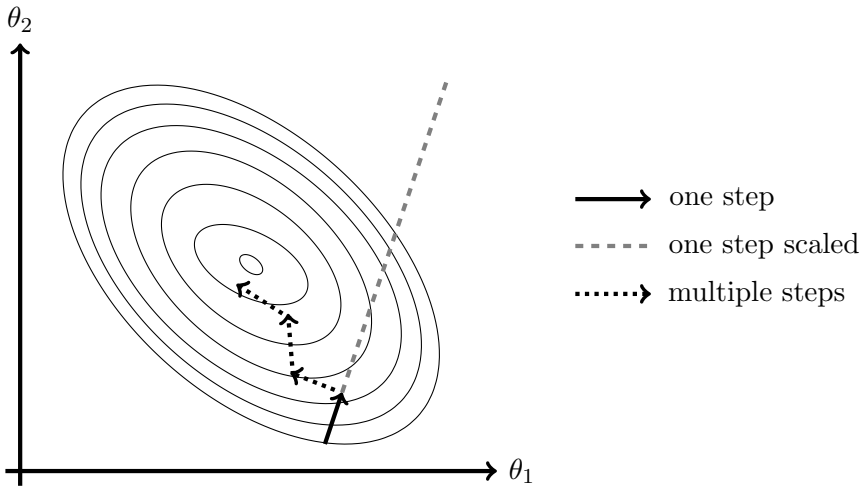



Figure 5: One step of gradient descent step compared to taking multiple [107]

An example of this is given in Figure 5. The solid black arrow shows the update performed after a single step of gradient descent. The dashed gray line visualizes how this update could have changed the loss if a different learning rate would have been used. In contrast to this, the dotted black arrows visualize the improvement when taking several steps of gradient de-

scent with a sufficiently small learning rate. By using the latter approach, the loss can be minimized much better. This effect is even stronger when the loss surface is more complex since individual updates usually only lead to small improvements in such a space.

Running several iterations locally before synchronizing with the server is a purely heuristic method. There are no theoretical guarantees that this technique improves convergence. When the learning rate is chosen poorly, the updates jump passed the optimum. Doing this multiple times decreases the quality of the update further and further. The average that the server computes is also not an unbiased estimator anymore. Because clients apply different intermediate updates locally, the final updates they sent to the server are generally based on different parameters.

Still, empirically this approach has led to large improvements. In some simulations it has been shown that running multiple iterations locally improves convergence by a factor between 3 and 100 [63]. Computationally this is also not a significant amount of additional work. Running a few additional steps of gradient descent locally is cheap since individual users only have little data. Other optimization algorithms can be adapted in similar ways, by essentially removing synchronization points to perform additional local work.

## 3.4 Early Stopping

Neural networks are universal function approximators [20]. They can approximate any function in $C^\infty$ arbitrarily well, given that they are composed of a sufficient number of neurons [44]. The set $C^\infty$ contains all functions that have derivatives of all orders. This allows neural networks to also approximate a function that acts as a lookup table to the training data. If the model is then only evaluated on the training data, it could work arbitrarily close to perfect.

Of course, such a model would also generalize very badly. Unseen data is not in the lookup table and so the accuracy based on such data is likely to be much worse. To check if overfitting occurred, the training accuracy should be compared to the accuracy on otherwise unused data. This idea remains the same if other metrics than accuracy are used.

Since neural networks are eventually going to learn a function that memorizes the training data, it is not enough to simply check for overfitting after the end of the training. As the network begins to overfit, the validation accuracy stops improving, as visualized in Figure 6. The standard solution to this problem is *early stopping* [76, 35]. After each iteration, the training accuracy is compared to the accuracy that is reached on validation data. In the end, the model with the best validation accuracy is selected.

Even though this is a straightforward solution, it is used in the training process of nearly any neural network. Generally the training is stopped if

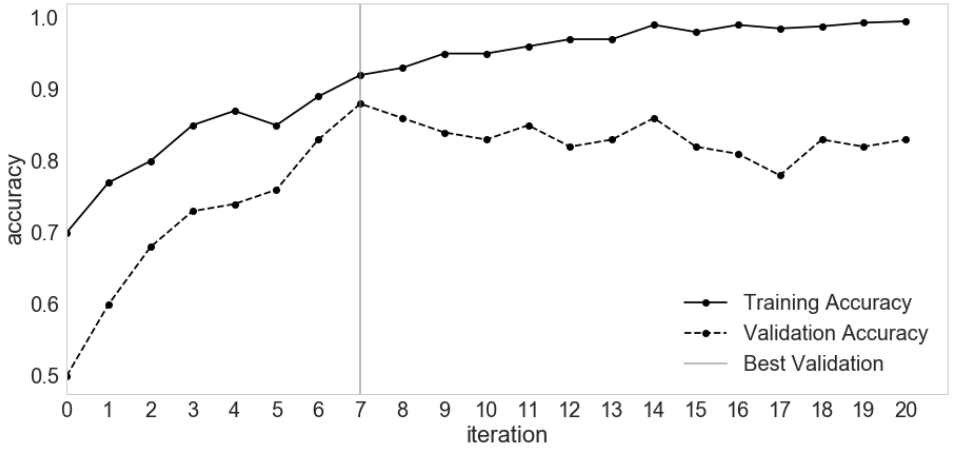



Figure 6: Overfitting after seven iterations

there was no improvement for a certain number of iterations. Supervising the learning process is also a useful tool when training other models. It can be used to see if the learning process can be stopped because the training converged. In the case of Federated Learning, it is additionally helpful to check if there are problems with the implementation or to see if everything is converging as expected.

Evaluating on validation data is distinctly different to using a test set. The validation set is used to choose between different models. The test set is only meant to be used once in the very end, purely to evaluate the quality of the selected model.

Testing is simple to implement for Federated Learning. After the end of the training process, users only report back about the model quality, instead of computing updates. On the other hand, evaluating using validation data and early stopping can be implemented in different ways. A straightforward solution is assigning the users sampled in an iteration to training and validation groups. Depending on their group, users either send back weight updates or validation feedback. This is equivalent to the standard approach of partitioning a dataset into training, validation and test sets in statistics.

For Federated Learning, this might be a bad solution because the validation set could be small, leading to much noise and variance in the validation feedback. A more elegant solution is possible when training on a stream of data. This means that data is only looked at once during training and directly discarded afterwards. An example of this is training a model to predict the next word. The model is trained while the user is typing, but none of that data is stored persistently. Each data point used during the training was just generated by the user and is directly forgotten afterwards.

In such a situation, all sampled users can provide both updates and validation feedback. When a new data point is generated, it is used to

compute the validation accuracy of the model in the current time step. Afterwards, it is used to calculate a model update for the next iteration. The data point is then completely discarded.

This is a proper way of validating because the validation feedback is based on data that was not used in the training of the evaluated model. The data for computing the validation accuracy for the model of time step $i$ is only used for training the model of time step $i + 1$. It is also a very efficient way of using the data. The same data points are used for both training and validating, meaning no data is lost unnecessarily. This has the effect that validation accuracy is less noisy because it can be based on more data. At the same time, the communication requirements are not increased significantly and no additional users are required.

This idea can only be applied if all of the data used is freshly generated in each iteration. If data is stored and reused across iterations, then there is a problem of information leakage between the model and the data that is used to evaluate it. In that case, the set of users has to be cleanly partitioned into two groups instead.

## 3.5 RProp

Vanilla gradient descent, as introduced earlier, requires a learning rate $\eta$ that is used to scale the gradient. Selecting this hyperparameter well is tremendously important. If it is too large, the optimizer quickly moves to a local optimum in the beginning. However, the learning rate will be too large to properly land in that optimum. Instead, the weights will oscillate around the local optimum.

If the learning rate is too small, we steadily keep going closer to the next local optimum. Still, the optimum will not be reached in any reasonable amount of time. A visualization of these cases is shown in Figure 7.

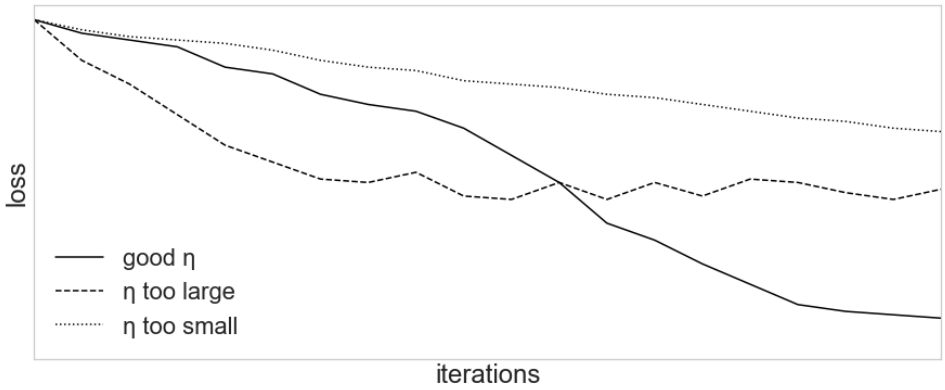



Figure 7: The learning rate is crucial to make the optimization work

Generally, the learning rate is selected by testing many values at different

orders of magnitude. This requires an expensive naive search. For this reason, various gradient descent variations have been developed that improve on this.

In Federated Learning, this is particularly important. Testing out different learning rates requires shipping the client part of the Federated Learning system to users and then trying out the respective learning rates in production. This is not desirable because most learning rates are not going to work well, leading to a bad user experience. Additionally, it takes a long time to try many different values like this.

The fundamental problem in the Federated Learning case is that we have no information about gradient magnitudes before deploying the system. Just collecting weight updates, without using them to compute new models, is not sufficient because the gradient magnitudes might differ strongly depending on the iteration. Since no data should be collected directly, an algorithm that can automatically deal with any gradient magnitudes is required.

One gradient descent variant that dynamically adapts the learning rate is *RProp* [78, 80], short for *Resilient Propagation*. A property that makes RProp unique among gradient descent optimization algorithms is that it uses only the signs of the components of the gradient and completely ignores its magnitude. Since RProp is a general optimization algorithm, its motivation is independent of Federated Learning.

To understand how RProp works and why it uses only the signs, it is worth taking a step back to consider most other popular gradient descent variants. They typically use the signs of the gradient components as well as the magnitude. The gradient points in the direction of steepest ascent. To find a local minimum, we go into the opposite direction. This direction is completely determined by the sign alone.

To decide on the step size, a scaled version of the gradient's magnitude is generally used. This heuristic often works well but there is no guarantee that it is always a good choice. To see that it can work extremely badly, and does not have to contain valuable information, we consider a function $f$. Figure 8 shows $f$ as well as two scaled versions.

All three of these functions have the exact same optima, so the step updates using gradient descent should all be similar. However, if we determine the step size using the gradient's magnitude, then the step sizes for the three functions differ by orders of magnitude. Even worse, the gradient virtually vanishes for the second function and explodes for the third. This is illustrated in Figure 9. The values of the derivative of the third function are either very small or very large unless they are extremely close to the original local optima.

This shows that the gradient's magnitude does not necessarily contain useful information for determining the step size. Even though optima can still be found by choosing appropriate learning rates, this makes it clear that using the gradient's magnitude at all is sometimes questionable. Using

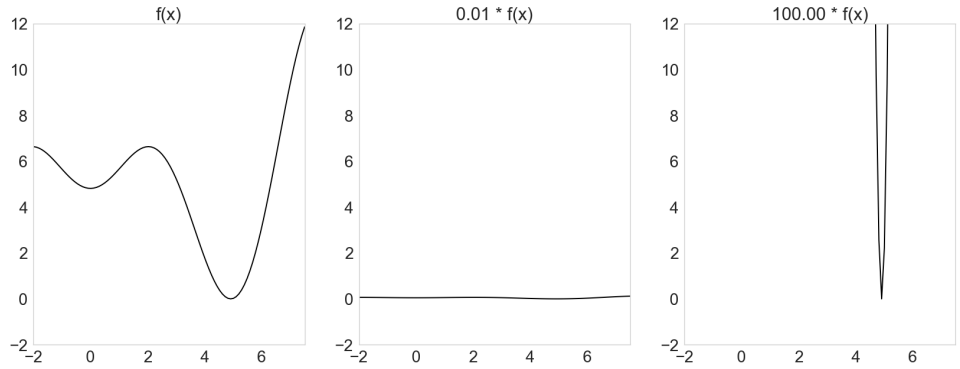



Figure 8: Three functions with the same optima but vastly different gradients

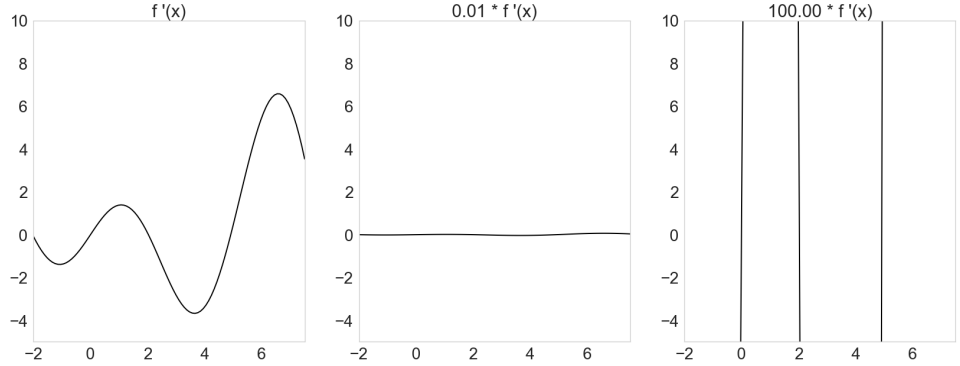



Figure 9: The first derivatives of the three functions

a fixed learning rate will also fail if only some parts of the function are scaled.

Modern gradient descent variants try to circumvent this problem by dynamically adapting the step size. RProp does this in a way that only requires the sign of the gradient components. By ignoring the gradient's magnitude, RProp has no problems if a function has a few very steep areas.

To implement this, RProp uses a different step size for each dimension. Let $\eta_i^{(t)}$ be the step size for the $i$-th weight in the $t$-th iteration of gradient descent. The value for the first and second iteration, $\eta_i^{(0)}$ and $\eta_i^{(1)}$, is a hyperparameter that needs to be chosen in advance. This step size is then dynamically adapted for each weight, depending on the gradient.

The weights themselves are updated using

$$\theta_i^{(t)} = \theta_i^{(t-1)} - \eta_i^{(t-1)} * \operatorname{sgn}\left(\frac{\partial L^{(t-1)}}{\partial \theta_i^{(t-1)}}\right) \tag{3}$$

where the sign of the partial derivative of the error in the last step with respect to the respective weight is computed. We go into the direction of

descent using the determined step size.

In each iteration of RProp, the gradients are computed and the step sizes are updated for each dimension individually. This is done by comparing the sign of the partial derivative of the current and previous iteration. The idea here is the following:

- When the signs are the same, we go into the same direction as in the previous iteration. Since this seems to be a good direction, the step size should be increased to go to the optimum more quickly
- If the sign changed, the new update is moving into a different direction. This means that we just jumped over an optimum. The step size should be decreased to avoid jumping over the optimum again

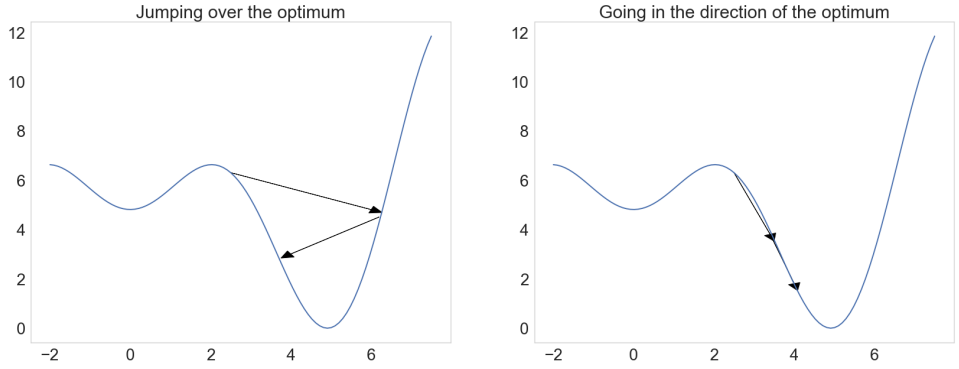



Figure 10: A change in gradient direction means we jumped over an optimum

A visualization of this idea is shown in Figure 10. To implement this update scheme, the following formula is used:

$$\eta_i^{(t)} = \begin{cases} \min(\eta_i^{(t-1)} * \alpha, \eta_{\max}) & \text{if } \frac{\partial L^{(t)}}{\partial \theta_i^{(t)}} * \frac{\partial L^{(t-1)}}{\partial \theta_i^{(t-1)}} > 0 \\ \max(\eta_i^{(t-1)} * \beta, \eta_{\min}) & \text{if } \frac{\partial L^{(t)}}{\partial \theta_i^{(t)}} * \frac{\partial L^{(t-1)}}{\partial \theta_i^{(t-1)}} < 0 \\ \eta_i^{(t-1)} & \text{otherwise} \end{cases} \tag{4}$$

where $\alpha > 1 > \beta$ scale the step size, depending on whether the speed should be increased or decreased. The step size is then clipped using $\eta_{\min}$ and $\eta_{\max}$ to avoid it becoming too large or too small. If a gradient was zero, a local optimum for this weight was found and the step size is not changed.

These seem like many hyperparameters to choose, but in practice there are known values for them that generally work well [47]. It is also not problematic if the clipping values $\eta_{\min}$ and $\eta_{\max}$ are respectively smaller and larger than necessary because an inconvenient step size is usually adapted quickly. Popular values for $\alpha$ and $\beta$ are 1.2 and 0.5. Heuristically, it works well to increase the step size slowly, while allowing for the possibility of

quickly decreasing it when jumping around an optimum. For fine-tuning the weights, it is important that $\beta$ is not the reciprocal of $\alpha$, to allow for many different step sizes.

One advantage of RProp that was not discussed so far is having a different step size for each weight. If one weight is already very close to its optimal value while a second weight still needs to be changed a lot, this is not a problem for RProp. Other gradient descent variants can have much more problems with such a situation, especially because the gradient magnitudes can be misleading here.

To use RProp in Federated Learning, clients send updates in form of gradients to the server. The server then averages the gradients and updates the learning rates and weights using Equations 4 and 3 respectively.

A potential disadvantage of RProp is that it is highly sensitive to random noise [78]. If the batch size, or the number of sampled users in Federated Learning, is too small, then this can lead to the average update having a high variance. This can significantly limit how close RProp can go to an optimum. If the signs of the partial derivative keep changing, then the learning rates effectively stay nearly stable or at least only change slowly. This can be because the signs just fluctuate much due to random noise, even though a larger batch size would lead to a stable movement in one direction. In practice, this is not necessarily a problem as long as it is possible to sample enough users in every iteration.

All in all, RProp can be highly useful for Federated Learning because it is not difficult to initialize its internal values well. Even if the initial learning rate is off, it is going to adapt itself quickly. Additionally, we do not need prior knowledge about the user data to make RProp work well in our distributed optimization setting.

# 4 Compression

## 4.1 Communication Cost

A naive implementation of the previously introduced Federated Learning protocol scales badly when used with more complex models. This is due to the fact that the protocol requires a lot of communication between clients and the server. In each iteration, sampled clients download a copy of the current model and upload updates for all of its parameters. This has the effect that the amount of required communication grows proportionally with the size of the model.

In the past few years, models have become much larger, with neural networks consisting of more and more layers. As of 2018, it is not uncommon anymore to use neural networks with millions of parameters. Overviews of models have shown that the number of parameters, and with them the total computational cost, of the largest models has doubled every three to four months between 2012 and 2018 [94].

In Federated Learning, it is often cheap to compute the updates on the computers of individual users since they generally have little training data compared to the entire population of clients. The more expensive part is communicating the updates. This is especially the case for mobile phones which often only have access to much slower connections [105]. Since the communication effort is the bottleneck of Federated Learning, this section is dedicated to compression techniques for reducing the required bandwidth.

To tackle this problem, it is important to keep in mind that network connections are generally asymmetric [106]. Downloading data is much faster than uploading it. In most countries, in 2018, downloads are on average between three and ten times faster than uploads [106, 105]. This section entirely focuses on compressing the upload of updates, to make them as efficient as downloads. Compressing the models themselves for fast downloading is a separate problem, with different goals and known algorithms.

There are many obvious techniques that can be used to improve the previously introduced protocol. For example, updates could be collected and only be sent once a good network connection becomes available. Furthermore, standard lossless compression techniques can be applied to reduce the number of bits required to encode the updates.

Instead of discussing these in more detail, this section is concerned with dedicated lossy compression techniques for Federated Learning. Fundamentally, these methods try to perform a distributed mean estimation with very little communication. Since the server only requires the mean to update the model, this is the only statistic that needs to be accurate for Federated Learning to work. As a result, it is completely acceptable to change individual update reports, to encode them using fewer bits, as long as the overall mean stays stable.

This differentiates the techniques in this section from standard compression techniques. Compressing the entire model for downloads is a another very different problem, where the goal is to keep the predictions as stable as possible [39]. This can however significantly decrease the quality of the model, and might thus not always be an option.

From a theoretical perspective, we would like to reduce the communication requirements for uploads by more than a constant factor. This will ensure that updates can still be transmitted efficiently , even if the bandwidth of network connections does not increase as quickly as the size of common models in the future. From a practical standpoint, this is not entirely necessary. Decreasing the size of uploads by an order of magnitude makes uploads as efficient as downloads. Since compressing the model itself for downloads might not be desirable, a constant compression factor for uploads can already be sufficient. Some of the methods introduced in the next subsections have hyperparameters that allow for either a constant compression factor or one that depends on the model size.

The compression algorithms presented here fall into several categories. *Sketched update* techniques aim to provide unbiased estimators of the true updates. These estimates are the true update on average but require fewer bits to encode them. *Structured update* methods change the optimization process itself in order to allow for more compact representations. Finally, it is discussed how RProp can be useful for efficiently encoding updates.

For simplicity's sake, all the methods introduced are described for a single matrix of weights. Of course, if the model consists of several matrices, the methods can be applied individually to all of them. The true weight update matrix is denoted by $H$, the compressed update matrix by $\tilde{H}$. Individual elements of these matrices are denoted by $h$ and $\tilde{h}$ respectively.

## 4.2 Sketched Updates

Sketched update methods first compute the full update and then compress it afterwards [50]. In earlier sections, we sometimes made use of unbiased estimators to efficiently approximate some values. This concept can also be applied to the compression of updates.

Concretely, sketched update methods want the compressed update $\tilde{H}$ to be an unbiased estimator of the true update $H$. This means that the compressed update is the true update on average, even though it can be encoded much more efficiently. This way we can still compute an unbiased estimate of the true gradient, just like when only working with a subset of clients in each iteration. These methods also give us a theoretical justification on why they work.

### 4.2.1 Sparse Masks

A straightforward approach for reducing the communication cost is forcing updates to be sparse. In this method, each user is only allowed to update a certain number of weights. The exact number, or the probability $p$ of being allowed to update a given weight, is a hyperparameter that needs to be tuned for the problem at hand. Because users only communicate updates for a subset of weights, the communication costs can be reduced drastically.

Which weights a client is allowed to update is chosen randomly on the user's device. A different random seed is used by each client. The random mask matrix $M$ of a client is then sampled from a Bernoulli distribution using the predefined probability $p$ of being allowed to update a given weight:

$$M_{ij} \sim Bernoulli(p)$$

This mask matrix has the same dimensions as the weight matrix $W$. To tell the server which weights a client updated, it only sends the random seed to the server. The mask matrix itself is not sent because the seed is enough to fully reconstruct it.

The weights that were not selected by the mask are treated as constants. Updates for the other parameters are handled as always. If $H$ is the normal update matrix, then the sketched update matrix $\tilde{H}$ is given by

$$\tilde{H} = (H \circledast M) * \frac{1}{p}$$

where $\circledast$ is the Hadamard product, i.e. pointwise multiplication:

$$(A \circledast B)_{ij} = A_{ij} * B_{ij}$$

It is clear that this method allows for a large compression factor because clients only send updates for the weights selected by the mask. It remains to discuss why this method can work well. Fundamentally, this is because no information is lost on average. Since $M_{ij}$ is Bernoulli-distributed, it is given that

$$\mathbb{E}[M_{ij}] = p$$

By using the linearity of expectation, it follows:

$$\begin{aligned}
\mathbb{E}[\tilde{H}] &= \mathbb{E}[(H \circledast M) * \frac{1}{p}] \\
&= (H \circledast \mathbb{E}[M]) * \frac{1}{p} \\
&= (H * p \circledast \mathbb{1}) * \frac{1}{p} \\
&= H \circledast \mathbb{1} \\
&= H
\end{aligned}$$

where $\mathbb{1}$ is the matrix that only contains ones, i.e. $\mathbb{1}_{ij} = 1$ for all $i, j$.

The average compressed update is thus the true update. To account for that fact that many zeros are summed up, we need to multiply with the reciprocal of $p$. In the actual implementation this would not be necessary as the masked entries are not sent to the server, so it is sufficient to directly compute the estimate.

While this method based on sparse random masks is conceptually simple, it also allows for a good compression. To ensure that the upload can be performed as quickly as the download, it is sufficient to choose $p = \frac{1}{10}$. A sublinear compression is also possible, for example by choosing $p$ so that the expected number of updated parameters per client is logarithmic in the number of total weights $m$:

$$p = \frac{\log m}{m}$$

Since the elements of $M$ are Bernoulli-distributed, we would thus expect $m * p = \log m$ updated parameters per clients. Even though these are strong compression factors, the mean is going to stay stable as long as we sample enough users.

### 4.2.2 Probabilistic Quantization

The method of the previous section reduced the communication cost by only sending selected elements of the update matrix to the server. An alternative to this is sending the entire update matrix but encoding each element of the matrix using fewer bits. *Probabilistic quantization* is such a technique. It is a sketched update method, so the full update matrix is first computed. Afterwards, each element of this matrix is compressed individually, totally independently of the others.

Instead of directly introducing this compression algorithm, we start off with a simpler binarization approach and then extend it to use multiple bits and probabilistic techniques. Let $h_{\min}$ and $h_{\max}$ be the smallest and largest values of the update matrix that we want to compress. To compress an element $h$ of the update matrix, it is compared to $h_{\min}$ and $h_{\max}$ and the closer value is selected:

$$\tilde{h} = \begin{cases} h_{\max} & \text{if } h_{\max} - h \leq h - h_{\min} \\ h_{\min} & \text{otherwise} \end{cases}$$

Because only two values are possible for every compressed weight, this information can be encoded using a single bit. If $h_{\max}$ was selected the bit is 1, otherwise it is 0. Additionally, $h_{\max}$ and $h_{\min}$ are sent to the server. Before the compression, $h$ was a 32- or 64-bit float, so a large compression factor is achieved using this method.

However, it is also a very lossy compression. As an extreme example, if $h_{\min} = 0, h_{\max} = 1$ but most values are marginally below 0.5, then mostly 0 is used for the compressed updates. If all users have similar such update matrices, then most averages computed on the server are going to be very different from the true averages.

One potential solution to this problem was already shown in the previous section. The compressed value $\tilde{h}$ should be an unbiased estimator of the true update $h$. To implement this idea here, the distances from $h$ to $h_{\min}$ and $h_{\max}$ are interpreted as probabilities:

$$\tilde{h} = \begin{cases} h_{\max} & \text{with probability } (h - h_{\min})/(h_{\max} - h_{\min}) \\ h_{\min} & \text{with probability } (h_{\max} - h)/(h_{\max} - h_{\min}) \end{cases} \tag{5}$$

where $(h - h_{\min}) + (h_{\max} - h) = h_{\max} - h_{\min}$ is used as a normalization factor. A large distance of $h$ to one of the possible values corresponds to a large probability for the other value. This is visualized in Figure 11.

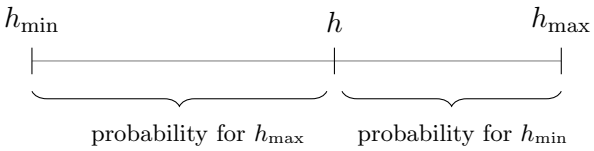



Figure 11: A visualization of the probabilities in probabilistic binarization, assuming normalized distances

In the case of the example distribution of $h$ that was previously discussed, this scheme works much better. Now, values of 0 and 1 are selected with similar frequency, leading to a good estimate of the true average. In the more general case, we can show that $\tilde{h}$ is truly an unbiased estimator of $h$ by using the definitions of $\tilde{h}$ and the expected value:

$$\begin{aligned}
\mathbb{E}[\tilde{h}] &= h_{\max} * \mathbb{P}[\tilde{h} = h_{\max}] + h_{\min} * \mathbb{P}[\tilde{h} = h_{\min}] \\
&= \frac{h_{\max} * (h - h_{\min}) + h_{\min} * (h_{\max} - h)}{h_{\max} - h_{\min}} \\
&= \frac{h_{\max} * h - h_{\max} * h_{\min} + h_{\min} * h_{\max} - h_{\min} * h}{h_{\max} - h_{\min}} \\
&= \frac{h_{\max} * h - h_{\min} * h}{h_{\max} - h_{\min}} \\
&= \frac{h * (h_{\max} - h_{\min})}{h_{\max} - h_{\min}} \\
&= h
\end{aligned}$$

If many users have similar weight update distributions, this is likely to be a good approach. Individual compressed updates might be far off from

the true update, but by averaging these differences will even out. In such a case, probabilistic binarization leads to little information loss although much fewer bits are required to communicate all updates.

For some other situations, probabilistic binarization will lead to a large error. This is the case if many users have different distributions or $h_{\min}$ and $h_{\max}$ are far away from each other. In such a case, additional bits can be used to reduce the error introduced by the compression scheme. The process is then called *quantization* instead of binarization. If $k$ bits are used, $2^k$ values can be encoded. These are selected by evenly segmenting the range $[h_{\min}, h_{\max}]$.

To compress an update $h$, the two closest possible values are determined. The probabilistic binarization scheme from Equation 5 is then used to decide between these two values. The resulting formula is still an unbiased estimator because the selected $h_{\min}$ and $h_{\max}$ are definitely smaller and larger than the respective $h$. This was the only requirement for $\tilde{h}$ to be an unbiased estimator.

To determine the two closest possible values efficiently, binary search could be used. But since we have the full information of how the possible values are computed, there is a closed formula that can be used to compute the index of the next smallest value. The length of the interval $[h_{\min}, h_{\max}]$ is given by $|h_{\max} - h_{\min}|$. The $k$ bits allow us to encode $2^k$ values which means there are $2^k - 1$ possible pairs of values that could be used for $(h_{\min}, h_{\max})$ in Equation 5.

To find the lower bound that should be used, we first compute the distance between the individual possible values. The distance $d$ is then given by:

$$d = \frac{|h_{\max} - h_{\min}|}{2^k - 1}$$

where the numerator corresponds to the total length of the interval and the denominator to the number of possible values for the lower bound. Subtracting 1 is necessary because the largest value that can be encoded cannot be used as the value for $h_{\min}$ in Equation 5.

The index $i$ of the closest smaller value can then be computed as follows:

$$i = \left\lfloor \frac{h - h_{\min}}{d} \right\rfloor$$

In the numerator, we compute the distance of $h$ to the start of the interval. This is then divided by the distance between the individual values. The index of the larger value is then given by $i + 1$. The formula allows finding the two closest values in constant time, independently of the number of used bits.

## 4.3 Structured Updates

In the previous subsection, the computation of updates was performed completely independently of the compression. We were able to show that the expected compressed version is equal to the true value. However, individual compressions can still be very lossy. While it is not possible to completely get rid of an error without switching to lossless compression algorithms, we still want to ensure that the compressed update leads to a good next model.

Structured update techniques enforce a certain form on the update that allows for a more efficient representation. The fact that the update has to have this structure is taken into account during the optimization process itself. Generally, this is implemented by adapting the computational graph of the model.

By doing this, the optimizer can find updates that follow the enforced structure but still minimize the loss. In other words, we can find a locally optimal update that can be encoded efficiently, since it has the predefined structure. In this process, the theoretical guarantees of unbiased estimators are lost, and the methods are only motivated on a heuristic level.

### 4.3.1 Sparse Masks

In the sparse mask compression method for sketched updates, each client was only allowed to update a certain set of weights in every iteration. This idea can be extended into a structured update technique. If a client can only update certain weights, then it could take this information into account during the optimization process.

Instead of computing a normal weight update and then making it sparse afterwards, updates should only be computed for the weights selected by the mask. To implement this, the other weights are marked as constants in the computational graph. When computing the updates like this in one SGD step, the results are not going to differ from the ones of the sketched method. In the process of computing partial derivatives, all the other weights are treated as constants anyways, so this does not change any results yet.

However, a structured update method allows us to change the optimization process itself. To this end, we perform several steps of SGD based on the sparse updates. After the updates of the first step were computed, they are applied locally and new sparse updates are computed. Only the weights chosen by the mask are updated in all these steps.

Like in the sketched update method, the final result is a sparse update matrix that can be efficiently shared with the server. The difference is that the updates from individual clients can be much more meaningful. By taking several SGD steps, the updates are likely to have a higher quality, as outlined in Section 3.3.

One disadvantage compared to the sketched update method is that the-

oretical guarantees are lost. The resulting matrix $H$ is not an unbiased estimator of $H$ anymore. Still, the empirical results from performing several SGD steps show that this might be a promising approach.

### 4.3.2 Matrix Decomposition

Another method for enforcing a structure on the updates in a way that allows for an efficient encoding is based on *matrix decomposition*. Instead of directly sending $H$, we find two matrices $A$ and $B$ that can be used to approximate $H$. Concretely, the update matrix can be factorized using A and B:

$$\tilde{H} = A * B$$

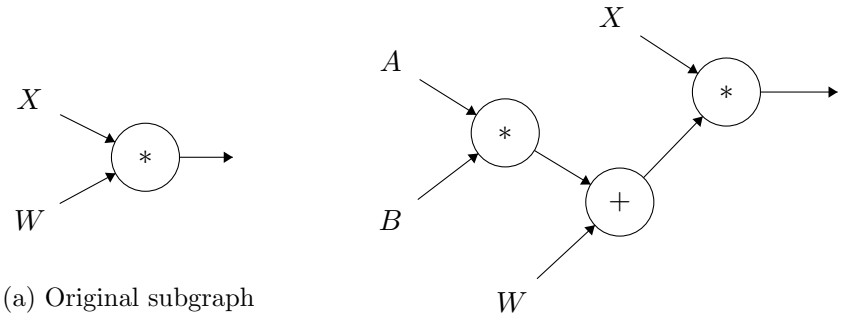
The dimensions of $A$ and $B$ are chosen accordingly. If $H$ is a $d_1 \times d_2$ matrix, then $A$ has dimensions $d_1 \times k$ and $B$ has dimensions $k \times d_2$. By doing this, we effectively limit the rank of $\tilde{H}$ to be at most $k$. This makes $\tilde{H}$ a *low-rank approximation* of $H$. Thus, $\tilde{H}$ contains less information than $H$, the trade-off being that it is easier to encode.

Both the compression factor and quality of the compression depend on the size of $k$. If $k$ is too small, not much information is retained and nothing complex can be learned. Conversely, only a weak compression can be achieved if $k$ is too large. The matrices $A$ and $B$ are $k/d_2$ and $k/d_1$ the size of $H$. What compression factor is viable depends on the task to be learned but in the experiments for this thesis a factor of 5 to 10 usually seemed feasible.

An important property of this method is that the low rank of the updates does not force the rank of $W$ to be low as well. Because users create update matrices which have different linearly dependent rows, the final weight matrix $W$ can still have a full rank. This is also the case because there are multiple rounds of communication.

Up until this point, this technique sounds like it might be a sketched update method. This would be true if $H$ is computed in the normal way and $A, B$ are chosen afterwards to approximate the selected $H$ as well as possible. However, if we know that $H$ needs to be factorized into two smaller matrices, then it makes sense to already take this information into account during the optimization process. Instead of just approximating $H$ afterwards, we directly try to find the $A$ and $B$ that optimize our loss function.

To implement this, computational graphs come in handy. In the computational graph that encodes our neural network, we replace each occurrence of $W$ with $W + A * B$. The matrices $A, B$ have to be declared as variables, while $W$ is now marked as a constant.




(a) Original subgraph

(b) Subgraph for a decomposed update

Figure 12: Each subgraph containing a weight matrix $W$ needs to be adapted

The computational graph allows us to easily partially differentiate with respect to the elements of $A$ and $B$. These two matrices can then be optimized with the same gradient descent process that could also be used to optimize W. The local optimization process performs several steps of SGD to keep improving $A$ and $B$. By doing all of this, we perform a structured update method where the enforced constraint is a low rank of $W$.

This also explains why we do not make use of standard decomposition algorithms like *singular value decomposition* [34]. While they might also find a sensible decomposition, they would be applied one step too late. If we directly optimize based on the fact that we want to decompose, then we can produce much more meaningful updates.

Like all weights matrices, $A$ and $B$ are randomly initialized and then optimized iteratively. An alternative to the method just presented is keeping either $A$ or $B$ fixed after initialization and only optimizing the other one. If clients use a local random seed to generate the fixed matrix, then it is enough to transfer the random seed since it fully encodes how the fixed matrix looks like. The other matrix is then optimized using the same computational graph idea. The advantage of that method is that the communication cost is reduced by around half, depending on the size of $d_1$ and $d_2$. The disadvantage is at hand: The decomposition found is a worse approximation than the one found by optimizing both matrices.

Konečnỳ et al. [50] performed a number of experiments where $A$ or $B$ were kept fixed. They found that keeping $A$ fixed generally worked much better than keeping $B$ fixed. No formal reasoning for this was given but on an intuitive level it can be explained like this: $B$ acts as a projection matrix. It maps the data to a smaller, $k$-dimensional space. $A$ is a reconstruction matrix. Given the $k$-dimensional encoding, it tries to reconstruct as much information as possible. If $B$ is fixed and randomly chosen, then a projection that can not be reconstructed well could be found. Choosing the reconstruction randomly is not as bad because this part touches the data at

a later part. The projection matrix can be optimized to prepare the data well for the random reconstruction.

## 4.4 Updates for RProp

All of the methods discussed so far focused on the client's role in the optimization process. A different way to tackle the problem is to think about the optimization algorithm used on the server. For Federated Learning, we are especially interested in optimization algorithms that allow us to encode the required information very efficiently.

The RProp optimization algorithm introduced earlier can be highly useful for this. It only uses the sign of gradient and ignores its magnitude. The sign of a weight update can be encoded using a single bit as opposed to 32 or 64 bits for encoding the entire gradient as a float. Using RProp with Federated Learning is simple. Each client computes its local gradient but only sends one bit per weight to the server: 0 for a negative sign, 1 for a nonnegative one.

To update the step size for a weight, the server then uses the more common sign. The third case in Equation 4 is discarded because one bit per client and weight does not allow us to conclude if 0 is the most common sign. This is not a problem because the chance of perfectly landing in an optimum is virtually non-existing anyways. This algorithm is also slighted different to the standard RProp in that the gradient magnitude is already ignored before computing the average update. Instead of first averaging and then taking the sign, we first compute the individual signs here and then select the most common one. Still, it closely resembles the original algorithm and behaves similarly if most users produce updates of similar size.

RProp allows for a compression factor equivalent to probabilistic binarization. However, no crucial information is lost at all because RProp was designed to purely work based on gradient signs. This makes RProp a promising optimization algorithm for Federated Learning. It can be motivated based on the fact that it is well suited for distributed optimization as well as based on its usefulness for compression.

## 4.5 Compression-Sampling Tradeoff

By sampling a subset of users in each iteration, we only estimate the update that should be applied to the model. This estimate has a smaller variance if more users are sampled. On the other hand, adding lossy compression makes it more difficult to compute a good estimate because the individual updates from users now contain less information.

To estimate updates well, with little communication per user, a tradeoff between the number of users and the compression factor has to be found [63]. Adding additional compression reduces the update quality but this can be

offset to some degree by sampling more users in each iteration. This allows us to effectively distribute the required communication across many users, making the protocol more efficient for individuals.

Figure 13 shows simulation results for this. A model is trained using the *frecency* simulation dataset [98], introduced in Section 7.3. Three different simulations are shown. The first simulation samples 200 users in each iteration and uses no compression at all. The other two simulations sample more users but use a sparse mask compression factor so that all simulations require the same amount of bandwidth in each iteration.

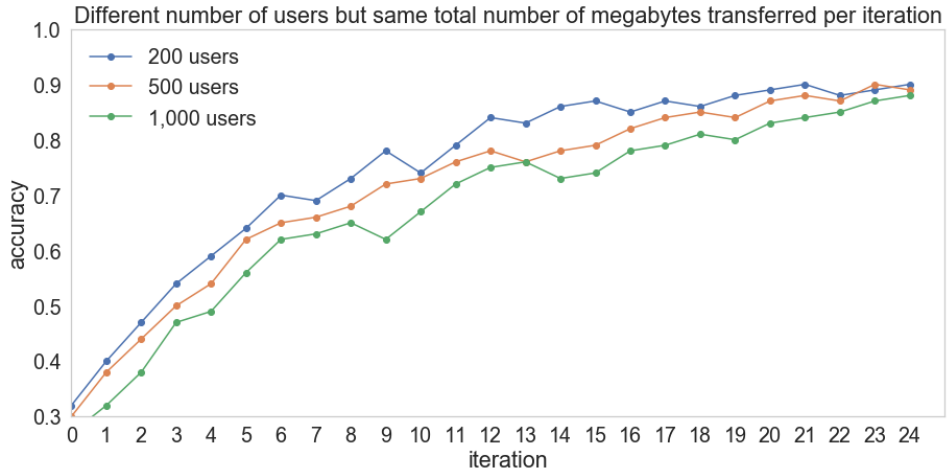



Figure 13: Adding more users allows us to use a stronger compression factor

It can be observed that the methods with compression converge only slightly slower, while reducing the communication per user quite strongly. Increasing the number of users is very helpful here. The convergence speed in a different simulation which uses the same compression methods but always samples 200 users is up to three times slower.

# 5 Privacy

## 5.1 Motivation

One primary motivation for introducing Federated Learning were privacy concerns with the way machine learning is conventionally done. Federated Learning seems like a major improvement in this regard since the data does not leave the users' devices anymore. However, it still remains to be discussed if it is possible to reconstruct data based on the updates that are sent to the server.

Since neural networks are universal function approximators, they are able to approximate a function that acts as a look-up table to all the data [20]. Neural networks with too many neurons typically memorize parts of the training data instead of learning more general pattern [35]. We have to assume that an adversary could intercept all messages and read the model weights. This adversarial actor could analyze the weights to figure out information about individuals.

Weights of neural networks have the reputation of being incredibly hard to analyze [31]. Still, there has been some work in this area. Fredrikson et al. [29] analyzed the weights of a facial recognition model. They were able to reconstruct some of the faces that were used in the training of the model.

While all of this remains incredibly difficult to do and there have been few successful such attempts, they show that attack vectors to Federated Learning do exist. When looking at the techniques introduced so far, we are not able to quantify how difficult it really is to figure anything out about the data of individuals. Having no formal guarantees for privacy is a major problem. Historically, there have been many cases where people tried to anonymize data to ensure the privacy of individuals. When there were no formal guarantees, this anonymization often looked solid but was later broken.

One of these cases was related to the *Netflix Prize* [9]. Netflix published a dataset that contained information about users and which movies they liked. The dataset was meant to be used in a competition to improve the Netflix recommender system. Obvious personal identifiers, such as names and user IDs, were removed from this dataset. Many users however also published their movie reviews on IMDb. By joining the Netflix and IMDb datasets, researchers were able to deanonymize parts of the Netflix dataset [67].

A similar case occurred in the 1990s, when a government agency in Massachusetts published a dataset about hospital visits of its employees [8]. This dataset was meant for research purposes. Again, personal identifiers, such as names and social security numbers, were removed. However, the dataset was still deanonymized in parts by joining it with a voter roll dataset. Datasets with information about people registered for voting can be bought legally in the US [108]. Since both datasets shared some fields, it was possible to join

them.

A few personal attributes make it surprisingly easy to identify people. In the US, 87% of people can be uniquely identified by gender, birth date and postal code alone [86]. In the case of the health-related dataset, the information was enough to identify the governor of Massachusetts in the dataset.

All these stories share a common pattern. When the data was published, steps for anonymization were taken but there were no formal guarantees. Later, someone found auxiliary information that could be used to deanonymize parts of the dataset. These cases show that formalization is desirable when it comes to privacy. While Federated Learning looks like it is much better for privacy, we want to have guarantees. These guarantees should even hold if attackers have access to additional information that we have no knowledge about.

## 5.2 Differential Privacy

To deal with the problems outlined in the previous section, we want to formalize what privacy means. *Differential Privacy* is a mathematical field that tries to do this using a stochastic framework [26]. It allows us to quantify how much certain algorithms respect privacy. We do not consider algorithms to be either privacy-preserving or to be bad for privacy. Instead of this binary view, we try to describe how difficult it is to make conclusions about the data of individuals.

On a high level, the goal of Differential Privacy is to compute accurate statistics on the entirety of users while not knowing anything about individuals with high confidence. For example, we might want to compute the mean value across many users without knowing the exact values of individuals. In the case of machine learning, we want to fit a model without knowing details about the data of individuals.

To describe this formally, two datasets $D_1$ and $D_2$ are considered. These datasets are *adjacent* to each other. The definition of adjacent can differ across applications but it usually means that the two datasets are identical except for one data point which is missing in one of the two datasets.

A statistical query $Q$ is then executed on both datasets. This query could, for example, compute the mean or fit a statistical model. It usually involves randomness. Even for queries that are not random by default, such as computing the mean, adding randomness helps to improve privacy guarantees.

**Definition 1.** *A query $Q$ is considered to be $\epsilon$-differentially-private if for all adjacent datasets $D_1, D_2$ and for every possible subset $R$ of results of the query, the following formula holds:*

$$\mathbb{P}[Q(D_1) \in R] \leq e^{\epsilon} * \mathbb{P}[Q(D_2) \in R]$$

That is, adding an additional data point to a dataset must not substantially change the result of the query. The formula is expressed using probabilities to account for the randomness in $Q$. If we guess the results of the query to be in a set of possible values $R$, then adding one data point should not change the probability of being correct by more than $e^{\epsilon}$. If this is not the case, then adding the new data point is bad for privacy because it is noticeable whether the data point was used by the query or not.

The value $\epsilon$ is called the *level of Differential Privacy*. The definition above captures what we intuitively understand as privacy. It should be very hard to figure out whether an individual contributed data, and much less so what their data looks like.

To make algorithms fit into the framework of Differential Privacy, randomization strategies are used [26, 28]. Instead of having users report their true data, they only share a version where random noise was added on top. In the case of discrete data where it is hard to add a little bit of noise, users could lie with some given probability [28]. By doing this, the entity collecting the data cannot make confident conclusions about individuals anymore. However, it is possible to estimate the overall random noise well when enough users are surveyed.

In the following subsection, we use a slight variation of the earlier definition.

**Definition 2.** *A query $Q$ is called $(\epsilon, \delta)$-differentially-private if for all possible subsets of results $R$ and all adjacent datasets $D_1, D_2$, the following holds:*

$$\mathbb{P}[Q(D_1) \in R] \leq e^{\epsilon} * \mathbb{P}[Q(D_2) \in R] + \delta$$

In contrast to the first definition, an additional $\delta$ is added. This allows for a probability $\delta$ of directly breaking Differential Privacy. We want to keep both $\epsilon$ and $\delta$ small to ensure good privacy.

To show that an algorithm conforms to some form of Differential Privacy, the concept of *sensitivity* is often used.

**Definition 3.** *The sensitivity $S(Q)$ of a query $Q$ describes by how much the result can differ if the query is executed on two adjacent datasets:*

$$S(Q) = \max_{D_1, D_2} ||Q(D_1) - Q(D_2)||_2$$

*where $|| \cdot ||_2$ is the $L_2$-norm. The sensitivity should be low, or, even better, bounded by a constant.*

## 5.3 Differentially-Private Federated Learning

In one iteration of Federated Learning, data of a specific user is either used entirely or not used at all. To account for this, a different definition of

adjacent datasets is required. We now consider two datasets $D_1, D_2$ to be adjacent if they only differ in the data of one single user. That is, the datasets are the same except that one of the datasets contains data from a user that is not present in the other dataset.

The motivation behind this is that is should be difficult to tell whether a user participated in training the model. The model should not differ much by adding a new user. If this is the case, then the model can not have memorized the data of that specific user.

To design a Federated Learning algorithm which can be proven to be $(\epsilon, \delta)$-differentially-private, we build on ideas from Abadi et al. [3]. They introduce a noisy version of SGD and present the following theorem:

**Theorem 3.** *A learning algorithm based on SGD computes a gradient estimate in each of $T$ iterations. The data used to compute the estimate is sampled using a probability $q$. The sensitivity of the estimate is bounded by a constant $d$ and noise sampled from $N(0, \sigma^2 d^2)$ is added to the estimate in each iteration. To compute the weights of the next iteration, the estimate is subtracted from the current weights.*

*Then, constants $c_1, c_2$ exist, so that the algorithm is $(\epsilon, \delta)$-differentially-private for any $\epsilon < c_1 q^2 T$ and $\delta > 0$ if noise is added using:*

$$\sigma \leq c_2 \frac{q\sqrt{T \log(1/\delta)}}{\epsilon}$$

McMahan et al. [64] then adapted Federated Learning to make it fit into the framework above. The remainder of this section explains how this can be done.

First of all, users are sampled with a probability $q$. This means that the number of sampled users can differ across iterations. The underlying proof of the theorem requires that the data was sampled independently from each other, so we have to sample with a probability $q$ instead of always sampling $K$ users. This also means that strategies like stratified or cluster sampling cannot be used as they introduce a bias. They might sample one user with a very high probability, making it hard to ensure this person's privacy.

To bound the sensitivity of the gradient estimate, we want to bound the size that individual updates $H_i$ can have by $s$. This is implemented by checking the $L_2$-norm of $H_i$ and scaling it down if necessary:

$$\tilde{H_i} = \begin{cases} H_i & \text{if } ||H_i||_2 \leq s \\ H_i * \frac{s}{||H_i||_2} & \text{otherwise} \end{cases}$$

An alternative way of limiting the $L_2$-norm for neural networks is to enforce different limits in the various layers. If $s_i$ is the limit of the $L_2$-norm

in the $i$-th layer and there are $l$ layers, then the overall limit is given by

$$s = \sqrt{\sum_{i=1}^{l} s_i^2}$$

since the sum in the square root can be expanded to the sum of squares of the individual components of the update vector. The bounds of different layers can be tuned to improve the learning process.

The RProp variation idea from Section 4.4 makes it easy to enforce a low upper bound on the update. Since all components of the update are either 1 or $-1$, the bound is already given by

$$||H_i||_2 = \sqrt{m}$$

where $m$ is the number of weights.

If the set of sampled users is denoted by $C$, then a simple way of estimating the gradient is

$$g(C) = \frac{\sum_{i \in C} n_i H_i}{\sum_{i \in C} n_i}$$

where the number of data points $n_i$ weights the importance of the user's update.

To show that this estimator has a bounded sensitivity, we first estimate $g(C)$ upwards. Let $N = \sum_{i \in C} n_i$ be the number of data points used in the current iteration:

$$\begin{aligned}
||g(C)||_2 &= ||\frac{\sum_{i \in C} n_i H_i}{\sum_{i \in C} n_i}||_2 \\
&= ||\sum_{i \in C} \frac{n_i}{N} H_i||_2 \\
&\leq \sum_{i \in C} ||\frac{n_i}{N} H_i||_2 \qquad (6) \\
&= \sum_{i \in C} |\frac{n_i}{N}|||H_i||_2 \\
&\leq \sum_{i \in C} |\frac{n_i}{N}|s \\
&= s
\end{aligned}$$

where we use the triangle inequality in step 6.

This allows us to bound the sensitivity of the gradient estimate:

$$\begin{aligned}
S(g) &= \max_{C,k} ||g(C) - g(C \cup k)||_2 \\
&\leq \max_{C,k} ||g(C)||_2 + || - g(C \cup k)||_2
\end{aligned}$$

$$
\begin{aligned}
&= \max_{C,k} ||g(C)||_2 + ||g(C \cup k)||_2 \\
&= \max_{C,k} 2s \\
&= 2s
\end{aligned}
$$

Since we know that the sensitivity of the gradient estimate is bounded, Theorem 3 can be applied. We thus add a sufficient amount of noise each iteration. In the implementation, an *accountant* object keeps track of the privacy loss over the iterations. In each iteration, the accountant checks what the level of Differential Privacy currently is. Once $\epsilon$ or $\delta$ pass the configured maximum level, the training is stopped, without applying the last aggregated update.

Some of the changes made to the algorithm are common regularization strategies. Gradient clipping is often used to deal with exploding gradients, which for example happen when the optimizer is in a very steep area [73, 35]. However, usually the final gradient estimate is clipped, while we clip the individual elements before computing the average here.

Adding noise is another popular regularization strategy [69, 5]. By adding some random noise, the model has a harder time memorizing data, which can combat overfitting. It has been shown that this can be equivalent to other forms of regularization, such as penalizing the size of the weights [11].

To ensure a good level of Differential Privacy, both of these methods might need to be used extensively. In this case, the regularization effect can become too strong, making learning much more difficult. Empirically, McMahan et al. [64] showed that they can reach the same level of accuracy with this algorithm. However, training takes roughly 60 times longer since the clipped gradients and the additional noise slow the convergence process down.

Another problem is choosing an appropriate level of Differential Privacy. It is not entirely clear what level can be considered a good choice in a given situation. This is a common problem with Differential Privacy in general [45, 55], independent of the application to Federated Learning.

Still, the fact that a comparable accuracy can be reached using this algorithm is interesting from a theoretical perspective. It shows that it is possible to train machine learning models in highly privacy-respecting ways. The data does not leave the device at all and we can prove that it is incredibly hard to analyze the abstract updates that are sent.

While there are no guarantees that these algorithms find good models, this is no different to most of machine learning. Most machine learning algorithms give no strong guarantees and are only evaluated empirically.

# 6 Personalization

## 6.1 Motivation

Personalizing models for individual users can be useful in many applications. One prominent example of this are recommender systems. Here, personal preferences of users can differ a lot, so having models incorporate additional user-specific context can boost performance [10, 4]. By slightly customizing models for individuals, we can try to match pattern in their data more closely.

From a theoretical perspective, users can generate data from different underlying distributions. By fitting just one model on all the data, it is only possible to detect pattern that exist in the data of many users. The central model can thus not be perfect for everyone. Personalization means that we adapt the model for each individual, to improve the performance on their data.

A naive approach of tackling this problem does not use Federated Learning at all. Instead, completely independent models are trained for each user, locally on their own computer. This solution generally works badly because individuals often do not have enough data to properly fit a model.

This section deals with methods between the two extremes of having one central, non-personalized model and having completely independent models. To this end, both approaches are combined: A central model is trained using Federated Learning. Because a lot of data is available for this model, it can have a good quality and will generalize well. Additionally, users take this model and also customize it locally to improve the performance even further.

Fine-tuning an already trained model is possible with much less data [71], so this is a more feasible approach. Compared to a standard Federated Learning system, this is only a small step engineering-wise since we are already training on the user's computer. To respect privacy, the customized model is not shared with the server. Information leaks in the collaborative training part can be controlled using Differential Privacy, as explained earlier.

To formally define the goals of personalization, we consider the overall loss across all users. If users customized the model, the loss is computed using that model. Otherwise, the central model is used. The first goal is that the loss should be smaller when using individually personalized models compared to using the central one.

Additionally, the quality of the central model should not be decreased. New users that do not have their own data yet will fall back to this model. Thus, it should work as well as when trained with a standard Federated Learning system. Making sure models work well for new users that have no data yet is known as handling the *cold start problem* in machine learning and recommender systems [53]. Personalization strategies should deal with

this problem.

The next two subsections describe two possible approaches to this problem. Since personalization is a new research area of Federated Learning that was just recently proposed [64], there is no existing literature describing these exact methods. Still, they built on existing ideas in machine learning and thus have some empirical evidence behind them. Finally, to evaluate them on one example dataset, simulation results are presented.

## 6.2 Transfer Learning

*Transfer Learning* is a common machine learning strategy for training models in situations where little data is available for the exact problem that needs to be solved [71, 87]. However, there is a lot of data available for a more general version of the problem. For example, one might want to train an object detection model that is able to recognize a certain kind of a traffic sign.

If there are not many training images of this exact traffic sign available, it can be hard to fit a good model. However, a lot of training data exists for more general object detection tasks, such as the ImageNet dataset [24]. These datasets have images of thousands of different kinds of objects, with traffic signs not necessarily being among them. Transfer Learning tries to make use of the larger dataset to be able to solve the actual problem of fitting a model on the small dataset.

Generally, this is a two-step process. First, a model is trained on the large dataset. For many well-known datasets, these models were already trained and are freely available online, for example for standard object detection tasks [103]. In the next step, we continue to train the same model but use the small dataset for this.

The motivation behind this technique is that many parts of the first model are reusable for the second problem. The training on the small dataset thus starts from a much better initial point. This also resembles more closely how humans learn [16]. When we learn to recognize a new object, we do not start from scratch but can learn more quickly because we already know how to recognize certain shapes and edges.

Sometimes, only parts of the model are fine-tuned [70]. During the training on the small dataset, the other layers are *frozen* and ignored by the optimization process. To freeze a layer, its weights are marked as constants in the computational graph. For example, convolutional layers are often only trained on the first dataset. This is because convolutional layers behave similarly in most object detection problems [89], e.g. they are doing general edge detection. The model generalizes better if only the final, fully-connected layers are fine-tuned in such cases.

Transfer Learning has been used successfully in many parts of modern computer vision [70] Many custom object detection models are based on

Transfer Learning as of 2018 [46]. Increasingly, natural language processing also makes use of Transfer Learning. By first learning on a large corpus of general text, it becomes easier to work with inputs that represent words [75, 13]. Models for machine translation or text summarization are based on this idea [7, 83].

The same idea can also be applied to Federated Learning. Training on the data of all users is the equivalent of training on the large, more general dataset here. We use this data to generate a good starting point for personalizing models locally. This is also a two-step process:

1. A central model is trained collaboratively by all users, using Federated Learning
2. Users individually fine-tune their model locally. Here, Transfer Learning techniques, such as freezing layers, are used

This technique can solve the general personalization problem, outlined in the previous section. Because the local training starts with an already well-trained model, it is easier to get a good custom model with little data. This approach can also handle the cold start problem. New users start by using the central model, which was trained by a lot of users.

The fundamental problem with this approach is that the central model can become outdated. As soon as users start fine-tuning their local model, they cannot collaborate to train the central model anymore. If the overall distribution of data changes over time, this could be problematic. For example, recommender systems might become outdated when new trends emerge.

## 6.3 Personalization Vector

Adapting the architecture of the model is an alternative approach to personalization in Federated Learning. This needs to be done in a way that allows users to personalize while also being able to continuously help train the central model. To this end, the model is given an additional input vector that differs across users. This personalization vector abstractly encodes user preferences and is not shared with the server at all. The values of the vector are learned, based on data.

In the beginning, only the central model is trained until it has sufficient quality. Then, we alternate between continuing to train the model parameters and optimizing the personalization vector. In every second iteration, partial derivatives of the loss with respect to the personalization vector are computed. The personalization vector is then locally updated by an optimizer that uses this gradient. By using computational graphs, it is straightforward to compute the partial derivatives with respect to the personalization vector. Over time, the personalization vector will reflect the user's preferences more closely.

In every other iteration, a vector of zeros is used as the additional input in place of the personalization vector. Only the weights of the model are then optimized. To do this, an update is computed and send to the server. The server uses these updates to improve the central model, and thus ensures that it does not become outdated. Even if users do not have their own personalization vector yet, they can rely on the central model. The cold start problem is thus addressed.

By introducing a personalization vector, we give the model the capability to store abstract information about the user. It can learn to remember information that allows itself to match pattern in the user's data more closely. Even if additional information is not useful, it could learn to ignore the personalization vector. The model with the personalization vector has thus the capacity to outperform the central model by itself because it has access to more information.

Just like the first approach, this idea builds on existing concepts. Alternating between optimizing different values is based on the *alternating least squares* algorithm that is commonly used for collaborative filtering [92]. Learning a vector that remembers the important part of the data seen so far is the basis of how recurrent neural networks process streams of information [35, 43].

Compared to the first approach, this method is much more flexible. All users can help train the central model indefinitely. The cost for this is an increased engineering effort since the protocol involved is more complicated to implement. The size of the personalization vector is also an additional hyperparameter that needs to be tuned.

## 6.4 Simulations

To empirically evaluate if these methods can work, they were implemented in a simulation. This simulation consists of three parts:

1. All users help to train a central model using Federated Learning
2. All users start to customize the model, using either the Transfer Learning idea or using an additional personalization vector
3. The way that users sample the data is slowly changed while the training itself still works in the same way as in the previous step

We report how the accuracy changes during these steps.

The *frecency* dataset [98] was used in this simulation. The goal is to predict which website a user wants to visit when they enter a certain search term. This is done based on features like how often and how recently they visited the candidate websites, whether they bookmarked them and so on. The dataset was created by encoding some assumptions such as how many websites users bookmark or how often they return to websites. To make sure

that individual users sample differently, they make use of different random seeds. A more thorough description of this data is given in Section 7.3.

During the training, new data is sampled in every iteration. Models are evaluated using the early stopping idea from Section 3.4. Sampled data is first used to compute an update and then used to evaluate the current model.

Figure 14 shows the results of training a model on this data using the Transfer Learning approach. Initially, everyone uses the central model, which improves over time. In iteration 12, the personalization stage begins. Users now fine-tune the model locally, while the central model is not trained anymore. This means that the accuracy of the central model essentially stays stable. Because of the randomness involved in evaluating the model using newly sampled training data, there are small fluctuations. The performance of the local models quickly increases because individual preferences can be matched much more closely now.

In iteration 19, the way that the data is sampled starts to change. There are now fewer bookmarks and users visited sites less recently. Since the central model is not updated anymore, its quality starts to degrade. The local models are not affected that much because users still update them. However, individuals also do not have enough data to fit a really good model anymore, leading to a small loss in accuracy.

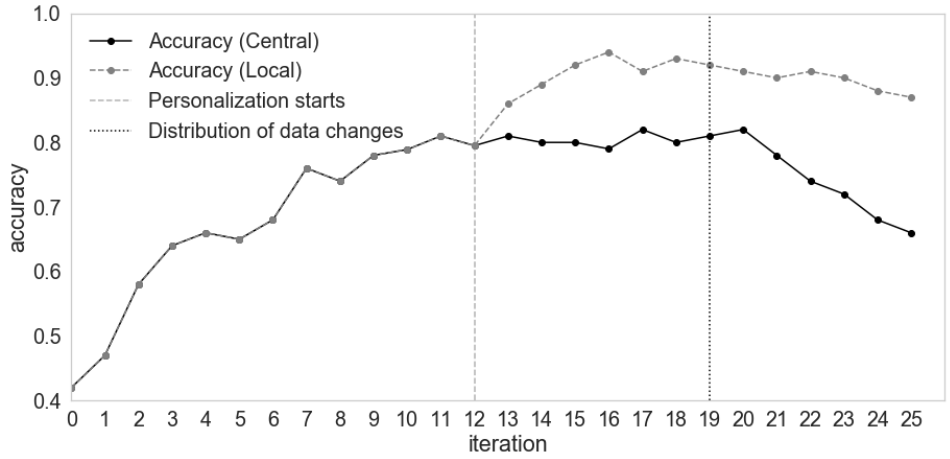



Figure 14: Personalization using Transfer Learning

Figure 15 shows the results of the same simulation when using a personalization vector. The central model now keeps improving continuously, even in the personalization phase. However, this also means that the local version improves more slowly since clients only spent part of their time optimizing the personalization vector.

When the underlying data starts to change in iteration 19, both, the

central model and the local vectors, continue to improve. This is because there is enough data to adapt the central model. The cold-start problem is handled much better by this approach since the central model has a high quality even when the way that data is generated slightly changes over time.

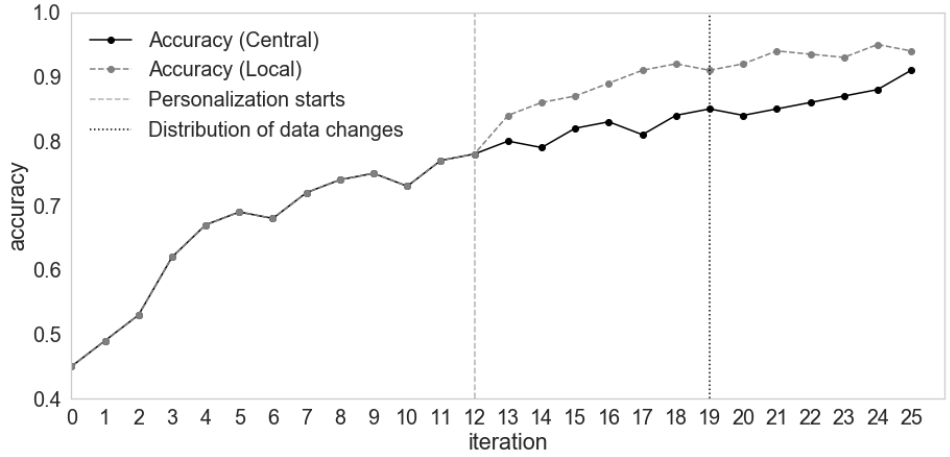



Figure 15: Personalization using a custom input vector

# 7 Implementation

This section describes an implementation of a Federated Learning system that was built for Firefox. The system was used to improve the history and bookmark search in the Firefox URL bar. 186,315 people participated in the training, while many others used the new model.

To the best knowledge of the author, this was the first large-scale Federated Learning system ever built outside of Google. The entire code of the project was open sourced and is freely available [96, 97, 98]. Many of the techniques introduced in this thesis were utilized in the process. The RProp optimizer is the centerpiece of the optimization process. The early stopping idea from Section 3.4 was used to validate the model during training.

To make it possible to develop the entire system during the work for this thesis, the trained model could not be too complex. This is why a model with fewer weights was trained. The compression techniques from Section 4 were thus not required.

## 7.1 Search in Firefox

The Firefox URL bar displays suggestions when typing a search query, as shown in Figure 16. Some of them are directly provided by a search engine. The others are generated by Firefox itself, for example based on the user's history, bookmarks or open tabs. Those suggestions are selected by a handcrafted algorithm that contains numbers that were not chosen in a data-driven process. To optimize them, Federated Learning can be used.

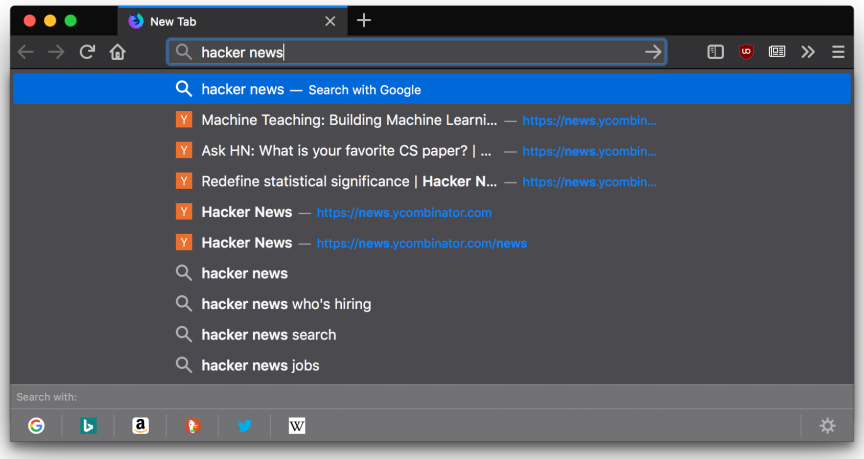



Figure 16: The Firefox URL with and five history and bookmark suggestions

Searching for history and bookmark entries in the Firefox URL bar is a

two-step process:

1. The search query is matched against the browser history and bookmarks. Matching is a binary decision. Pages either match the query or do not

2. The set of matched links is ranked based on the user's history

The Federated Learning system optimizes the ranking part of this process, while the matching part itself is not changed. Before diving into the current implementation, it is worth taking a step back to understand how ranking in machine learning works. This makes it easier to see how the current algorithm fits into a machine learning system. Fundamentally, there are three different approaches for learning a ranking algorithm [58, 18]:

1. *Pointwise ranking*: Each item is given separately to the model, which assigns a score to the item. The ranking is then determined by sorting all items using their respective scores. Essentially, this is a special type of a regression model since we are assigning a real number to every input

2. *Pairwise ranking*: The model learns to compare pairs of items. Its task is to decide which of the two items should be ranked higher. Intuitively, this method can be motivated by the fact that the learned comparison function could then be used by various sorting algorithms. In this approach, we treat the problem as a classification task since the model can only have two possible outputs

3. *Listwise ranking*: Instead of only working with individual items in each step, these methods try to operate on the entire list. The motivation behind this idea is that the evaluation metric can be optimized directly this way. In practice, this turns out to be fairly difficult because many evaluation metrics are not differentiable and the models need to work with many more inputs. Another difficulty is that the list could have an arbitrary length

All these approaches have different advantages and disadvantages. The existing ranking algorithm in Firefox is very similar to a pointwise ranking approach. Since this algorithm should be optimized using machine learning techniques, this gives us a clear set of techniques that could be useful in the optimization process.

The ranking of possible suggestions in the Firefox URL bar is performed using *frecency* [102], an algorithm that weights pages by how *frequently* and *recently* they were visited. To do this, a frecency score is assigned to each history entry and bookmark entry. After computing the score, it is cached. When searching, the matched results are then sorted using this score. The

remainder of this section introduces the existing frecency algorithm, while the next one explains how it could be improved.

Frecency does not only take frequency and recency into account but also uses other information, such as how the page was visited, if it is bookmarked or if there are any redirects. It does this by looking at the latest visits to the respective site. The value visit($v$) of one single visit $v$ is then defined by how recent that visit was, scaled by the type of visit:

$$\text{visit}(v) = \text{recency}(v) * \text{type}(v)$$

Frecency scores have to be cached in order to allow an efficient ranking while the user is typing. This means that the recency aspect has to be modelled using time buckets. Otherwise the score would change all the time and caching would not work. In the current Firefox implementation, there are five time buckets. With this approach, the recency score only changes when a visit changes time buckets:

$$\text{recency}(v) = \begin{cases} 100 & \text{if visited in the past 4 days} \\ 70 & \text{if visited in the past 14 days} \\ 50 & \text{if visited in the past 31 days} \\ 30 & \text{if visited in the past 90 days} \\ 10 & \text{otherwise} \end{cases}$$

Sites can be visited in many different ways. If the user typed the entire link themselves or if it was a bookmarked link, we want to weight that differently to visiting a page by clicking a link. Other visit types, like some types of redirects, should not be worth any score at all. This is implemented by scaling the recency score with a type weight:

$$\text{type}(v) = \begin{cases} 1.2 & \text{if normal visit} \\ 2 & \text{if link was typed out} \\ 1.4 & \text{if link is bookmarked} \\ 0 & \text{otherwise} \end{cases}$$

Now that a score can be assigned to every visit, the full points of a page could be determined by summing up the scores of each visit to that page. This approach has several disadvantages. For one, it would scale badly because some pages are visited a lot. Additionally, user preferences change over time and we might want to decrease the points in some situations.

Instead, we compute the average score of the last 10 visits. This score is then scaled by the total number of visits. The full frecency score can now be efficiently computed and changes in user behavior can be reflected fairly quickly. Let $S_x$ be the set of all visits to page $x$, and let $T_x$ be the set of the 10 most recent of these. The set $T_x$ contains fewer than 10 elements if

there were fewer visits to the respective page. The full frecency score is then given by:

$$\text{frecency}(x) = \frac{|S_x|}{|T_x|} * \sum_{v \in T_x} \text{visit}(v)$$

Note that this is a simplified version of the algorithm. There is some additional logic for special cases, such as typing out bookmarks or different kinds of redirects [102]. The description here only shows the essence of the algorithm in a mathematical form.

## 7.2 Optimization

While frecency has been working fairly well in Firefox, the weights in the algorithm were not decided on in a data-driven way. Essentially, they are similar to magic numbers in software engineering [61]. It is hard to understand why these exact numbers should be used. Even worse, there is no evidence that they are optimal. Maybe different time buckets or different weights would lead to much better results.

The Federated Learning implementation replaces these constants by variables that are optimized for. This is done for all numbers in the previous section, except for two:

- number of considered visits (10): If this number increases too much, it would be more expensive to compute frecency scores. The current value represents a good trade-off between performance and using a sufficient amount of information
- number of time buckets (5): Optimizing this would be hard with an approach based on gradient descent since this value affects how many variables there are. In the current implementation there was also no easy way of changing this

There are 22 remaining weights in the full algorithm that are optimized using Federated Learning. By doing this, more optimal values can be found, or it can at least be confirmed that the current ones were already chosen very well. It is also a safe way of experimenting with the Firefox URL bar. The optimization process starts off from the current set of values and then tries to improve them from there.

This process is based on the users' interactions with the URL bar. They are shown a set of suggestions that were ranked using our model. If they do not choose the top one, our model can use this feedback as a signal that it needs to change the weights to improve the rank of the selected item. Even if the top item was chosen, we can teach our model to be more confident in this decision.

Using Federated Learning for this makes it straightforward to use the underlying data in the optimization process. Alternative solutions that send the browser history to a server would be too privacy-invasive.

To describe the optimization goal formally, a loss function, which is used to evaluate how well the model works, needs to be defined. To this end, an adapted form of the *SVM loss* [91] is used. A similar form of the SVM loss has been used for pairwise ranking before [17]. This idea was transferred to pointwise ranking for this implementation. In our pointwise ranking setting, the loss function takes a set of items with their assigned score and the index of the selected item. The optimization goal is that the selected item should have the highest score.

But even if that was the case, our model might not have been too confident in that decision. One example for this is the selected item having a score of 100 and the second item having a score of 99.9. The model made the correct prediction, but only barely so. To make sure it does a good job in similar cases, we need to provide a signal to the model which shows that it can still improve.

If the URL bar displayed suggestions for pages $x_1, \ldots, x_n$ in that order and suggestion $x_i$ was chosen, then the SVM loss for ranking is given by:

$$\text{loss}((x_1, \ldots, x_n), i) = \sum_{j \neq i} \max(0, f(x_j) + \Delta - f(x_i))$$

where $f(x_j)$ is the score assigned to page $x_j$ by the pointwise ranking model.

We iterate over all suggestions that were not chosen and check that their score was smaller than the one of the selected page by at least a margin of $\Delta$. If not, an error is added. The full loss should be minimized. The margin $\Delta$ is a hyperparameter that needs to be decided on before the optimization process starts.

Figure 17 shows a visualization of this loss function. Each bar represents a possible suggestion, with the selected one being shown in black. The y-axis displays how many points the model assigned to the respective suggestion. The hatched areas show the SVM loss. Everything above the selected suggestion and below it by a margin of $\Delta$ adds to the full loss. Even though the selected suggestion had the second highest score, four suggestions contribute to the penalty.

Every time a user performs a history or bookmark search in the URL bar, the SVM loss is computed based on that search. To compute an update, we then try to move the weights a little bit into a direction where this loss is minimized. The update corresponds to the gradient of the SVM loss with respect to the weights in the frecency algorithm that we optimize for.

Gradients can be computed elegantly using computational graphs. By using machine learning libraries, the function that we want to compute needs to be constructed. Afterwards, automatic differentiation techniques can be

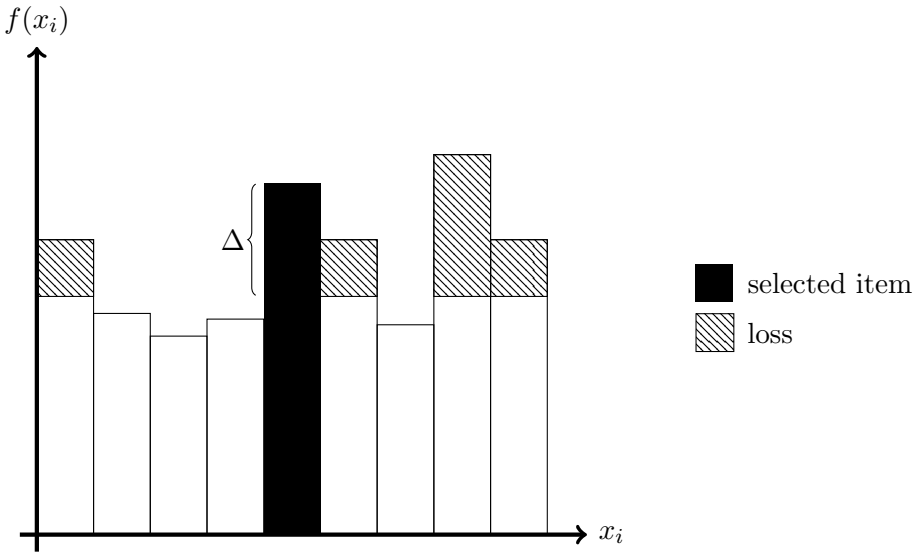



Figure 17: A visualization of the SVM loss when adapted for ranking

used to obtain the gradient. The initial prototyping was based on this idea. The major advantage is that it is very easy to change the model architecture.

The current frecency implementation in Firefox, however, is written in C++, while the client-side part of this experiment works using JavaScript. To launch the experiment, the Firefox *SHIELD* mechanism was used [104]. SHIELD allows directly sending new code to Firefox clients, without having to go through major version releases which only happen roughly every six weeks. To be able to do this, SHIELD experiments have to be written in JavaScript and can only make very limited use of C++ components.

This made it hard to add a computational graph to the existing C++ frecency module. Reimplementing the full algorithm in JavaScript seemed like a bad idea. Performance-wise there would be a huge penalty and it is hard to reconstruct the way the current implementation handles all the edge cases.

Instead, a simple finite-difference technique was used. To compute the gradient estimate of a function $g$ at the point $x$, we check the difference of values close to that point, given by a very small $\epsilon$:

$$g'(x) \approx \frac{g(x+\epsilon) - g(x-\epsilon)}{2 * \epsilon}$$

This formula is very close to the definition of derivatives. To compute the gradient of a multivariate function, this process is then performed by iterating through all dimensions independently. In each dimension, the value is changed by $\epsilon$ in the two directions, while all other values stay constant. The resulting vector is our gradient estimate.

This method is both easy to understand and implement. It is simple to change the frecency weights from JavaScript in our experiment without changing the actual algorithm. For large models there is a performance penalty since $g$ needs to be evaluated two times for every dimension. In $m$ dimensions, $\mathcal{O}(m)$ function evaluations are required, as opposed to $\mathcal{O}(1)$ for computational graphs. But since we only work in 22 dimensions here, this is not a major problem. However, numerical stability can suffer when using this method with a very small $\epsilon$.

The finite-difference method also allows us to essentially treat frecency as a black box. Our model does not need to know about all edge cases. It is sufficient for the model to see how different decisions affect the output and the loss.

The client-side part described so far is part of Firefox itself. This part of the implementation observes how users are interacting with the URL bar and retrieves all necessary information to compute the gradient. That update and some statistics about how well the model is doing are then sent to a Mozilla server. This works by using Mozilla's telemetry [99], a system designed to send information back to servers. It is a well-designed system with clear rules about what can be collected. There is a lot of infrastructure around using it and dealing with the data on the server.

On the server-side, all messages sent by clients are stored in a Parquet [95] data store. This is a special column-oriented database [1]. In column-oriented data stores, each column is stored in a contiguous sequence of memory. This is in contrast to row-oriented databases, most SQL systems, where data is stored by rows. Column-oriented databases provide major performance improvements for big data applications because data in the same column can be aggregated more quickly. This is useful in our case as we are interested in computing the average update quickly at the end of each iteration.

The Spark [90] MapReduce system is used to periodically compute new updates. Every 30 minutes, a Spark job reads the new updates, averages them and performs the optimization step. This is done in a MapReduce fashion across a cluster of six to 10 nodes, depending on how many updates users sent. The average update is then given to an optimizer and applied to the model. The resulting model is published and fetched by clients. Clients retrieve the model directly after a new version was published and additionally when Firefox is first opened.

It is possible to compute the average update in an online fashion, without loading much data into memory [85]. The naive solution of computing the full sum and then dividing it by the number of updates works badly because the sum can be large. Numerical stability can thus be lost easily. Instead, the average can be computed by iteratively updating the average of the updates that were considered so far. Let $\overline{H_i}$ be the average after $i$ updates

were taken into account.

$$\overline{H_1} = H_1$$

The average when taking into account one additional update, can then be computed by:

$$\overline{H_{i+1}} = \frac{i}{i+1}\overline{H_i} + \frac{1}{i+1}H_{i+1}$$

The first part of the sum essentially scales the previous average back to the full previous sum. Afterwards, the new average is computed. Of course, this does not help with numerical stability, so the above formula can be transformed further:

$$\begin{aligned}\overline{H_{i+1}} &= \frac{i}{i+1}\overline{H_i} + \frac{1}{i+1}H_{i+1} \\ &= \overline{H_i} + \frac{1}{i+1}H_{i+1} - \frac{1}{i+1}\overline{H_i} \\ &= \overline{H_i} + \frac{H_{i+1} - \overline{H_i}}{i+1}\end{aligned}$$

The final formula gives us a recurrence relation that can be used to compute the next average based on the current one. It is more numerically stable because we avoid working with huge numbers. The new average is given by the old average in addition to a small changed based on the new element.

This streaming algorithm can be used to compute the next average update while data is coming in. For this thesis, it was important to store all updates persistently so that they could be analyzed later. If this is not necessary, then the updates can directly be discarded after the average was updated using this algorithm.

After the average update was computed, it is given to an optimizer. Since no data was directly collected, it is hard to tune the optimizer beforehand. Even values like the learning rate are hard to set since we have no information about the gradient magnitude. Trying out many different optimizers in production would take a substantial amount of time and could lead to a bad user experience. Directly collecting some data to decide on this beforehand conflicts with the goal of doing machine learning in a privacy-respecting way.

This problem was tackled in two ways. First of all, simulations were created which use a made-up dataset that should be similar to the one we expected to see in production [98]. This allowed simpler experimenting with different optimizers and helped with making early design decisions. It also made it possible to quickly iterate on ideas to see if they could work.

The second way of dealing with the fact that it is hard to set hyperparameters was using the RProp optimizer. Because it ignores the gradient magnitude, it is straightforward to make it work with different kinds of data

that users might generate. The learning rate is automatically adapted, so the initial one does not have to be perfect.

One additional advantage in our case was that RProp updates are very interpretable. RProp makes it easy to understand how weights can change in a given iteration. By setting the hyperparameters accordingly, all updates are at most 3. This means that the frecency weights cannot explode after one iteration because one gradient magnitude was too large. When looking at the existing weights, this still allows the optimization process to significantly change all weights in a dozen iterations. The initial learning rates were 2 for recency weights and 0.02 for type weights, the other hyperparameters $\alpha = 2, \beta = 0.6$.

After RProp produces an update, two additional constraints are enforced:

1. Weights have to be nonnegative. This means visiting a site cannot directly have a negative effect

2. The time buckets have to be sorted by the last day they take into account. In other words, the $(i + 1)$-th time bucket needs to contain older visits than the $i$-th time bucket. This is to ensure that the client-side frecency implementation continues to work

These essentially act as safeguards to make sure that user experience does not degrade too much if the optimization process fails. The constraints are enforced by trimming the update magnitude if necessary.

Additionally, the optimization process was designed in a way that allows it to deal with integer updates. Some of the weights, like the number of days for time buckets, have to be integers. To allow for discrete optimization, updates are computed with $\epsilon = 1$. The RProp optimizer then makes sure that the final updates are rounded respectively.

## 7.3 Simulations

To check if all the ideas presented above could work and to quickly iterate on them, a simulation [98] was created before developing the actual system for Firefox. In this simulation, the users and the server are represented by objects. To communicate with clients, the server simply calls methods of the respective objects. This makes it possible to simulate an entire Federated Learning optimization process in little time, without having to wait for network connections.

To tune the process of computing weight updates, both an approach based on computational graphs and the finite-difference method were implemented. This allowed evaluating whether the finite-difference method can lead to a similar accuracy as the analytical method based on computational graphs. The simulated server performs exactly the same steps as the web server that is queried by real Firefox users.

The only major part that differs between the simulation and the actual implementation is what data is used for training. Since no data should be collected, the simulation could not be based on real data from users. Instead, a mock dataset was created. This dataset was designed to resemble the data we expected users to generate.

Since there was no way of knowing how the data is actually distributed, several assumptions had to be made. How recently websites were visited was initially modelled using a uniform distribution over the past 180 days. Afterwards, the distribution was slightly skewed towards more recent visits. Websites are assumed to be visited by clicking a link 60% of the time, by typing out a link 20% of the time and by using a bookmark 20% of the time. While the numbers for typed out links and bookmarks seem high, they made it easier to guarantee that more diverse data was sampled by clients.

To decide on how frequently websites are visited, an exponential distribution with $\lambda = 7$ is used. This mean that there are many websites that are only visited few times and few websites that are visited a lot. For simplicity's sake, recency, type and frequency are assumed to be independent of each other. The number of websites that match a query is sampled from a normal distribution with $\mu = 4, \sigma^2 = 10$. To ensure that individual users generate different data, they fix different random seeds.

To model what suggestion a user clicks on, the existing frecency algorithm is used to compute a score for each suggestion. Random noise, sampled from a normal distribution with $\mu = 0, \sigma^2 = 30$, is then added to the score. The dataset assumes that the suggestion with the highest score is selected. By using the existing frecency algorithm with some noise, it is easy to see if the simulation finds useful weights, as they should be similar to the ones of the current algorithm. The initial weights of the model in the simulation are configured to differ substantially from the ones we expect to find.

To supervise the model's performance, the loss and accuracy are checked. In this case, accuracy refers to the portion of queries where the model assigned the highest score to the suggestion that the dataset also considered to be the most important one. The difference in absolute scores is not important as long as both model and dataset agree on which item the user clicks on. This metric differs to the ones used in the actual Firefox implementation because more information related to the user interface is available there.

It is worth noting that this dataset is likely to differ substantially from the data generated by real users. It is difficult to perfectly describe how the data looks like, without having seen any of it. Still, creating the dataset allowed for quick prototyping, which made it much easier to make many design decisions.

## 7.4 Study Design

The client-side part of the experiment was shipped to 25% of Firefox Beta, which corresponds to roughly 500,000 daily active users. Since it takes some time to roll out updates, only a part of the users was enrolled in the study before the optimization process was completed.

Users were split into three groups:

1. *treatment*: The full study was shipped to these users. They compute updates, send them to the server, and start using a new model every 30 minutes
2. *control*: This group is solely observatory. No behavior in the URL bar actually changes. We are just collecting statistics for comparison to treatment
3. *control-no-decay*: Firefox decays frecency scores over time. Our treatment group loses this effect because we are recomputing scores every 30 minutes. To check if the decay is actually useful, this group has no decay effect but uses the same original algorithm otherwise

60% of users were assigned to the treatment group and 20% to both control groups respectively.

The criteria for a successful experiment were defined before anything was rolled out to users. This was done to ensure that everything was done in a statistically sound way. Concretely, there were three goals:

1. Do not significantly decrease the quality of the existing Firefox URL bar
2. Successfully train a model using Federated Learning
3. Stretch goal: Improve the Firefox URL bar

Actually improving the quality of the ranking for users was only a stretch goal. The primary goal of the study was to see if it is possible to make the distributed optimization process work. Essentially this meant consistently decreasing the loss of the model. At the same time, the quality of the URL bar should not decrease. The reason for distinguishing between these is that our optimization goal could have been misaligned. It is possible to minimize some loss function without actually improving the experience for the user.

To measure the quality of history and bookmark suggestions in the URL bar, two metrics were used. First, the number of typed characters that a user types before selecting a result should be minimized. Users should have to type as few characters as possible to find what they are looking for. Secondly, the rank of the suggestion that is selected is considered. The item that is selected should be as far on top of the list as possible. The top most item has a rank of 0. Both of these metrics should thus be minimized.

If the quality of any of these two metrics increases, the stretch goal is considered to be reached. Prior to the study, it was unclear if both metrics could be improved. One theory for this was that maybe users always type a similar number of characters before choosing one of the suggestions. The alternative could also be possible: Users always type until the first suggestion displayed is the one they were looking for. For this reason, only one of the two metrics needs to be improved. The first goal meant that both metrics should not get significantly worse.

An important part of designing studies is a power analysis [25, 27]. It tries to answer the question of how many people are required to get statistically significant results in an experiment. If too few people are enrolled, the results will contain too much random noise to rely on them. If a lot of people are enrolled, we can be confident in the results but the cost of the study will be much higher.

In the case of Firefox, this cost consists of two factors. For one, if our study enrolls most Firefox users, we would block other studies that want to experiment with changes in the URL bar. Another reason is that the experiment might break parts of Firefox. If this happens, it should not affect unnecessarily many people.

For this reason, a power analysis was performed to decide on the sample sizes for treatment and both control groups. Concretely, this analysis consisted of two parts:

1. How many users do we need to have enough data to train a model? (relevant for treatment)

2. How many users do we need to show certain effects confidently? (relevant for treatment and control)

The first part was answered using simulations [98]. By using an adapted form of the simulation that was used to decide on optimization hyperparameters, it was possible to get some idea on how many users were needed. Existing statistics from Mozilla about search queries were helpful for this, as they showed how many history and bookmark searches users generally perform every day [100].

The second part of the power analysis was tackled using classical hypothesis testing. There was no prior data on the number of typed characters, so no power analysis was possible for this metric. To analyze the rank of the selected item, the *Mann-Whitney-U test* [60, 62] was used since the data was not following a distribution that allows for parametric tests. The Mann-Whitney-U test is non-parametric, which means that it does not make any assumptions about the underlying distribution of the data.

These tests concluded that about 60,000 users per group were necessary to show sufficient effects. Treatment required roughly 200,000 daily searches to get enough data for the model. This analysis turned out to be helpful

since it showed that the control groups could be smaller than the treatment group.

## 7.5 Analyzing the Results

Over the course of the experiment, 723,581 users were enrolled in the study. The model was fetched 58,399,063 times from the server. 360,518 users participated in sending updates and evaluation data to the server, accounting for a total of 5,748,814 messages. The optimization phase of the experiment consisted of 137 iterations of 30 minutes each, or just under three days. In this phase, 186,315 users sent pings to help in the training process.

A separate phase of purely evaluating the model was started afterwards and took a total of 10 days. In this phase, 306,200 users send 3,674,063 pings, which included statistics detailing how well the model worked for them. Since all these users were assigned to treatment or control groups, the new model can be compared well to the old one that was used by the control groups. Some users were enrolled but did not help with optimization or evaluation because they performed no history and bookmark searches.

During the optimization process, the loss of the model was supervised to check how well the training was going. This was done by using the ideas from Section 3.4. Instead of additionally splitting users in treatment into training and validation sets, their data was first used to compute the loss and only afterwards used to compute the next model.

Figure 18 shows how the loss changed over time, across all three study variations. The loss of the treatment group goes down continuously. This shows that the optimization process generally worked. After 40 iterations, less than one day of optimization, the loss of the treatment group is always below the loss of the control groups. The second goal of the study was thus reached.

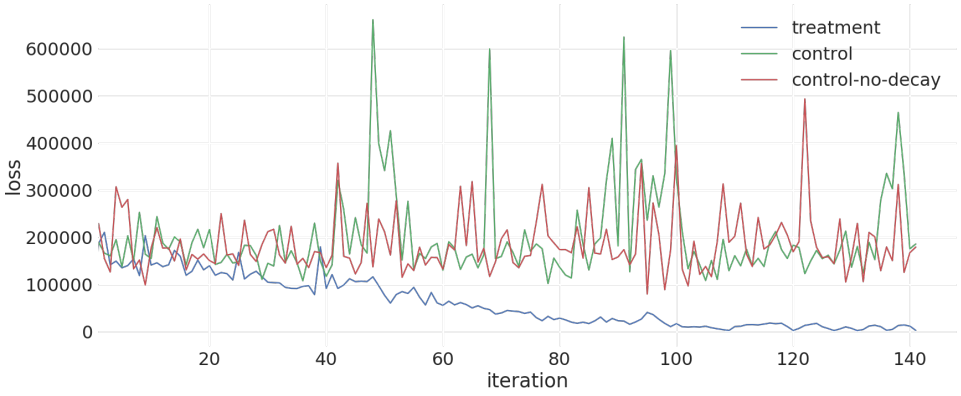



Figure 18: The reported loss over time

Since only a small portion of users reports back in a given iteration,

the loss evaluation data can be very noisy. If the model of an iteration is performing worse than the one of the previous iteration, it could be purely because not enough users reported back, which lead to inaccurate estimates. The loss is also sensible to outliers. It can grow very large just because a few users did not select a suggestion with a large frecency score. While it is not desirable to outright ignore outliers, a less noisy plot can be helpful for interpreting the results of the first few iterations.

Figure 19 shows a plot of a smooth version of the loss. This is generated by plotting the average loss of the last five iterations for every iteration. Now it is more clearly visible that the treatment group improved over the other two groups from an early point on.

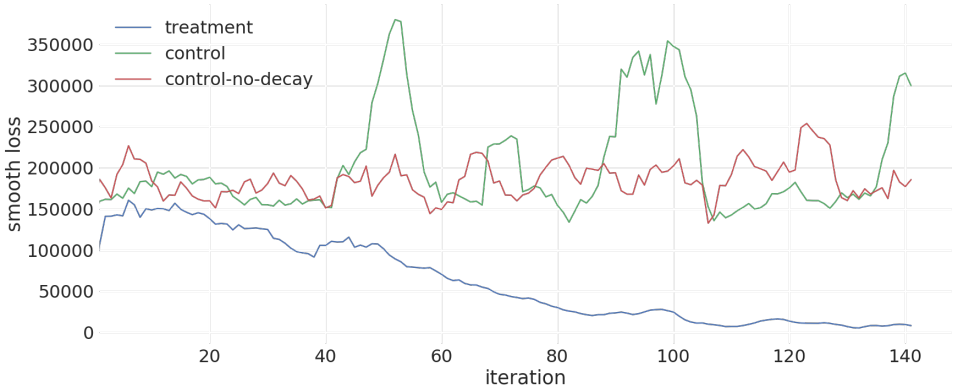



Figure 19: A smooth version of the reported loss over time

The Federated Learning protocol used in this experiment slightly deviated from the one introduced in Section 1.3. Instead of sampling $K$ users per iteration, users were assigned to a study variation once. All users from the treatment group could then help train the model in every iteration. This made sense because fewer people had to be enrolled in the study this way.

To understand why loss estimates can be noisy, it is helpful to look at the number of pings over time. Figure 20 visualizes the number of different reports in the various iterations. The plot shows that Firefox Beta users are more active during certain parts of the day. Since Firefox Beta is strongly skewed towards users from Asia, this effect can be severe. The most active iterations have around four times more users than the ones with the least activity.

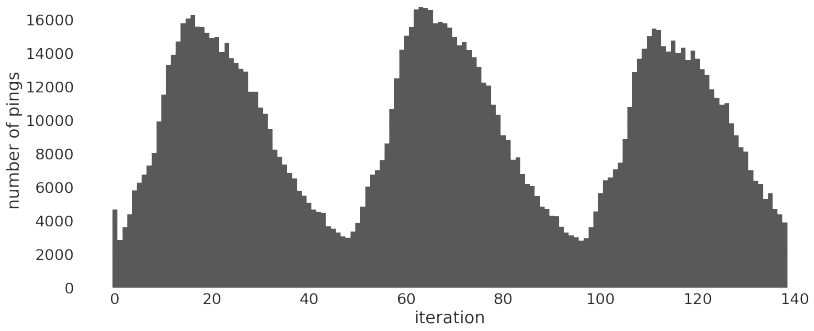



Figure 20: The number of pings sent by clients over time

In retrospective, the Federated Learning protocol used was too simple. It could be improved by dynamically determining the iteration length depending on how many updates were sent to the server so far. This way, there would be no iterations with very few updates. Furthermore, there could be more iterations during periods with many active users, allowing for a faster optimization process.

| | **mean number of typed characters** | **mean selected rank** |
|---|---|---|
| treatment | 3.6747 | 0.37435 |
| control | 4.26239 | 0.35350 |
| control-no-decay | 4.24125 | 0.35771 |

Figure 21: Results of the evaluation phase

After the optimization process ended, an evaluation phase began to determine how well the new model works. This is equivalent to the testing phase in machine learning. The model is evaluated on new data that was not used for training or validation.

Figure 21 shows the results. On average, users in the treatment group type about half a character less to find what they are looking for. This is a strong improvement over both control groups. However, users in the treatment group also choose suggestions that were ranked slightly worse.

To determine if these changes are statistically significant, hypothesis testing has to be used. Before analyzing the data, a significance level of 5% was set. Since data from both metrics does not follow any distributions that allow for parametric tests, the Mann-Whitney U test is used again. The null hypothesis states that there was no real difference between the study variations and that all data is actually sampled from the same underlying distribution. This null hypothesis is tested using the Mann-Whityney U test. The resulting p-values are displayed in Figure 22.

| groups compared | mean number of typed characters |
|---|---|
| treatment vs. control | $2.11938 * 10^{-105}$ |
| treatment vs. control-no-decay | $1.26503 * 10^{-79}$ |
| control vs. control-no-decay | 0.01019 |

| groups compared | mean selected rank |
|---|---|
| treatment vs. control | $2.26015 * 10^{-118}$ |
| treatment vs. control-no-decay | $1.02944 * 10^{-72}$ |
| control vs. control-no-decay | $1.41092 * 10^{-5}$ |

Figure 22: p-values when comparing the results of different study variations

P-values describe the probability of seeing an effect that is at least as strong as the one observed just by pure chance. Since six tests are performed, the significance level should be corrected so that it is less likely to incorrectly reject a null hypothesis just because many tests were done. The *Bonferroni correction* [66, 25] divides the significance level by the number of performed tests. In our case, we use 0.05/6 as the new significance level.

All of the changes observed in the treatment group are highly statistically significant since the p-values are well below the significance level. Because the metrics differ strongly between the treatment and control groups and due to the large sample size of 3,674,063 data points, we can reject the null hypothesis with high confidence when comparing treatment to the control groups. We can conclude that there is sufficient statistical evidence that the observed effects in the treatment group were not just by pure chance. The only null hypothesis we fail to reject is that there is a difference between control and control-no-decay when checking the number of typed characters. The p-value is above the Bonferroni-corrected significance level here.

From a user perspective, it is not clear if these changes improve the user experience. While users now have to type a good amount less, they also selected suggestions that were not on top of the list more often. One potential explanation for this could be that the items they were looking for are displayed earlier in the suggestion list. Since they spent less time typing, they might be willing to take the time to select an item that is not the top ranked one.

It is hard to determine purely based on these two metrics if this change is good, since it is not clear how their importance should be weighted. Instead, surveying users would be required to decide if goal 3 was met. But even if users are not satisfied with the new model, the Federated Learning system is still highly useful. Since the optimization process worked, one would only need to find a loss function that correlates more closely with what users want. We consider goal 1 to be reached since at least one of the metrics improved.

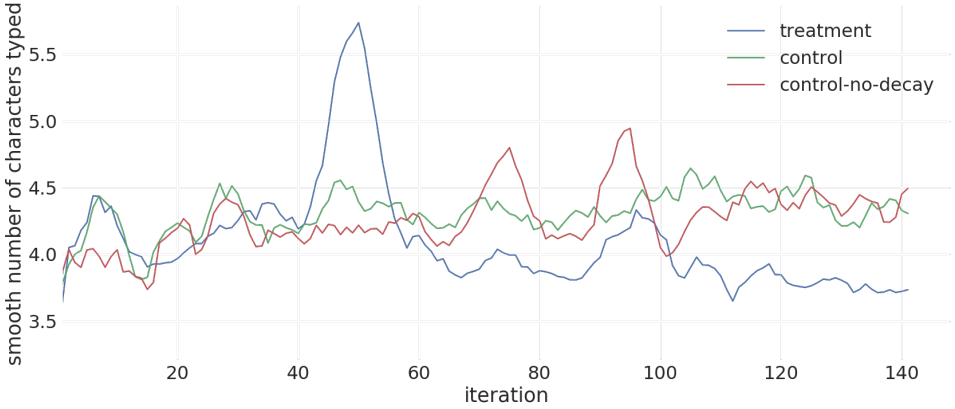



Figure 23: Number of typed characters over time (smoothed results)

Figure 23 and 24 show how these two metrics developed over time during the optimization process. In the early stages of the training, both metrics behaved similarly across all study variations. The number of typed characters in treatment significantly went down after iteration 60. When looking at the rank of the selected item, treatment generally performed worse than the control groups from about iteration 40 on. It is worth noting that these results are more noisy than the final evaluation data because there were much fewer users in any given iteration of the optimization process.

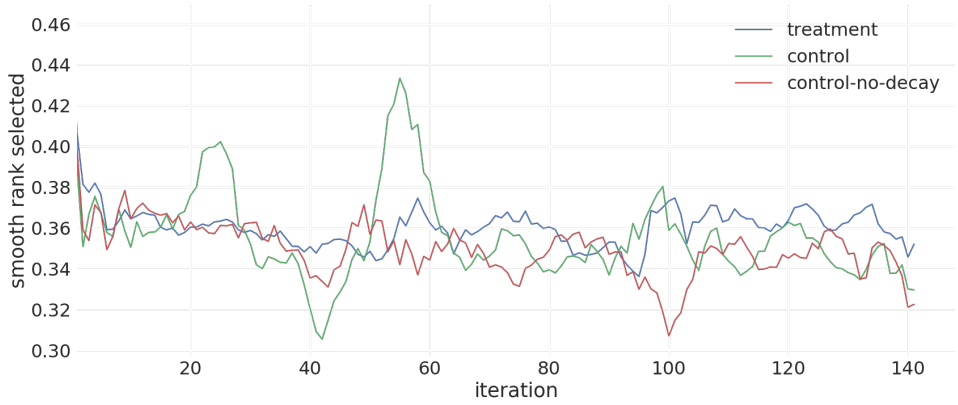



Figure 24: Selected rank over time (smoothed results)

To analyze how many users were really required in the optimization process, it is interesting to check the quality of update estimates. To this end, a simulation based on the actual updates was performed where every iteration could only use 2,000 update reports. These 2,000 updates were randomly sampled from the ones received in that iteration. Because of the randomness involved, this process was repeated 50 times. Since these

simulations were highly computationally expensive, it was very useful that the data was stored in Parquet and could easily be queried using MapReduce Spark jobs.

Figure 25 shows the average difference between the new estimate and the actual update, as well as the standard deviation. The mean is displayed as a dot, while the standard deviation is shown as a bar. To compute a distance between the two update vectors, the $L_1$-distance is used. As time goes on, the distance decreases. From iteration 95 on, it is possible to compute good update estimates based on only the 2,000 updates. This shows that more users are necessary in the earlier iterations of the optimization process. In the beginning, there is more variance in what updates users might propose.

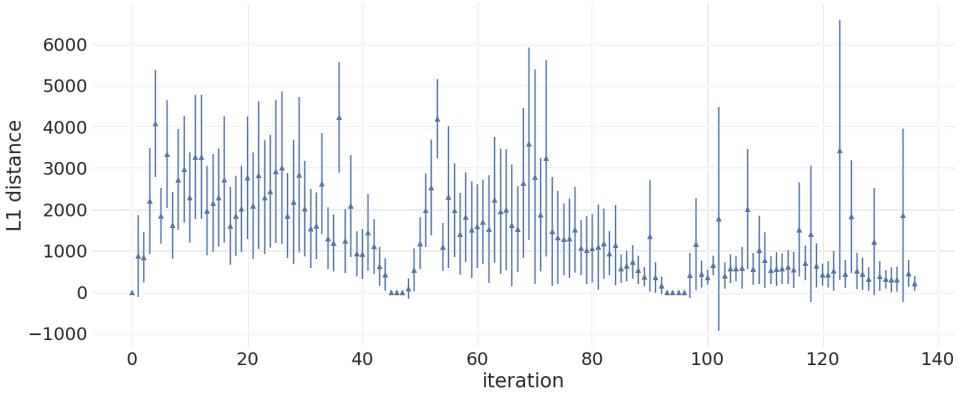



Figure 25: Mean and standard deviation of difference in update quality when only using 2,000 updates

Since this study was based on the RProp optimizer, these updates can be analyzed in more detail. Concretely, only the sign of the update is important for RProp. Figure 26 shows the same simulation but only the signs of the updates are used. Now, even in the last few iterations, the results are not that stable. This shows that the number of required users heavily depends on the exact optimizer that is being used. Just reducing the variance in gradient magnitudes by sampling more users is not useful for RProp as long as this does not improve the estimate of the signs.

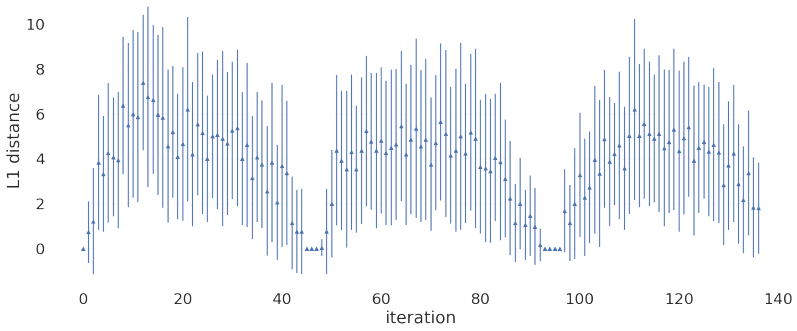



Figure 26: Mean and standard deviation of difference in the signs of the update when only using 2,000 updates

Estimating the loss well is also important since it is supervised during the training process. Figure 27 shows the same simulation when checking the loss. As time goes on, it becomes much easier to estimate the loss. The standard deviation decreases strongly. At least partly, this is because the loss itself is reduced over time. Since more clients report a loss of 0 towards the end, the loss becomes easier to estimate, even when only using a subset of clients.

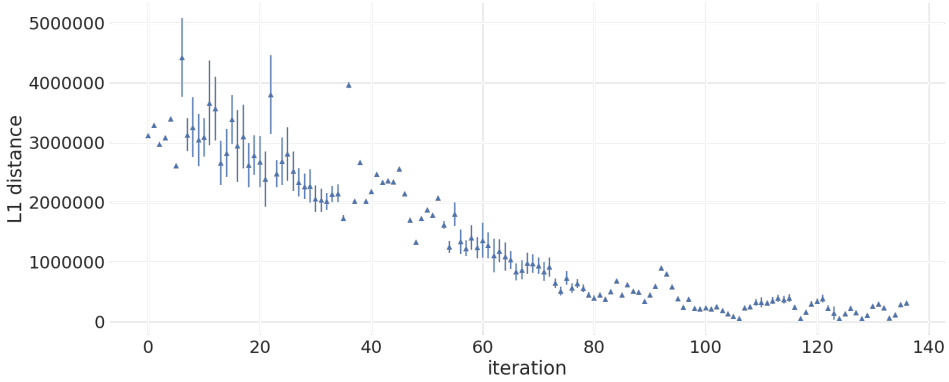



Figure 27: Mean and standard deviation of difference in loss quality when only using 2,000 updates

We can conclude that it makes sense to set iteration lengths more dynamically in this experiment. For one, this would allow making better use of data during time periods of strong user activity. Furthermore, iteration lengths could be increased or decreased depending on how stable the current update and loss are.

# 8 Conclusion

This thesis introduced Federated Learning and gave an overview over the various subareas related to it. Privacy concerns are the primary motivation behind this approach to machine learning. A lot of data on consumer devices is considered private and should not be sent to a server. Federated Learning allows training a model on this data by first processing it locally. Only weight updates derived using the data are sent to a server.

It was shown that this optimization process can compute unbiased gradient estimates, similar to mini-batch gradient descent. However, it can take a long time until individual iterations are completed since the server needs to wait until users can respond with updates. To decrease convergence time, several optimization-related strategies were introduced.

A direct implementation of Federated Learning can require a lot of communication between clients and the server. To make Federated Learning scale efficiently with the size of the model, special compression techniques were discussed. It was shown how they can be used to make uploading data as fast as downloading.

To ensure that Federated Learning is truly better for privacy, Differential Privacy techniques were discussed. By bounding how much individuals can influence the model weights and by randomizing updates, we can quantify how difficult it is to make conclusions about individuals. While this approach does come at a large computational cost, it is interesting from a theoretical perspective because it shows that models can be trained in completely privacy-respecting ways.

Furthermore, new strategies for personalizing models in Federated Learning were proposed. A simple approach based on Transfer Learning makes it possible to customize models locally. Through a more complex architecture, clients can continue to help train the central model while also locally personalizing it.

To evaluate how well Federated Learning can work in a real application, an implementation for Mozilla Firefox was developed. A model was trained on the data of over a million URL bar searches performed by Firefox users. Since the loss consistently decreased over the iterations, this shows that the optimization process can work well in practice.

For future work, it still remains to be shown that more advanced techniques such as differentially-private Federated Learning and the personalization approaches can work in practice. The model implemented for Firefox had to be simple to make it possible to develop the entire system during the work behind this thesis. Literature about training more complex models outside of simulations is still missing.

While differentially-private Federated Learning has theoretical guarantees for the level of privacy, the additional computational cost is huge. Future research could focus on making it more feasible to use these techniques.

One area of research that is outside the scope of this thesis is cryptography. By using encryption techniques, it can be ensured that the server can only read updates from users once a certain number of them were received [14, 42]. These methods make man-in-the-middle attacks much more difficult.

While there is future work left, this thesis showed that Federated Learning can be useful for training models in strongly privacy-respecting ways. This is especially valuable to organizations that do not want to collect certain kinds of data but that still want to make use of machine learning techniques. By training on data that could otherwise not be used, it is possible to fit new kinds of models and to build more intelligent applications.

# References

[1] Abadi, D. J., Boncz, P. A., and Harizopoulos, S. (2009). Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016a). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[3] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. (2016b). Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM.

[4] Adomavicius, G. and Tuzhilin, A. (2011). Context-aware recommender systems. In *Recommender systems handbook*, pages 217–253. Springer.

[5] An, G. (1996). The effects of adding noise during backpropagation training on a generalization performance. *Neural computation*, 8(3):643–674.

[6] Bahdanau, D., Cho, K., and Bengio, Y. (2014a). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[7] Bahdanau, D., Cho, K., and Bengio, Y. (2014b). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[8] Barth-Jones, D. (2012). The're-identification'of governor william weld's medical information: a critical re-examination of health data identification risks and privacy protections, then and now.

[9] Bennett, J., Lanning, S., et al. (2007). The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA.

[10] Berkovsky, S., Kuflik, T., and Ricci, F. (2008). Mediation of user models for enhanced personalization in recommender systems. *User Modeling and User-Adapted Interaction*, 18(3):245–286.

[11] Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1):108–116.

[12] Bishop, C. M. (2006). *Machine learning and pattern recognition*. Information Science and Statistics. Springer, Heidelberg.

[13] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

[14] Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. (2016). Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482*.

[15] Botev, Z. and Ridder, A. Variance reduction. *Wiley StatsRef: Statistics Reference Online*.

[16] Brown, A. L. and Kane, M. J. (1988). Preschool children can learn to transfer: Learning to learn and learning from example. *Cognitive Psychology*, 20(4):493–523.

[17] Cao, Y., Xu, J., Liu, T.-Y., Li, H., Huang, Y., and Hon, H.-W. (2006). Adapting ranking svm to document retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 186–193. ACM.

[18] Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., and Li, H. (2007). Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136. ACM.

[19] Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Olukotun, K., and Ng, A. Y. (2007). Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288.

[20] Csáji, B. C. (2001). Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48.

[21] Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*.

[22] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941.

[23] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[24] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

[25] Diez, D., Barr, C., and Çetinkaya-Rundel, M. (2015). *OpenIntro Statistics*. OpenIntro, Incorporated.

[26] Dwork, C., Roth, A., et al. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407.

[27] Ellis, P. D. (2010). The essential guide to effect sizes: An introduction to statistical power. *Meta-Analysis and the Interpretation of Research Results.: Cambridge University Press*.

[28] Erlingsson, Ú., Pihur, V., and Korolova, A. (2014). Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067. ACM.

[29] Fredrikson, M., Jha, S., and Ristenpart, T. (2015). Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333. ACM.

[30] Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York.

[31] Garson, G. D. (1991). Interpreting neural-network connection weights. *AI expert*, 6(4):46–51.

[32] Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y. (2011). Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM.

[33] Ghahramani, Z. (2004). Unsupervised learning. In *Advanced lectures on machine learning*, pages 72–112. Springer.

[34] Golub, G. H. and Reinsch, C. (1970). Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420.

[35] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

[36] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

[37] Griewank, A. (2012). Who invented the reverse mode of differentiation? *Documenta Mathematica, Extra Volume ISMP*, pages 389–400.

[38] Halevy, A., Norvig, P., and Pereira, F. (2009). The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12.

[39] Han, S., Mao, H., and Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[40] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. (2014). Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*.

[41] Härdle, W. and Simar, L. (2007). *Applied multivariate statistical analysis*, volume 22007. Springer.

[42] Hardy, S., Henecka, W., Ivey-Law, H., Nock, R., Patrini, G., Smith, G., and Thorne, B. (2017). Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677*.

[43] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[44] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

[45] Hsu, J., Gaboardi, M., Haeberlen, A., Khanna, S., Narayan, A., Pierce, B. C., and Roth, A. (2014). Differential privacy: An economic method for choosing epsilon. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 398–410. IEEE.

[46] Huh, M., Agrawal, P., and Efros, A. A. (2016). What makes imagenet good for transfer learning? *arXiv preprint arXiv:1608.08614*.

[47] Igel, C. and Hüsken, M. (2003). Empirical evaluation of the improved rprop learning algorithms. *Neurocomputing*, 50:105–123.

[48] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM.

[49] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[50] Konečnỳ, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency.

[51] Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8).

[52] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[53] Lam, X. N., Vu, T., Le, T. D., and Duong, A. D. (2008). Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 208–211. ACM.

[54] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

[55] Lee, J. and Clifton, C. (2011). How much is enough? choosing $\varepsilon$ for differential privacy. In *International Conference on Information Security*, pages 325–340. Springer.

[56] Liang, Y., Balcan, M.-F., and Kanchanapally, V. (2013). Distributed pca and k-means clustering. In *The Big Learning Workshop at NIPS*, volume 2013.

[57] Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7.

[58] Liu, T.-Y. et al. (2009). Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331.

[59] Mack, C. A. (2011). Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2):202–207.

[60] Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

[61] Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[62] McKnight, P. E. and Najab, J. (2010). Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1.

[63] McMahan, H. B., Moore, E., Ramage, D., Hampson, S., et al. (2017). Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*.

[64] McMahan, H. B., Ramage, D., Talwar, K., and Zhang, L. (2018). Learning differentially private language models without losing accuracy. In *International Conference on Learning Representations (ICLR)*.

[65] Medsker, L. and Jain, L. (2001). Recurrent neural networks. *Design and Applications*, 5.

[66] Napierala, M. A. (2012). What is the bonferroni correction. *AAOS Now*, 6(4):40.

[67] Narayanan, A. and Shmatikov, V. (2006). How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105*.

[68] Navarro, C. A., Hitschfeld-Kahler, N., and Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329.

[69] Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., and Martens, J. (2015). Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*.

[70] Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724.

[71] Pan, S. J., Yang, Q., et al. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

[72] Panagiotakopoulos, C. and Tsampouka, P. (2013). The stochastic gradient descent for the primal l1-svm optimization revisited. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 65–80. Springer.

[73] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

[74] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[75] Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

[76] Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer.

[77] Ricci, F., Rokach, L., and Shapira, B. (2011). Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer.

[78] Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE.

[79] Rojas, R. The eightfold path to linear regression. *Freie University, Berlin, Tech. Rep.*

[80] Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media.

[81] Rojas, R. (2015a). The bias-variance dilemma. *Freie University, Berlin, Tech. Rep.*

[82] Rojas, R. (2015b). Logistic regression as soft perceptron learning. *Freie University, Berlin, Tech. Rep.*

[83] See, A., Liu, P. J., and Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*.

[84] Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024.

[85] Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*. MIT press.

[86] Sweeney, L. (2000). Simple demographics often identify people uniquely. *Health (San Francisco)*, 671:1–34.

[87] Torrey, L. and Shavlik, J. (2010). Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, pages 242–264. IGI Global.

[88] Voinov, V. G. and Nikulin, M. S. (2012). *Unbiased Estimators and Their Applications: Volume 1: Univariate Case*, volume 263. Springer Science & Business Media.

[89] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328.

[90] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

[91] Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM.

[92] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. (2008). Large-scale parallel collaborative filtering for the netflix prize. In *International Conference on Algorithmic Applications in Management*, pages 337–348. Springer.

## Online References

[93] Advancing state-of-the-art image recognition with deep learning on hashtags. `https://code.facebook.com/posts/1700437286678763/advancing-state-of-the-art-image-recognition-with-deep-learning-on-hashtags/`. Accessed: 2018-06-07.

[94] Ai and compute. `https://blog.openai.com/ai-and-compute/`. Accessed: 2018-08-03.

[95] Apache parquet. `https://parquet.apache.org`. Accessed: 2018-08-07.

[96] Federated learning – client implementation. `http://github.com/florian/federated-learning-addon`. Accessed: 2018-08-08.

[97] Federated learning – server implementation. `https://github.com/mozilla/telemetry-streaming/tree/master/src/main/scala/com/mozilla/telemetry/learning/federated`. Accessed: 2018-08-08.

[98] Federated learning – simulations. `https://github.com/florian/federated-learning`. Accessed: 2018-08-07.

[99] Firefox telemetry. `https://firefox-source-docs.mozilla.org/toolkit/components/telemetry/telemetry/`. Accessed: 2018-08-07.

[100] Firefox telemetry – rank of selected url bar suggestions by type. `https://mzl.la/2ONXI5N`. Accessed: 2018-08-07.

[101] For self-driving cars, there's big meaning behind one big number: 4 terabytes. `https://newsroom.intel.com/editorials/self-driving-cars-big-meaning-behind-one-number-4-terabytes/`. Accessed: 2018-06-07.

[102] Frecency – mozilla developer network. `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Places/Frecency_algorithm`. Accessed: 2018-08-07.

[103] Pretrained imagenet models. https://github.com/cvjena/cnn-models. Accessed: 2018-08-14.

[104] Shield – mozilla wiki. https://wiki.mozilla.org/Firefox/Shield. Accessed: 2018-08-19.

[105] Speedtest mobile report 2018. https://web.archive.org/web/20180521150459/http://www.speedtest.net/reports/united-states/#mobile. Accessed: 2018-06-09.

[106] Speedtest report 2018. https://web.archive.org/web/20180516152912/http://www.speedtest.net/reports/. Accessed: 2018-05-16.

[107] tikz visualization template adapted from. https://github.com/davidstutz/latex-resources/tree/master/tikz-gradient-descent. Accessed: 2018-06-17.

[108] Voter list information. http://voterlist.electproject.org. Accessed: 2018-08-13.