Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Fakultät für Informatik und Wirtschaftsinformatik

Master-Thesis

Name:	Tobias Wink
Thema:	A decentralized communication approach for federated learning
Arbeitsplatz:	Hochschule Karlsruhe – Technik und Wirtschaft, Karlsruhe
Referent:	Prof. Dr. Nochta
Korreferent:	Prof. Dr. Körner
Abgabetermin:	30.09.2020

Karlsruhe, 01.04.2020

Der Vorsitzende des Prüfungsausschusses

Prof. Dr. Heiko Körner

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature

Abstract

Artificial intelligence (AI), mostly based on machine learning (ML), has already found its way into daily life areas. The core component of software based on ML is the model, which is often trained with specially prepared data. A clear trend can be seen that models are becoming more and more complex so that both more training data and more computing capacity are needed for successful training. This dilemma is to be compensated by distributed learning. Distributed learning enables the distribution of the necessary computations, the so-called training, within a data center. This requires that the training data is available at a central location for training, becoming increasingly difficult due to stricter data protection laws, such as the General Data Protection Regulation (GDPR).

For this reason, the field of federated learning has emerged, in which Google is pathbreaking with a technique called federated optimization (FO). This way, training data no longer has to be collected centrally but can remain on the training devices. For FO, the training devices send the training results to a central server that combines the received results into an overall model. A disadvantage of the previous federated learning approaches is that they require a central server to generate the overall model. That implies the central server operator can access all transmitted training results and thus gain advantages. Thereby possible usage scenarios are limited. Companies would not get involved in developing a common model if one could gain advantages through server operation hidden from the others. However, the jointly developed model could be better than that of a single party.

This thesis shows an approach for developing a common model without a central server to expand the application possibilities of federated learning. For this purpose, the necessary basics are first taught. Subsequently, the current approaches for distributed and federated learning will be discussed to develop approaches without a central server. Experiments will be used to show how well the approaches developed in this way work.

Kurzfassung

Künstliche Intelligenz, insbesondere auf Basis von maschinellem Lernen, hat bereits in immer mehr Bereichen des täglichen Lebens Einzug gehalten. Die Kernkomponente einer Software, die auf maschinellem Lernen basiert, ist das sogenannte Modell, welches oftmals mit Hilfe von speziell aufbereiteten Daten trainiert wird. Es ist ein klarer Trend zu erkennen, der die Modelle immer komplexer werden lässt, sodass sowohl mehr Trainingsdaten als auch mehr Rechenkapazität für das erfolgreiche Training benötigt werden. Dies soll durch verteiltes Lernen kompensiert werden. Dadurch wird es möglich, die notwendigen Berechnungen, das sogenannte Training, innerhalb eines Rechenzentrums zu verteilen. Beim verteilten Lernen ist es erforderlich, dass die Trainingsdaten an einem zentralen Ort für das Training bereitliegen, was durch strenger werdende Datenschutzgesetze, wie der Datenschutzgrundverordnung (DSGVO), immer schwieriger wird.

Aus diesem Grund ist der Bereich des föderierten Lernens entstanden, in welchem Google mit einer Technik namens föderierte Optimierung federführend ist. So müssen die Trainingsdaten nicht mehr zentral gesammelt werden, sondern können auf den trainierenden Geräten verbleiben. Bei föderierter Optimierung werden die Trainingsergebnisse an einen zentralen Server gesendet, der die erhaltenen Daten zu einem Gesamtmodell zusammenfasst.

Ein Nachteil bei den bisherigen Ansätzen föderierten Lernens ist, dass sie einen zentralen Server für die Erzeugung des Gesamtmodells benötigen. Dies impliziert, dass der Betreiber des zentralen Servers Zugriff auf alle übermittelten Trainingsergebnisse erhalten und sich so Vorteile verschaffen kann. Hierdurch werden mögliche Einsatzszenarien eingeschränkt, da bspw. Konkurrenten sich nicht auf die Entwicklung eines gemeinsamen Modells einlassen würden, wenn sich ein Wettbewerber durch den Serverbetrieb Vorteile verschaffen kann, die den anderen verborgen bleiben, obwohl das gemeinsam entwickelte Modell besser werden könnte als das eines Einzelnen. Um die Einsatzmöglichkeiten föderierten Lernens zu erweitern, wird in dieser Arbeit gezeigt, dass die Entwicklung eines gemeinsamen Modells ohne zentralen Server möglich ist. Dafür werden zuerst die notwendigen Grundlagen vermittelt. Anschließend werden die aktuellen Ansätze für verteiltes und föderiertes Lernen beleuchtet, um darauf aufbauend Ansätze zu entwickeln, die ohne zentralen Server auskommen. Wie gut die so entwickelten Ansätze funktionieren, wird mit Hilfe von Experimenten gezeigt.

Acknowledgement

I wish to thank all the people whose assistance was a milestone in the completion of this project.

I want to thank my thesis advisor Professor Zoltan Nochta of the Faculty of Computer Science and Business Information Systems at Karlsruhe University of Applied Science. The “door” to Professor Nochta‘s office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work but steered me in the right direction whenever he thought I needed it.

Furthermore I would like to thank all proofreaders. Without you, this work would probably only be half as understandable. I would especially like to thank Stefan and Steffen for their time and support.

Finally, I must express my very profound gratitude to my wife, Andrea, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without her. Thank you.

Contents

Acronyms

2PC secure two-party computation. 27

AI artificial intelligence. 4, 13

ConvBlock convolutional block. 71

DL deep learning. 15

DNN deep neural network. 20

FO federated optimization. 4, 13

GAMIN Generative Adversarial Model INversion. 45

GDPR General Data Protection Regulation. 4, 13

GPU graphics processing unit. 15

HDF5 Hierarchical Data Format version 5. 51

IdBlock identity block. 71

IID identically and independently distributed. 9, 36

ML machine learning. 4, 13

MPC secure multi-party computation. 27

SGD stochastic gradient descent. 20

SL split learning. 39

1 Introduction

> *"As a technologist, I see how AI and the fourth industrial revolution will impact every aspect of people's lives."* (Fei-Fei Li [62])

Artificial intelligence (AI), mostly based on machine learning (ML), has already found its way into daily life areas, *e. g*. autonomic parking, intelligent personal assistants, or word completion on smartphone keyboards, to name a few. The central core of ML-based software is called the model.

To create such a model, or more precisely to train it, often specially prepared data is required. Initially, and still widely used today, this data was or is stored in a central location. This is the reason why this procedure is called centralized learning. A clear trend can be seen that models are becoming more and more complex so that both more training data and more computing capacity are needed for successful training. This dilemma is to be compensated by distributed learning. Distributed learning enables the distribution of the necessary computations, the so-called training, within a data center. Therefor the training data must be available at a central location. This is becoming increasingly difficult due to stricter data protection laws, such as the General Data Protection Regulation (GDPR).

For this reason, an area called federated learning has been developed in which Google is pathbreaking with a technique called federated optimization (FO) [69]. Federated optimization (FO) allows a model's training without transferring the training data to a central location. An existing model is trained locally with the local data, and it is sent afterward to a central server, where all received models are then reintegrated into an overall model. This technique is used for example to improve mobile keyboard prediction by Google [58, 89, 109] and Apple [47].

Current techniques for federated learning are designed to give one party an advantage, namely the party with the central location under its care. The central location is where all training results converge and are processed further. What if several parties want to develop a common application model without giving one party the named advantage? For example, some banks want to develop a model for credit card fraud detection. Often better results can be achieved with more data [37]. Therefore, said parties could develop a better model if they would cooperate for the training. No company is willing to share its data, especially not its user data, particularly not with competitors. As Ginni Rometti states

> *"I want you to think about data as the next natural resource."* ([93])

In this scenario, the current techniques for federated learning are not sufficient. Competitors do not want one party to have an advantage in cooperation. Therefore, an approach is needed here which prevents such an advantage by making the central location unnecessary and does not disclose any data. This thesis aims to demonstrate such a solution. It will be shown how federated learning is possible without sharing training data and without the need to operate a central infrastructure. It will also be shown how federated learning is possible with a decentralized communication approach while preserving privacy.

1.1 Structure

Chapter 2 covers some basics which are then used throughout this thesis.

In Chapter 3, related work is described.

Starting with Chapter 4, the author's contribution to this field of research begins. This chapter deals with the analyses of existing approaches and the design of practical solutions.

Chapter 5 gives an overview of the realization of a prototype and highlights technically exciting approaches.

Chapter 6 shows how the selected approaches prove themselves in different scenarios.

In Chapter 7, a conclusion is drawn.

2 Basics

For a complete understanding of this thesis‘s contents, knowledge in the fields of AI, more precisely ML, especially deep learning (DL), and cryptography, more precisely secure computation, is required.

2.1 Artificial intelligence and machine learning

John McCarthy coined the term artificial intelligence (AI) as part of a proposal for a research project in 1955 [78]. In general, AI describes *“the effort to automate intellectual tasks normally performed by humans”* [29]. Therefore, the field of AI is vast. In the first 30 years, the focus was on systems for specific areas, in which the developers tried to make all necessary decisions through explicit rules, the so-called expert systems. Apart from the first chess computers, ELIZA [108] is probably one of the best-known expert systems representatives. ELIZA is a therapist-inspired natural language processing computer program developed by Joseph Weizenbaum in the mid-1960s. It attempts to maintain a conversation with the user by searching for the most critical keywords in the message and generating either statements or questions from them.

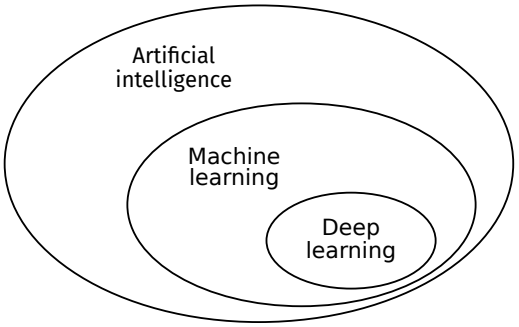

Figure 2.1: Connection between Artificial intelligence, machine learning and deep learning [29]

In addition to expert systems, there is also the broad area of machine learning (ML). The foundations of ML were laid early on, as will be shown in the following sections. Since the creation of ML systems, the so-called training, requires a relatively large amount of computing power and input data, it has been neglected for decades. Due to the widespread use of the internet and the associated progressive digitization, ever-larger amounts of data is available. Due to the significant advances in processors‘ computing power, especially graphics processing units (GPUs), the focus has now shifted towards ML, especially DL. To better understand the relationship between AI, ML, and DL, this is shown graphically in Figure 2.1. This change of focus has led to specific hardware, tools, and frameworks in this area. While computer games were almost exclusively responsible for GPUs‘ rapid development from the beginning, there are now separate graphic card series that are specially designed for use in the ML context, such as the NVIDIA A100[1].

[1] https://www.nvidia.com/de-de/data-center/a100/

2.1.1 A brief overview of machine learning

The term machine learning (ML) was coined by Arthur L. Samuel 1959 [97], often quoted with *"Field of study that gives computers the ability to learn without being explicitly programmed."* [4, 14], which is probably the interpretation of *"Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort."* [97].

Samuel developed a computer program for the game checkers [97]. The next move is chosen with the help of a scoring function, which estimates the chances of winning based on the current game state. The program used the current game state and also all past game states so that the program was able to "learn".

Around the same time, Frank Rosenblatt developed the perceptron [94], a simplified model of a biological neuron, as described by Warren McCulloch and Walter Pitts [79]. This perceptron can work as a linear binary classifier. For correct classification, the perceptron can be trained using data for which the correct classes are already known. For more details, see Section 2.1.3.

Between 1956 and 1967, the K-means algorithm was developed independently by Steinhaus (1956) [101], Lloyd (proposed in 1957 but published in 1982) [73], Forgy (1965) [46], and MacQueen (1967) [75]. K-means is an algorithm for cluster analysis, which can be defined as follows: *"a statistical classification technique for discovering whether the individuals of a population fall into different groups by making quantitative comparisons of multiple characteristics"* [83]

These three examples are all examples of ML since, in all three cases, knowledge is derived from the data by algorithms. However, the procedure for these three examples is different in each case because each example represents a different category of ML. At the top level, a distinction is made between the following ML categories:

Reinforcement learning An agent interacts with the world and learns with rewards. His goal is to maximize the reward or to minimize the punishment, *e. g.* Samuels checker program.

Supervised learning An algorithm (f_θ) learns a mapping from data (x) to the corresponding answer (y): $f_\theta(x) = y$, *e. g.* Perceptron.

Unsupervised learning Unsupervised learning aims to find knowledge in the form of structure in data, *e. g.* K-means.

Since the currently known algorithms for federated learning are related to supervised learning, this thesis will focus on that area.

Supervised learning

Besides the previously abstract mathematical definition of supervised learning, it is perhaps more accessible to compare it with classical software development. In classical software development, the developer of a program defines rules that convert a particular input into a particular output. With supervised learning, such rules are unknown in the beginning. They are derived during the training phase from the inputs (training data) and the corresponding desired outputs (label) in a

model. This trained model is then used in later applications to convert unknown inputs into outputs. It is essential to ensure the resulting outputs meet the expectations that the used training data is as representative as possible for the selected problem.

Over time, various types of algorithms have been developed in the field of supervised learning, such as support vector machines [19], decision trees [24], or artificial neural networks [79], to name a few.

Typical application scenarios for supervised learning are:

Classification To which predefined class does an object correspond, *e. g.* is this a picture of a cat or dog?

Regression What is the next continuous value (Prediction), *e. g.* temperature forecast.

The algorithm created in the context of supervised learning is called model. The process of creating or learning is generally referred to as training. Three things are needed to train a model:

1. Input data
2. Expected output
3. A weighting function to determine how well the algorithm has achieved the expected results, the so-called loss. In practice, the loss will be measured and an optimizer changes the model‘s parameters to minimize the loss. The loss function and the optimizer depend on the specific task.

Input data, in conjunction with the expected output, is called annotated data. The result of the weighting function is used as a feedback signal to adjust the model so that future results better match the expected results. This process is called training.

2.1.2 Tensor

Tensors are the most basic data structure in the context of ML. F. Chollet describes them as: *“At its core, a tensor is a container for data—almost always numerical data. So it’s a container for numbers. [. . .] tensors are a generalization of matrices to an arbitrary number of dimensions [. . .].”* [29].

Three key attributes define a tensor:

The number of dimensions Alternatively: Number of axes. For instance, a vector (1D tensor) has one axis, a matrix (2D tensor) has two axes, and a scalar corresponds to a 0D tensor.

Shape This integer tuple describes the number of values per axis, *e. g.* a $3x3$-Matrix has shape $(3, 3)$.

Data type This specifies the data type of the contained data. For the MNIST dataset (see Section 6.1.1), float32 will encode each pixel‘s gray value.

To get a more vivid idea of tensors, here are some typical examples from ML for input data [29]:

Scalar data	1D tensor
Vector data	2D tensors of shape (samples, features)
Timeseries data	3D tensors of shape (samples, timesteps, features)
Images	4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
Video	5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

2.1.3 Artificial neural networks and deep learning

Artificial neural networks consist of artificial neurons that are interconnected as in a network. They describe a family of algorithms for supervised learning and can accordingly be used for classification and regression. The idea for artificial neural networks came up in 1943 and can be traced back to Warren McCulloch and Walter Pitts [79].

Perceptron

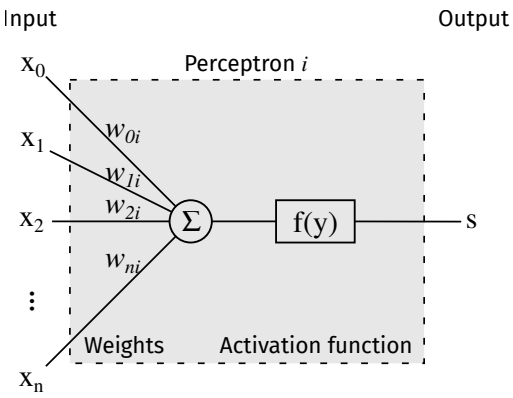

Figure 2.2: Schematic representation of a perceptron

As already mentioned, in 1958, Frank Rosenblatt developed a simplified model of a biological neuron as a linear binary classifier, which he called perceptron [94]. Figure 2.2 shows a perceptron as a graph. The output s of a perceptron i is the dot product of the input vector $\vec{x}$, and a weighting vector $\vec{w}_i$ applied to an activation function $f(y)$. The formula is as follows:

$$s = f(\vec{x} \bullet \vec{w}_i) = f(\vec{x}^T \vec{w}_i) = f(\Sigma_{i=0}^{n} x_i w_i)$$

Often x_0 is fixed 1 to shift the dot product‘s result into the correct range using w_0. Then w_0 is also known as *bias*. In this case, one also refers to a biased perceptron. In practice, the fixed value 1 is not placed in front of the input vector $\vec{x}$; instead w_0, the bias, is added at the end so that the formula used is the following:

$$s = f(\vec{x}^T \vec{w}_i) + w_0$$

To differentiate between bias and other weights, from now on, w_0 is referred to as bias and the other weights as kernel weights, a notation which is typical for TensorFlow and Keras (Section 2.1.6).

Algorithm 2.1 Train a perceptron

Input:
Training set $S = (x_1, s_1), \ldots, (x_n, s_n)$, with x_i as input vector and s_i as desired result
Learning rate r between 0 and 1
An initialized $\vec{w}$ with random values between 0 and 1

```
function FIT(S, r)
    for all (x_j, s_j) ∈ S do
        y = f(w • x)                                   // Calculate result
        for all w_i ∈ w⃗ do
            w_i = w_i + r · (s_j − y)x_j    // Adjust the weighting to approximate the desired result
        end for
    end for
end function
```

The task of the activation function is to separate two classes. The most basic activation function is a step function, as shown in Figure 2.3, which returns 0 for negative input values and 1 for positive input values. So if the output of a perceptron is 1, a value belongs in class 1, otherwise in class 2.

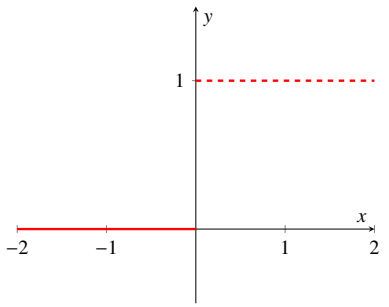

Figure 2.3: Step function

For the perceptron to classify correctly, it must calculate the correct classes from the input data. In order for this to work, the weights must have the correct values. The correct values can also be determined manually for simple examples, but in most cases, this is trained using annotated data, as shown in Algorithm 2.1. The learning rate r shown there limits the progress per execution step so that possible minima are not skipped.

Neural networks

A single perceptron, as a linear classifier, is incapable of classifying non-linear problems. For the classification of non-linear problems, a non-linear classifier is therefore required. Such a classifier can be constructed by using several perceptrons and dividing them at least into two layers, as shown on the left in Figure 2.4. Now, non-linear functions are required as activation functions of the first layer to turn it into a non-linear classifier. Such a construction is also known as a multi-layer perceptron. If a layer has no connection to the input or output values, this layer is called a hidden layer, see Figure 2.4.

If a hidden layer is present, a new way to train the weights is needed. The backpropagation algorithm provides a solution. The term backpropagation and its use in neural networks were published in 1986 [96]. A prerequisite for this algorithm is the existence of a loss function. As mentioned before, a loss function‘s task is to measure a model‘s failure, e. g. the number of misclassified inputs. Backpropagation is a way to compute the gradients of the weights of all layers regarding the loss and consists of two phases:

1. Forward Pass Computation of the loss function values, for example, by comparing the determined classes with the desired classes.

2. Backward Pass Computation of the gradients‘ values of the different weights with subsequent optimization by using a gradient descent algorithm.

The loss function used depends on the problem definition or the architecture, *e. g.* for binary classification, binary cross-entropy is used. For regression, mean squared error is used.

The algorithm used for the gradient descent is called optimizer function or just optimizer. The optimizer‘s typical algorithm is stochastic gradient descent (SGD), a stochastic approximation of gradient descent. In the meantime, other optimization functions appeared, often based on the SGD, *e. g.* AdaGrad, RMSProp, or Adam.

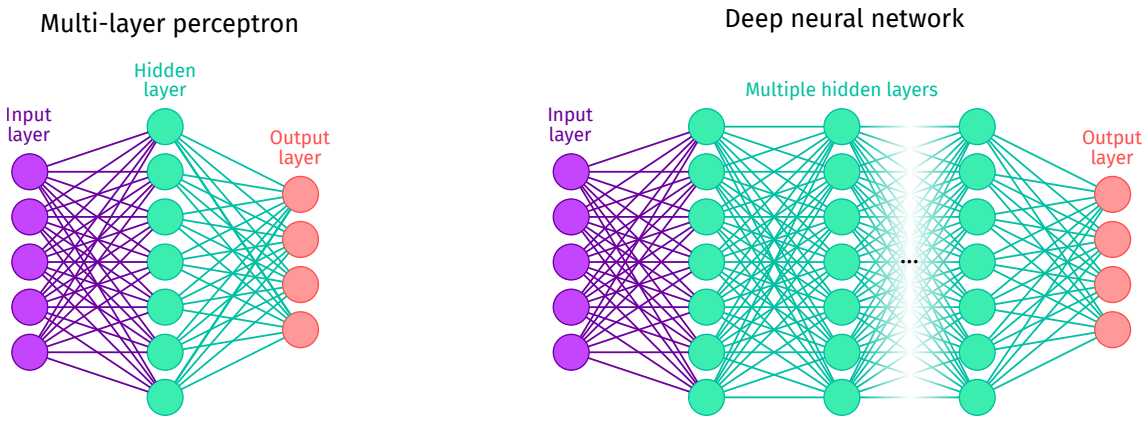

Figure 2.4: A multi-layer perceptron in comparison to a deep neural network, inspired by [107]

Neural networks with one hidden layer can solve all classification problems, as initially proved by Kolmogorov in 1957 [66] and transferred to neural networks by Hecht-Nielsen in 1987 [61]. Nevertheless, approaches using several hidden layers have become more and more popular in the last decade. A possible explanation for this is provided by [6]. There it states: *“The results suggest that the strength of deep learning may arise in part from a good match between deep architectures and current training procedures, and that it may be possible to devise better learning algorithms to train more accurate shallow feed-forward nets. For a given number of parameters, depth may make learning easier, but may not always be essential.”* [6]

There are different other reasons why deep neural network (DNN) usually produces better results, but among the probably most important are certainly:

- With each additional hidden layer, the number of optimization tasks increases, but each such optimization task‘s complexity is less than one hidden layer and the same total number of perceptrons.

  The downside is that the learning cycle (feed-forward and backpropagation) will be extended by each additional layer because, in each layer, the dot product (feed-forward) and the associated gradients (backpropagation) must be calculated.

  Here is an example: A network with 1200 perceptrons in a hidden layer must find an optimized weight vector $\vec{w} \in \mathbb{R}^{1200}$ during optimization.

If the 1200 perceptrons are divided into three hidden layers in the ratio 600 : 400 : 200, the optimization task has to find three optimized weight vectors $\vec{w}_{h1} \in \mathbb{R}^{600}$, $\vec{w}_{h2} \in \mathbb{R}^{400}$, and $\vec{w}_{h3} \in \mathbb{R}^{200}$.

- Particular types of layers can be used for specific tasks. So a model can deliver better results despite less training. For example, there are so-called convolutional layers for image recognition, which allow a network to recognize shapes independent of their current position, see Section 2.1.5.

Since their first major success in the ImageNet 2012 competition, these so-called DNNs have developed into a separate sub-field of ML deep learning (DL). Figure 2.4 shows a comparison between a multi-layer perceptron or a classical neural network and a DNN.

Nevertheless, DNNs do not only have advantages. For example, there is the vanishing gradient problem, which can cause early layers to learn significantly slower than later layers. The problem occurs when the chain rule is used in the gradient calculation, and an activation function is used whose derivatives have values between 0 and 1. By using the chain rule, the respective gradients have a multiplicative connection to the subsequent layers. This, combined with small values derived by the activation function, results in smaller and smaller gradients. The problem can be mitigated by using other activation functions or by using alternative model architectures in which the interconnection of the layers is not strictly left-to-right, such as in recurrent neural networks or residual neural networks.

2.1.4 Activation functions

As already mentioned in Section 2.1.3, activation functions are executed after the actual link operation, *e. g.* for a perceptron, the computation of the dot product. Typically, non-linear functions are used to increase the overall system's expressiveness, in this case, ReLU and softmax.

ReLU

ReLU stands for rectified linear unit. It leaves all positive values unchanged and converts all negative values to 0. Figure 2.5 shows a plot of the ReLU function. The corresponding formula is:

$$\text{relu}(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases}$$

The corresponding gradient is:

$$\text{relu}'(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases}$$

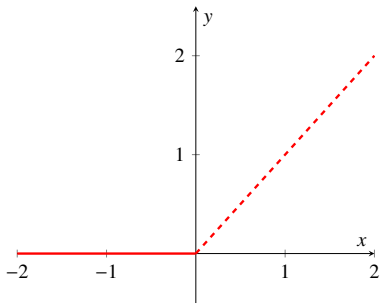

Figure 2.5: ReLU function

Since the gradient for positive numbers is constant 1, the vanishing gradient problem has a much weaker effect. This property might be one reason for the incredible popularity of this activation function.

Softmax

The softmax function is a function to normalize a vector into a probability distribution. This means that all elements of the result lie in the interval $[0, 1]$, and the sum of all elements is 1. The corresponding formula is:

$$\text{softmax}(\vec{x}) = \frac{\exp(x_i)}{\sum_{k=1}^{K} \exp(x_k)} \text{ for } i = 1, \ldots, K \text{ and } \vec{x} = (x_1, \ldots, x_K) \in \mathbb{R}^K$$

It is typically used as an activation function in the last layer for classification problems with more than two classes.

2.1.5 Layers

This section provides a brief overview of the layers used in the models for the experiments in Chapter 6.

Dense layer

A dense layer or fully connected layer corresponds to the already introduced biased perceptron (see Section 2.1.3). Therefore the result can be expressed with the following formula:

$$output = activation((\overrightarrow{input} \bullet \overrightarrow{kernel\ weights}) + bias)$$

Typically, the following parameters are specified:

Units The dimensionality of the output tensor.

Activation The used activation function.

Convolutional layer[2]

Convolutional layers were introduced by LeCun in 1989 [72]. The name convolution already indicates the essential property of the layer, the mathematical operation convolution. Goodfellow sums it up in his book Deep Learning:

"*Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*" [55]

A convolution S is generally specified as:

$$S(x) = (I * K)(x) = \int I(\tau) K(x - \tau) d\tau$$

[2]The main reference for this part is [55].

In ML cases, the discrete convolution is sufficient, which can be defined as:

$$S(x) = (I * K)(x) = \sum_{\tau=-\infty}^{\infty} I(\tau)K(x-\tau)$$

In convolutional layer terminology, the first argument I is often referred to as input and the second argument K as the kernel. The output is often referred to as the feature map.

In ML applications, especially in image recognition, one usually has to deal with multidimensional arrays of data, and the kernel is often multidimensional as well. Therefore the infinite summation can be implemented as a summation over a finite number of array elements.

For example, a two-dimensional image I could be used as input, and the kernel K could also be two-dimensional:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m, j-n)$$

Many ML libraries, including Keras, implement a related function, the cross-correlation, but call it convolution:

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i+m, j+n)K(m,n)$$

Goodfellow writes about this:

"*In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.*" [55]

Convolutional layers offer three essential properties that can lead to better learning results:

- Sparse weights
- Parameter sharing
- Equivariant representations

Sparse weights describe that in a convolutional layer, the kernel is much smaller than the input. In a dense layer, each output unit n interacts with every input unit, but in a convolutional layer, the output units n only interact with the kernel k. Therefore the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. In practice, kernels of the size $(5,5)$ and $(3,3)$ are the most common.

In a dense layer, each weight is used precisely once while computing the output of a layer. In a convolutional layer, the kernel is used at every position of the input. Thereby, the model's storage requirements are reduced to $|k|$, where k is the kernel.

This particular form of parameter sharing causes the layer to have a property called equivariance to translation. This means that if the input changes, the output changes in the same way. For this reason, it is possible for these layers to recognize objects or shapes, for example, in images, regardless of their position.

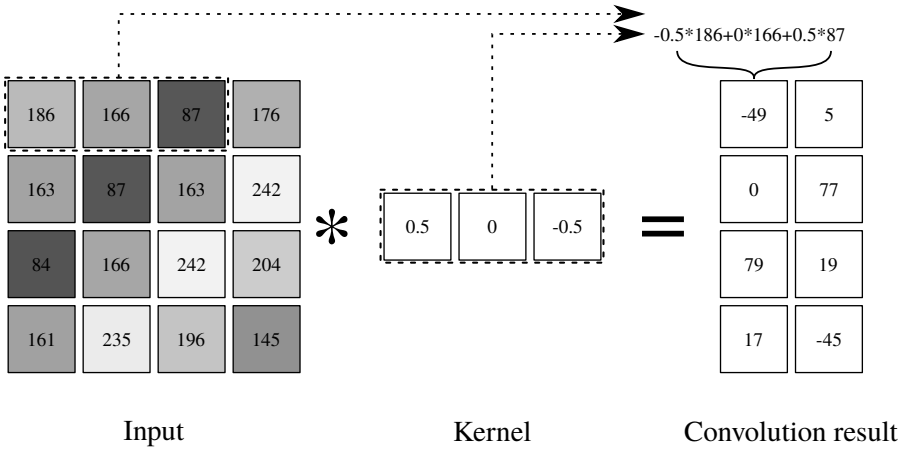

Figure 2.6: An example of a convolution.

The convolution operation in images can also be described visually. The kernel represents a window, which is horizontally mirrored gradually placed over the entire image. A weighted sum is formed at each position by multiplying the image‘s superimposed fields and the kernel and adding up all the multiplication results. All weighted totals determined in this way then result in the output. Figure 2.6 shows an example of a convolution. The output was restricted to only positions where the kernel lies entirely within the image, called “valid” padding. Another variant of padding is “same”, where the borders are filled with 0.

In practice, convolutional layers detect not only one convolution but several, the so-called filters. The name comes from the fact that convolutions are used in computer vision, for example, for smoothing filters.

Typically, the following parameters are specified:

Filters The number of (output) filters.

Kernel size A tuple to specify the dimensionality for the convolution window, *e. g.* $(3, 3)$ to use a window of width 3 and height 3 for an image.

Padding How to proceed with the edges of the input?

Step size By how many pixels should the kernel be moved after each convolution?

Pooling layer

The task of a pooling operation is to downsample the input data. Extraction windows in the size of the specified pool size are superimposed on the input data and combined to one input value each. The used aggregation function depends on the specific characteristics of the pooling layer. Typical examples of pooling layers are *MaxPooling*, *MinPooling*, and *AvgPooling*, where the maximum, minimum, or average of the extraction window values are determined and applied. Pooling enables downstreamed convolutional layers to recognize area-wide or spatial structures. An example of a max–pooling operation is shown in Figure 2.7.

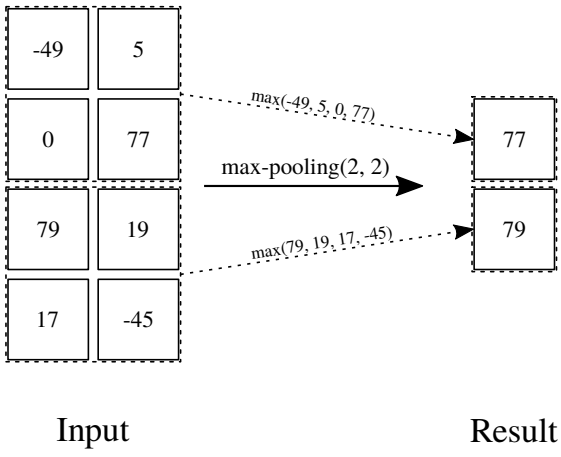

Figure 2.7: An example of max–pooling with a pool size of $(2, 2)$.

Dropout layer

Dropout layers can help to prevent overfitting. Overfitting in this context means that a model becomes too much adjusted to the training data through repeated training. Therefore, the Dropout layer randomly sets input values to 0, and the $rate$ parameter determines the probability of an input value being set to 0. To avoid changing the total sum of input values, all values that are not changed are upscaled by $1/(1 - rate)$.

Batch Normalization layer

Batch normalization [64] is a technique to improve the learning speed by stabilizing the learning process. During backpropagation, the weights of all layers are updated, leading to unwanted effects, *e. g.* the accuracy is much lower than before the update. Through batch normalization, the inputs are so transformed that the output‘s mean value is close to 0, and the standard deviation is close to 1. Thus, the layers before and after the batch normalization layer are more decoupled.

2.1.6 TensorFlow

TensorFlow is an open-source library for ML. It was first developed by the Google Brain team and released in November 2015 under the Apache License 2.0. TensorFlow provides stable APIs for Python and C to develop ML applications. The execution is done via high-performance C++. If an NVIDIA GPU is present, that can be utilized via CUDA. For use with AMD GPUs, there are also first experimental versions based on the ROCm[3] platform.

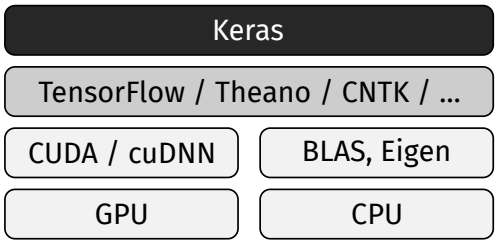

Figure 2.8: Keras software and hardware stack [29].

TensorFlow computations are expressed as stateful dataflow graphs. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a tensor.

Keras

Keras is an open-source library for DL written in Python. It was first developed by François Chollet and released in March 2015 under the MIT license. With Tensorflow 2.0, Keras was integrated into Tensorflow to no longer need to be installed separately. It contains numerous implementations of commonly used neural-network building blocks such as layers, activation functions, and optimizers. One focus of Keras is on user-friendliness. Another focus of Keras is that it can use different libraries as backend, see Figure 2.8. It offers three separate APIs:

Sequential API Focus on user-friendliness but restricted to a linear stack of layers and precisely one input and exactly one output.

Functional API Less user-friendly but no limitation in linking the layers or in the number of inputs and outputs.

Model subclassing Fully-customizable to enable the implementation of custom forward-pass models. Model subclassing is mainly intended for research.

The central element in Keras is an instance of the class `tf.keras.Model`, which groups layers with training and inference functionalities. Such an instance is created with the help of one of the APIs and then configured with optimizer, loss function, and desired training metrics. Due to the structure of `tf.keras.Model` it is not possible to access the weights directly; instead, they are accessible via the respective layers. This characteristic becomes relevant in Chapter 5.

[3] ROCm is an open software platform from AMD for high performance computing and ML. URL: https://www.amd.com/en/graphics/servers-solutions-rocm

2.2 Secure multi-party computation

Previous approaches to federated learning require a central server. This is to be changed by this work. Without such a server, the tasks have to be implemented differently. Good alternatives are protocols of the secure multi-party computation (MPC) field. The MPC goal is to enable a group to perform joint computations without each group member‘s data becoming known to the others. It is a subfield of cryptography. In contrast to the traditional cryptography tasks, where the adversaries come from outside, it is important to protect the individual members‘ data from the other participants.

Secure computation goes back to Andrew Yao [110], who presented the so-called Millionaire's Problem and its solution in 1982. The problem describes two millionaires who want to know which of them is richer without getting more information about the other's wealth. Since two parties are involved, it is called secure two-party computation (2PC). A distinction is often made between 2PC and MPC, because sometimes more efficient protocols can be used for communication between two parties than if there are more parties involved.

A secure computation or secure function evaluation are alternative terms, whereby the definitions in some cases drift apart. Micali and Rogaway [84] designate the original ideas and concepts on this topic and define secure function evaluation to eliminate difficulties they had with the original definition of secure computation. In contrast, Beerliová-Trubíniová *et al*. [10] with MPC mean the generalization of secure function evaluation, which can also store intermediate results. In this thesis, all three terms are used equally.

There are a few basic methods with which MPC protocols are implemented. Sometimes they are combined: Secret sharing, oblivious transfer, garbled circuits, fully homomorphic encryption, and functional encryption. These methods are briefly described below. Subsequently, security models are introduced that can be used to categorize the behavior of adversaries. Finally, the Real World/ Ideal World paradigm explains a procedure for testing MPC protocols‘ security.

2.2.1 Secret sharing

Secret sharing schemes refer to methods for distributing a secret among several parties. The basic idea is to share a secret s among n parties in such a way that only the combination of a sufficiently large number of k parts together can reconstruct the secret. Secret sharing is a primitive that forms the core of many cryptographic protocols, especially in MPC protocols. It was invented independently by Adi Shamir [98] and George Blakley [16] in 1979.

Depending on the value of the threshold k, there are three families of secret sharing schemes:

$k = 1$ The trivial case, since the secret can easily be distributed to all n parties. It is also called a shared secret.

$k = n$ The so-called *k-out-of-k* secret sharing scheme, where all parts are needed to reconstruct the secret. An example of this is additive secret sharing.

$1 < k \leq n$ In this case, the secret is divided into n parts, but already k parts are sufficient to reconstruct the secret. An example of this is Shamir's secret sharing [98].

In 1985 Chor *et al*. [30] introduced the concept of verifiable secret sharing. *"The property of verifiability means that shareholders are able to verify that their shares are consistent."*[59] This enables all parties to check that they have received correct shares. This is ensured by commitment procedures [23].

Two well-known schemes are discussed below.

Additive secret sharing

Additive secret sharing [76] is probably the simplest secret sharing scheme. It is an example of the k-out-of-k secret sharing scheme. To split a secret s into n shares $n-1$ field elements $s_1, \ldots, s_{n-1}$ are chosen randomly and s_n is defined as $s_n = s - \sum_{i=1}^{n-1} s_i$. Each party then gets one of these shares. To reconstruct the secret, all shares are summed up.

Shamir's secret sharing

Shamir's secret sharing [98] is one of the first secret sharing schemes. It belongs to the so-called (k, n) threshold schemes, where $1 < k \leq n$ applies. This means that only k of n shares are needed to reconstruct the secret s. The construction of the shares consists of two parts:

1. Construction of the polynomial $f(x) = s + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1}$, where the values for all coefficients $a_1, \ldots, a_{k-1}$ are chosen randomly.
2. Generation of n value pairs $(x_i, s_i = f(x_i))$, for $i = 1, \ldots, n$, the values of all x_i are chosen randomly and differently with the restriction $x \neq 0$. The value pairs are afterward distributed to the parties involved. The values for x_i are public while the s_i, the shares, should be kept secret.

One needs k value pairs to determine the polynomial $f(x)$ uniquely, so up to $k-1$ partial secrets can be compromised without endangering the secret. The original scheme was based on the finite field $\mathbb{F}$ of size p, where p is a prime number, and p is bigger than both s and n. The coefficients $a_1, \ldots, a_{k-1}$ are randomly chosen over the integers in $[0, p)$ and the values $s_1, \ldots, s_n$ are computed modulo p.

2.2.2 Oblivious transfer

"*Oblivious Transfer (OT) is a cryptographic primitive defined as follows: in its simplest flavour, 1-out-of-2 OT, a sender has two input messages M_0 and M_1 and a receiver has a choice bit c. At the end of the protocol the receiver is supposed to learn the message M_c and nothing else, while the sender is supposed to learn nothing.*" [32]

The concept of oblivious transfer was presented in 1983 by Even *et al*. as a protocol for signing contracts [43]. [25, 32, 85] use oblivious transfer to implement efficient 2PC and MPC protocols. According to [65] it is even possible to implement all cryptographic tasks with it.

The protocol in Figure 2.10 [32] serves as an example of an 1-out-of-2 oblivious transfer protocol. Interestingly, the protocol is based on the Diffie-Hellmann key exchange [36] (Figure 2.9), even though this comes from a different cryptography field. Although the Diffie-Hellmann key exchange is widely known, it will now be discussed first, to emphasize the steps needed to transform it into an OT protocol. In general, both protocols require a group $\mathbb{G}$ and a generator g. The key exchange occurs as follows: Alice takes a random sample a, computes $A = g^a$ and sends the result to Bob. Bob does the same and also takes a random sample b, computes $B = g^b$ and sends B to Alice. Both are now able to calculate $g^{ab} = (g^a)^b = (g^b)^a$, from which they can both derive the key k.

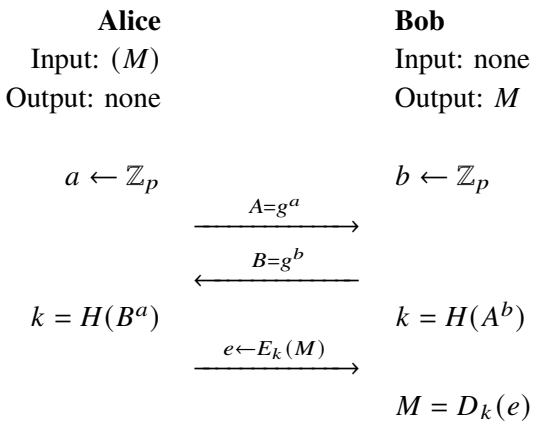

Figure 2.9: Diffie-Hellmann Key Exchange

"The key observation is now that Alice can also derive a different key from the value $(B/A)^a = g^{ab-a^2}$*, and that Bob cannot compute this group element (assuming that the computational DH problem is hard)."* [32]

The possibility that Alice can derive two keys is now utilized in the protocol of Figure 2.10. The first actions of Alice or the sender are identical to the Diffie-Hellmann key exchange. Bob or the receiver, in turn, makes the value B dependent on its input parameter c. Alice then uses k_0 and k_1 to derive different keys in both possible ways, which she then uses for the messages M_0 and M_1, respectively. Bob can only decrypt one of the two messages, depending on c.

2.2.3 Garbled circuits

Circuits, or more precise Boolean circuits, describe a mathematical model for digital circuits in complexity theory. They provide an alternative representation of computations. A garbled circuit, in turn, is a way to "encrypt" a computation. The goal is to reveal only the output or result, but not the input or intermediate results.

According to [52], the idea goes back to Yao, who probably expressed it orally during presentations on [111]. The first written document about garbled circuits, which also proves that any computation can be implemented with garbled circuits, was in [51]. The first mention of the term "garbled circuit" was in [9]. [3] describes a way to garble arithmetic circuits.

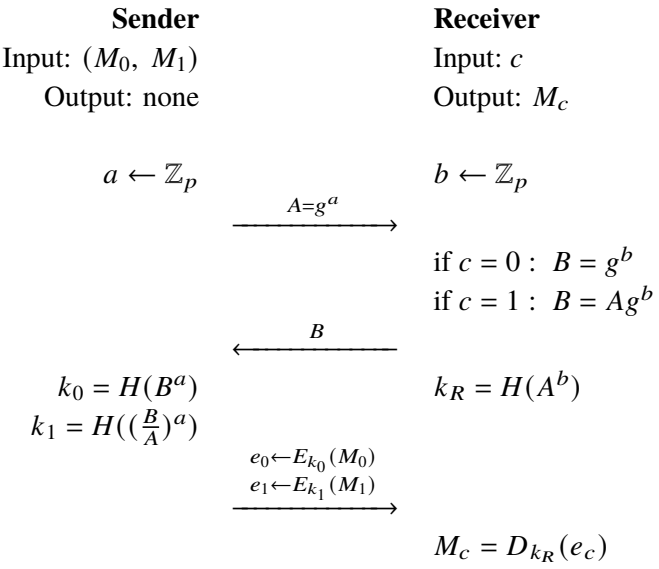

Figure 2.10: "The Simplest Protocol for Oblivious Transfer"[32]

In general, a garbling scheme consists of:

- A way to convert a circuit C into a garbled circuit $\widehat{C}$.
- A way to convert any input x for the circuit into a garbled input $\widehat{x}$.
- A way to take a garbled circuit $\widehat{C}$ and garbled input $\widehat{x}$ and compute the circuit output $C(x)$.

According to [42], Yao's garbled circuit protocol [111] is the best known MPC technique. This protocol will now be used as an example to show the procedure for an AND circuit:

To garble a circuit, the circuit must first be created. This is done using the truth table of the AND function in the first table of Table 2.1. The next step is for the so-called garbler to create a garbled version of this table. To do this, he generates random keys[4] for all possible values of a, b and c. He then encrypts the output values using such a cipher, as shown in the second table in Table 2.1. Then the rows of the truth table of the garbled version are randomly permuted, and the output column is sent to the other person, the so-called evaluator.

Input		**Output**
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Input		**Output**
a	b	c
k_0^a	k_0^b	$E_{k_0^a}(E_{k_0^b}(k_0^c))$
k_0^a	k_1^b	$E_{k_0^a}(E_{k_1^b}(k_0^c))$
k_1^a	k_0^b	$E_{k_1^a}(E_{k_0^b}(k_0^c))$
k_1^a	k_1^b	$E_{k_1^a}(E_{k_1^b}(k_1^c))$

Table 2.1: Truth table of the AND function and the garbled version.

[4]These keys must be able to act as keys for an authenticated symmetric encryption cipher, such as AES.

Now it is the evaluator's turn. He now needs the corresponding input values from the garbler and himself. It is easy to get the value from the garbler because he knows which key matches his value. The evaluator‘s input value is converted to the matching key with the help of an 1-out-of-2 oblivious transfer protocol (Section 2.2.2). Equipped with both keys, the evaluator can now decrypt precisely one of the output columns‘ received rows. Afterward, he can tell the result to the garbler if it is a 2PC.

Of course, more complex circuits are also possible without setting up an overall truth table. Instead, the encoded outputs, such as column c in the second table of Table 2.1, are used as input in the next circuit, and the evaluator gets the output columns of all involved circuits. This nesting allows the realization of circuits of any complexity.

2.2.4 Fully homomorphic encryption

A central task of ordinary encryption schemes is the protection of data. This protection usually includes both protection against unauthorized reading and unauthorized modification of the data. With homomorphic encryption, the protection is now intentionally reduced. The data should still be protected against unauthorized readout, but it should be possible to calculate with the encrypted data without decrypting it. Depending on which and how many functions for the computations are supported by a homomorphic encryption scheme, a distinction is made between partially homomorphic encryption and fully homomorphic encryption. With partially homomorphic encryption, either not all functions are supported, or the functions may not be used as often as desired. This restriction does not apply to fully homomorphic encryption schemes.

Many known encryption schemes such as ElGamal [41] or RSA [92] in the textbook variant belong to the partially homomorphic encryption. With both, it is possible to multiply the encrypted plaintext, the cipher rate, with other cipher rates that were encrypted with the same key to obtaining the encrypted product of both plaintexts. For the real use in the application scenarios mentioned initially, we try to eliminate the homomorphism, which is, *e. g.* achieved with RSA by using padding schemes like optimal asymmetric encryption padding (OAEP) [12]. For this reason, there are various cryptosystems in the area of partially homomorphic encryption. A first hypothesis that fully homomorphic encryption might be possible was made by Rivest in 1978 [91]. The first fully homomorphic encryption scheme was published in 2009 by Craig Gentry [49] and implemented in 2010 [50].

To be able to represent all functions, addition and multiplication must be available as homomorphic operations. In [48], Gentry describes the generation of a fully homomorphic encryption scheme based on a symmetric encryption scheme. It handles addition, subtraction, and multiplication as homomorphic operations, thus fulfilling a fully homomorphic encryption scheme‘s requirement. When applying homomorphic operations to encrypted contents; however, noise is generated, which means that at some point, depending on the strength of the noise, the value can no longer be decrypted correctly. Multiplication increases the noise more than addition and subtraction. To use fully homomorphic encryption with the limitation of this scheme, an operation called bootstrapping is introduced. Bootstrapping removes noise from the cipher by re-encrypting it. Bootstrapping is so essential that other published full homomorphic encryption schemes are based on it [18, 20, 21, 22, 28, 45, 74].

For the use of fully homomorphic encryption as an MPC protocol, each party must be able to encrypt its input so that the other parties cannot decrypt it. However, at the same time, it must be possible that all inputs can be used in a computation. For this use case, Lopez-Alt 2013 has introduced a new category of encryption scheme, *multi-key fully homomorphic encryption* [74], which allows the realization of k-out-of-k secret sharing schemes. In 2018 *threshold multi-key fully homomorphic encryption* [8] was introduced to realize MPC protocols where the required number of active computation parties is smaller than the total number of parties involved.

2.2.5 Functional encryption

Asymmetric encryption, also known as public-key encryption [36, 92], allows different keys to be used for encryption and decryption. The public key can be public, and all data encrypted with its help can be decrypted with the corresponding secret key. Thus, secure public-key encryption ciphers ensure that one has either full access or no access to the plaintext, depending on whether one has the secret key or not.

Functional encryption describes a generalization of public-key encryption. It provides fine-grained access control by creating functions that return the function‘s result on ciphertexts as plain text and no other information. For example, it is possible to implement a document management software for encrypted documents that can read metadata from the encrypted documents and display them to a user using functional encryption. Functional encryption was formally specified in 2011 [17].

In order to create an MPC protocol based on functional encryption, it must, of course, be possible for each party to encrypt their inputs and to perform the desired computation on all encrypted inputs. The easiest way to achieve this is to have a trusted, neutral party[5] do the encryption. An improvement to this method is Multi-Client Functional Encryption [53, 56], which still requires a neutral party to create a master secret key. However, it allows all parties to encrypt their input independently. Based on this, Decentralized Multi-Client Functional Encryption [31] was presented, which allows the functionality without a central party.

2.2.6 Security models

To be able to compare the security of protocols, it is common practice to measure security using uniform security models. Two representatives of such security models will now be briefly introduced below: Semi-honest security and malicious security.

Semi-honest security

“A semi-honest adversary is one who corrupts parties but follows the protocol as specified.” [42] This type of adversary is often referred to as passive or honest-but-curious. This level of security prevents unintentional leakage of information between cooperating parties. Therefore, this security level of MPC protocols should at least be achieved. In contrast to malicious security, protocols can often be implemented efficiently in the semi-honest model.

[5] If such a party exists, one could generally reduce the computation effort by having this party do the entire computation and send the results to all peers.

Malicious security

"A malicious (also known as active) [adversary] may instead cause corrupted parties to deviate arbitrarily from the prescribed protocol in an attempt to violate security." [42] In addition to passively reading the log, a malicious adversary can also change any input. Therefore protocols of this security level need to be able to verify any input so that one will find verifiable secret sharing rather than simple secret sharing here. If a protocol can resist such adversaries, a high level of security is achieved.

2.2.7 Real World/Ideal World paradigm

There are different ways to ensure the security of cryptographic protocols. One way is to create a list of all relevant attacks and check the protocol for them. However, such a test result only provides information about the protocol‘s security against already known attacks. Another possibility is a mathematical proof, in which the protocol is reduced to its mathematical foundations, and the proof of the mathematical foundations is generalized to the protocol. With MPC protocols, it is often impossible to formalize the involved party‘s knowledge and the protocol‘s correctness for such proof. On the one hand, it cannot be claimed that the parties involved do not learn anything from the computation, because the output should be made available to all parties involved. On the other hand, the correctness of the output cannot be guaranteed, since it depends on the input, but maybe corrupted if there is at least one adversary. Thus, a new approach was established, the Real World/Ideal World paradigm [7, 54].

The Real World/Ideal World paradigm consists of two concepts of the ideal world and the real world. In the ideal world, secure computation is performed by a trusted, neutral party. The input from each party is communicated directly to the neutral party, and the output computed by the neutral party is communicated to everyone. Since this neutral party is trusted, it cannot be taken over by an adversary, and it does not falsify any input. The adversary can corrupt only the other parties. Since such a scenario is improbable, it represents the ideal world and only serves as a benchmark for the real world.

In the real world, there is no trusted party. Instead, this trusted party is represented by a cryptographic protocol. Adversaries can corrupt the parties. If this happens at the beginning of the protocol, it corresponds to ideal world‘s attack scenario. A cryptographic protocol is considered secure if any damage that an adversary can do in the real world is also possible in the ideal world.

3 Related Work

This chapter describes three sub-fields of ML that have the same goal: Create a model using distributed data. Each sub-field sets different priorities.

Distributed ML	How can a centralized model with central data be trained as quickly and efficiently as possible?
Split learning	How to train a common model without sharing training data?
Federated optimization	How can a centralized model be trained with massively decentralized and unbalanced data?

3.1 Distributed machine learning

Distributed machine learning also called distributed optimization describes techniques used to train a model with central data on distributed computing resources. The computing resources can vary from several GPUs to computing clusters with hundreds of GPUs or more. In the beginning, distributed machine learning was necessary because computing resources were very limited. Nowadays, these approaches are further developed to train more complex models with much more data. For example, the Turing Natural Language Generation is a language model with 17 billion weights introduced by Microsoft in February 2020 [95].

Over the years, three strategies for distributed optimization have emerged:

1. Data parallelism
2. Model parallelism
3. Pipeline parallelism

Additionally, recent approaches combine the advantages, which will be described in Section 3.1.4.

3.1.1 Data Parallelism

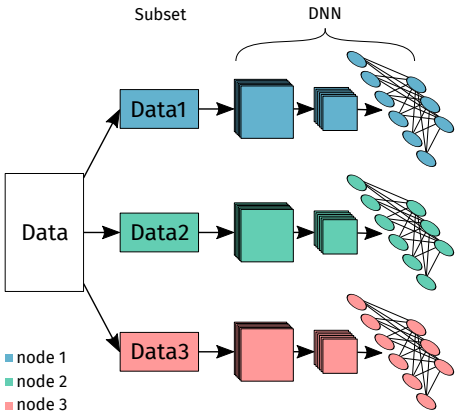

Figure 3.1: Data Parallelism

Data parallelism means to divide the training data into several computing units, as shown in Figure 3.1. This method is probably the most frequently used strategy, as it is already known from other computer science areas. The problem can be summarized in the following two questions:

1. What is the best way to distribute the data?
2. What is the best way to bring the partial results together again?

Hsu *et al*. [11] describe two basic approaches for distributing training data to multiple nodes. The so-called instance-based sharding and the so-called feature-based sharding, as depicted in Figure 3.2. Instance-based sharding means that the whole training data is split to the number of nodes. Thus, the load per node is automatically reduced because only a fraction of the data needs to be processed and stored on each node. The other way is to split the training data in terms of its features. While this does not reduce the number of instances per node, it does reduce the number of features associated with an instance for each node and, thus, the amount of data associated with each instance. This increases the potential throughput per node. Instance based sharding is the more common strategy.

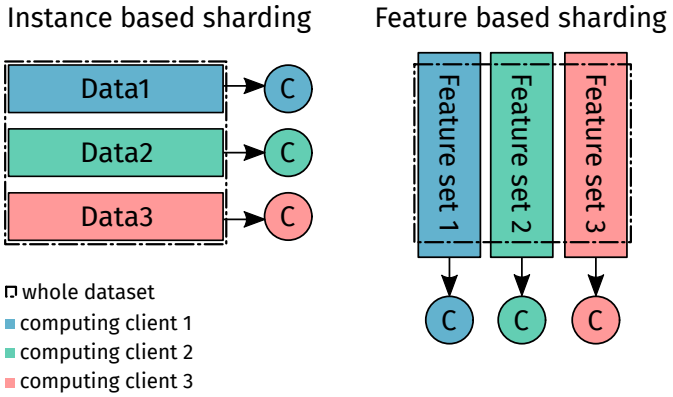

Figure 3.2: Two approaches to data splitting. Left: instance shards. Right: feature shards. [11]

According to [77], there are three common methods to train and combine the data parallelism‘s partial results:

Distributed gradient computation method

If the training data is identically and independently distributed (IID), the weights‘ gradients can be calculated in parallel for each shard‘s iteration. The different gradients can then be used to calculate the exact global gradient on a single machine. Then the optimization step, weights‘

update, and distribution of the new weights to the nodes can be performed. These computations can be performed using map-reduce [33], where each iteration consists of a map phase to calculate the gradients and the corresponding reduce phase for the update step [77].

Majority vote method

The first phase of this method is identical to the previous method. Each node uses its training data to train its model. The models trained in this way are then used for classification by using the result of the majority vote based on the trained models for an input value [77].

Mixture weight method

This method is an optimization of the majority vote method. In contrast to the majority vote method, the respective weights‘ mean values are determined centrally and are distributed again to all nodes before the next epoch. The weights determined in this way can also be used directly for classification [77].

The significant disadvantage with data parallelism is its poor memory efficiency. The entire model and optimizer state must be replicated across all nodes involved. According to [88], the largest model that data parallelism can run has less than 1.5 Billion weights, assuming 32GB GPUs[1]. Although not very large models are supported, with data parallelism, it is potentially possible to train faster with the same amount of training data or to train with more training data for the same training duration.

3.1.2 Model Parallelism

In model parallelism, single or multiple layers of the DNN are divided and distributed to different nodes, as shown in Figure 3.3. Thus, each node has neurons from each layer so that training can be done in parallel.

This approach is much more memory efficient than data parallelism. However, the communication effort increases strongly, since the distributed subareas of the network partly need other subareas‘ results (activations) as input data, *e. g.* dense layers (see Section 2.1.5). In [88] it says:

“We tested a 40B parameter model using Megatron-LM across two DGX-2 nodes and observe about 5 T flops per V100 GPU (less than 5% of hardware peak).”

Another big drawback is that model developers often have to adapt their model specifically to model parallelism to support such splitting, while data parallelism works out of the box.

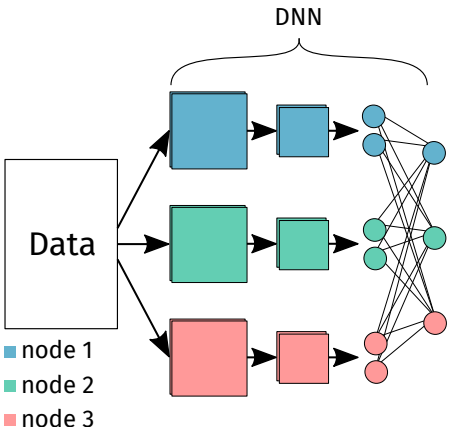

Figure 3.3: Model Parallelism

[1] Represents the current state of the art.

3.1.3 Pipeline Parallelism

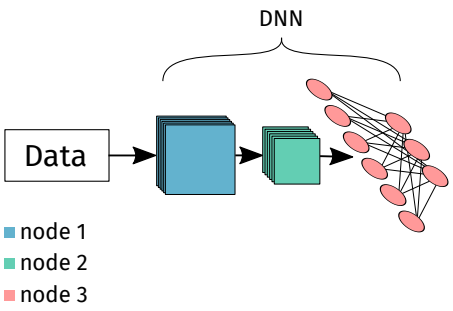

Figure 3.4: Pipeline Parallelism

In pipeline parallelism, the DNN is divided so that individual or successive layers are distributed to the nodes, as shown in Figure 3.4. In practice, it is also combined with data parallelism so that individual slices can occur several times.

An advantage of this parallelization form is that the communication between nodes is limited to the slice boundaries‘ activations and gradients. Additionally, only the corresponding layer(s) weights have to be kept and updated, which can reduce memory round-trips [13]. The disadvantage is that it is challenging to use all nodes equally to utilize the entire system fully. Additionally, the latency increases proportionally with the number of nodes [13].

3.1.4 ZeRO

ZeRO is a technique for optimizing the memory in data parallelism. It promises to combine the advantages of the previously listed strategies while eliminating the disadvantages. As a result, larger models can be trained with the same memory consumption. More about this at the end of this section. ZeRO stands for Zero Redundancy Optimizer and is developed at Microsoft [88]. Rajbhandari *et al*. divide the poor memory efficiency of data parallelism into two areas, which they improve with different techniques:

ZeRO-DP optimizes the storage of model and optimizer states.

ZeRO-R optimizes the residual memory usage, which includes activation, temporary buffers, and memory fragmentation.

The techniques described by ZeRO can already be used in practice as they are part of DeepSpeed [34], a DL optimization library based on PyTorch[2].

ZeRO-DP

ZeRO-DP increases the efficiency of the required memory footprint by partitioning optimizer states, gradients, and weights to the individual nodes. The optimization consists of three stations: optimizer state partitioning, gradient partitioning, and weight partitioning. In each of these partitions, the associated memory area is distributed evenly among the individual computing units, which results in an approximately linear distribution. At the same time, the total communication volume increases to 1.5 times. Furthermore, it is stated: *“ZeRO powers DP to fit models with arbitrary size as long as there are sufficient number of devices to share the model states.”* [88]

[2]PyTorch is a machine learning framework initiated by Facebook [44]

ZeRO-R

ZeRO-R covers three techniques to keep the remaining memory footprint as small as possible:

Partitioned Activation Checkpointing
Model parallelism requires replication of the activations, which leads to redundancies across the nodes. ZeRO-R eliminates these redundancies by partitioning the activations and materializing them only where they are needed for the computation. Besides, these activation checkpoints can be swapped from the GPU–RAM to the CPU–RAM, reducing the memory footprint to nearly zero.

Constant Size Buffers
Some high-performance libraries, such as NVIDIA Apex, merge weights into large buffers during training to provide high bandwidth for mass operations. This increases memory consumption, *e. g.* a 32-bit buffer for 3B weights requires 12GB of memory. To achieve a good price-performance-ratio between memory consumption and performance, ZeRO-R uses a constant size buffer for these operations when the model becomes too large.

Memory Defragmentation
A separate memory management for temporary data was created to use as much memory as possible. This avoids out-of-memory terminations due to a lack of continuous memory.

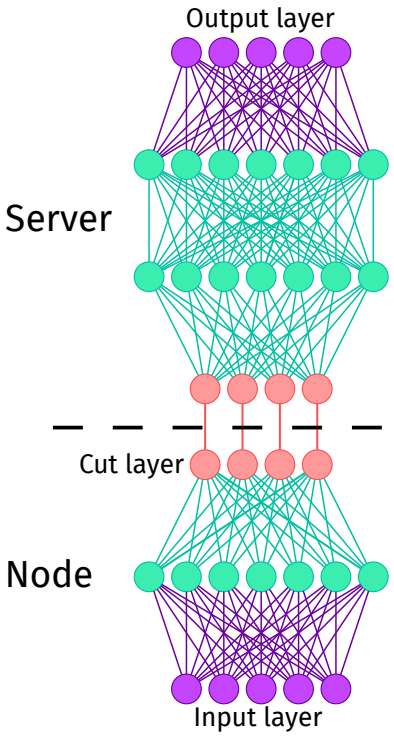

Figure 3.5: Basic split learning setup showing distribution of layers across node and server

3.2 Split learning

Split learning (SL) [57, 87, 105, 106] is another approach to train a common model without sharing raw training data. As shown in Figure 3.5, this approach splits the model into a server and a node part, hence its name. The last layer of the node part is called the *cut layer*. Figure 3.6 shows a schematic illustration of one learning round with a single node. Before the first start, the server sends the topological description of the model‘s node part to the node. The basic procedure for SL is that a node starts the forward pass with a sample. The resulting tensor of its last layer, the *cut layer*, is sent to the server. The server, in turn, uses this to train its model part. During the backward pass, the server performs backpropagation as usual and, in the end, sends the gradient tensor of its first layer to the node. The node then uses the obtained gradient tensor for its backpropagation, so that at the end, a complete training round was executed. Depending on the network latency, the training will be significantly slower than it takes place entirely locally.

For adaptation to multiple nodes, two modes are offered, one is called “centralized mode” while the other is called “peer-to-peer mode”. In centralized mode, the nodes communicate exclusively with the server. After each round, the node uploads its weights to the server to be made available to other nodes. In peer-to-peer mode, the server tells new nodes the last trained node‘s address to check out the current weights. Besides, the weights are shared between the nodes [100], but unfortunately, this is not explained in detail. In both modes, the restriction is that only one node can train at a time.

In order for the training to be successful, the nodes must tell the server the expected label. Possible solutions without label sharing that require another network roundtrip are also covered in [106].

One may call SL a combination of data and pipeline parallelism since the nodes can train with different data (data parallelism). The model is divided into two partitions one node and one server (pipeline parallelism).

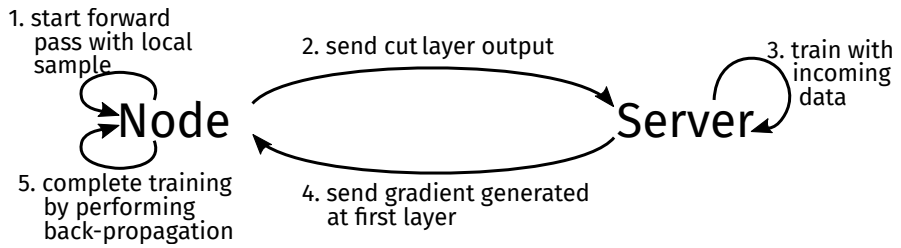

Figure 3.6: A schematic illustration of a split learning round with a single node.

To reduce the risk of leakage of input data during transmission between node and server, SL was further developed into NoPeekNN [105]. NoPeekNN involves adapting the neural network to use two loss functions, distance correlation ($DCOR$) on the node and categorical cross-entropy (CCE) on the server. Distance correlation generalizes the correlation R of two tensors in the following ways [102]:

1. $R(X, Y)$ is defined for X and Y in arbitrary dimensions;
2. $R(X, Y) = 0$ characterizes independence of X and Y

Thus, in this case, using distance correlation as a loss function between training data and the result of the cut layer optimizes the weights so that both tensors are as independent as possible. The total loss function for a sample X, cut layer result $\hat{Z}$, true label Y, and predicted label $\hat{Y}$ is

$$\alpha_1 DCOR(X, \hat{Z}) + \alpha_2 CCE(Y, \hat{Y})$$

α_1 and α_2 are scalar weights to control the ratio of both functions. The greater the influence of distance correlation, the more independent input data, and transmitted data will be. The consequence of this inserted noise is that, depending on the size of α_1, significantly more training is required to achieve an accuracy comparable to that without noise.

3.3 Federated optimization

FO is an approach to create and improve a common model with decentralized data. It was developed at Google and was originally intended for the further training of models on mobile devices [27, 67, 68, 69, 81, 82]. FO is used in GBoard[3] for word and emoji prediction [26, 58, 89]. FO's optimization problem is designed for specific key characteristics, which differentiate it from distributed learning:

Limited communication By focusing on mobile devices, network connectivity, and the available battery capacity may be limited. Therefore only one communication round per day is expected.

Non-IID data Training data is typically very user-dependent, so a particular node's data will not represent all nodes.

Unbalanced data The amount of training data varies significantly between nodes. Some will have a multiple of the data available to others.

Massively distributed It is assumed that the number of nodes will be much larger than the average amount of training data.

Algorithm 3.1 `FederatedAveraging`. The K nodes are indexed by k; B is the local minibatch size; E is the number of local epochs; η is the learning rate. [81]

function FedAvg // Executed on server
    initialize w_0
    for each round $t = 1, 2, \ldots$ **do**
        S_t = (random set of $max(C \cdot K, 1)$ nodes)
        for each node $k \in S_t$ **in parallel do**
            $w_{t+1}^k \leftarrow NodeUpdate(k, w_t)$
        end for
        $w_{t+1} \leftarrow \Sigma_{t=1}^{K} \frac{n_k}{n} w_{t+1}^k$
    end for
end function
function NodeUpdate(k, w) // Executed on nodes k
    for each local epoch i from 1 to E **do**
        batches $\leftarrow$ (data P_k split into batches of size B)
        for batch b in batches **do**
            $w \leftarrow w - \eta \nabla l(w; b)$
        end for
    end for
    return w to server
end function

Algorithm 3.1 shows the basic algorithm of FO called `FederatedAveraging`. The single central server determines random nodes for each round. Each node trains the model locally with its data and then sends the updated model weights to the server. The server computes the average values from

[3] GBoard is a keyboard application for iOS and Android

the weights received and thereby updates his model. For this algorithm to work, the common model must be distributed to the nodes regularly, if possible, before a new round begins. A schematic illustration of one round is shown in Figure 3.7.

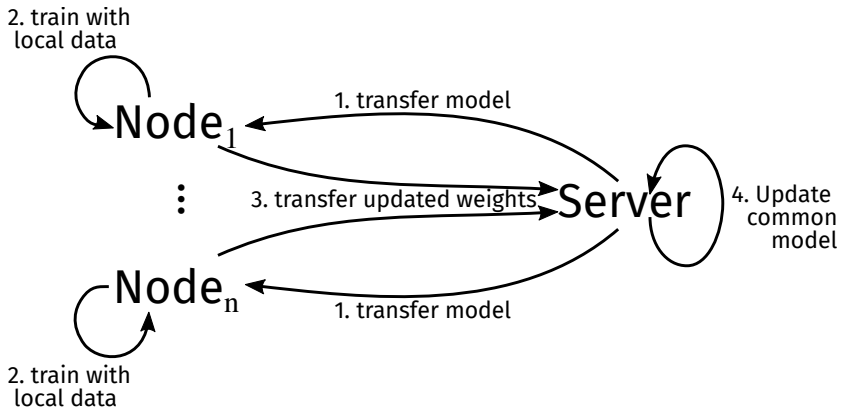

Figure 3.7: A schematic illustration of one federated optimization round. It consists of four phases: 1. Transfer the central model from server to all participating nodes. 2. The nodes train the model with their local data. 3. The updated weights are sent to the server. 4. The server updates the common model by averaging the weights.

This algorithm provides higher data privacy even in this basic version than the presented distributed learning approaches because the information transmitted between nodes and server is limited to the updated weights. This means that neither the training data is collected in a central location, nor does the training data leave the device.

In 2006, a new mechanism called *Differential Privacy* [39, 40] was introduced to improve privacy protection. More precisely, the goal is formulated with: *"the risk to one's privacy [· · ·] should not substantially increase as a result of participating in a statistical database."* [39]

In [80], the basic FO approach was not only extended by the application of differential privacy but additionally, an extension of TensorFlow called TensorFlow Privacy [2] was created, which allows a straightforward application of these principles. At about the same time, another approach was presented, private FO [15], which combines FO with differential privacy. Private FO uses differential privacy not only on the server-side but also in a modified form called separated differential privacy on the nodes.

In recent years, so-called adaptive optimizers have become increasingly popular, as they often converge faster than stochastic gradient descent (SGD). Adaptive in this context means that the learning rate can be adapted dynamically per weight. For example, the learning rate can be increased if the gradient had the same sign twice. With [90], an approach for adaptive federated optimization has now been introduced. A generalized version of FedAvg is derived, which then allows different optimizers for node and server-side.

4 Analysis & Design

Distributed learning, FO, and SL solve the problem of creating a common model in different ways. A short comparison is shown in Table 4.1. Distributed learning is primarily about faster training times within a data center. FO and SL both try to enable federated learning for any data distribution through different approaches. However, since both rely on a centralized communication model, it is impossible to implement the initial scenario: Parties, *e. g.* competitors, to create a common model without one party gaining more advantages over the others.

	Distributed learning	Split learning	Federated optimization
Communication architecture	Client-server	Client-server and peer-to-peer	Client-server
Data distribution	Balanced and IID	Balanced/ Unbalanced and IID/ non-IID	Balanced/ Unbalanced and IID/ non-IID
Main purpose	Distribute to train models faster with centralized training data	Create a common model without sharing training data	Create a common model without sharing training data

Table 4.1: Comparison of distributed learning, federated optimization and split learning

The concepts of SL and FO can serve as an interesting basis for a decentralized communication model. The challenge is to replace the central server with something decentralized and then examine such a change‘s implications. The following sections, therefore, show how both approaches can be implemented without a central server. Thus, the approaches‘ communication behavior is first analyzed to then present alternative decentralized communication models and finally analyze them.

4.1 Decentralized communication approach for split learning

4.1.1 Analysis of split learning

On closer inspection of SL, it is noticeable that two different communication channels are used:

1. A bidirectional connection between the client that is currently training and the server (client-server).
2. A unidirectional connection between the current training client or last trained client with at least one other client to propagate the new weights (peer-to-peer).

The bidirectional network connection during training has a significant negative impact on the training duration because SL has to exchange data between client and server for each pass (forward and backward), *i. e.* for each training sample twice per training epoch. Divided according to the respective passes, the additional work involved in splitting the model can be described by the following processes:

Forward pass:
- Serialization of the data from the client GPU.
- Network transport to the server.
- Deserialization of the data on the server-side.

Backward pass:
- Serialization of the data from the server GPU.
- Network transport to the client.
- Deserialization of the data on the client-side.

Due to the latencies arising in that way, the client cannot be used to full capacity for the entire duration of the training, which results in unused resources and increased time consumption. These processes would be omitted by switching to a decentralized communication approach and the associated elimination of the central server.

There are also the following challenges for which no concrete solution has been described:

- System-related only one training session may take place at a time.
- After each training session, the new valid model must be communicated to all peers.
- New peers must be provided with the currently valid model.

4.1.2 Design of decentralized version

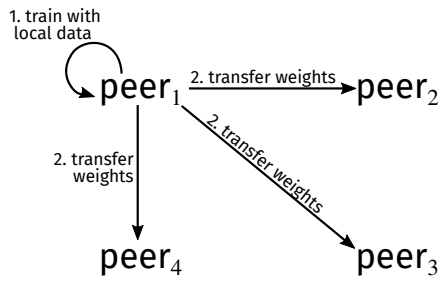

Figure 4.1: The distribution strategy when one peer sends the updated weights to all others.

Without a central server, each client or preferably peer would have to train the entire model locally and then propagate all weights to the other peers. This change eliminates the basic idea of SL, the splitting. Instead, it is the continuous training of a model. The training of an already pre-trained model is not a new idea; it belongs to an area called transfer learning [103]. The difference is that Transfer Learning usually uses models that have been trained with huge general datasets to use such pre-trained models for faster training for a specific task [5, 99].

Different strategies are conceivable for the propagation of weight values after training, whereby two strategies are probably the most obvious. Either the weights are propagated from the peer who has just finished training to all others, as shown in Figure 4.1, or the weights are always propagated to the next peer, comparable to a bucket chain (Figure 4.2). The second strategy distributes the necessary communication effort among all peers.

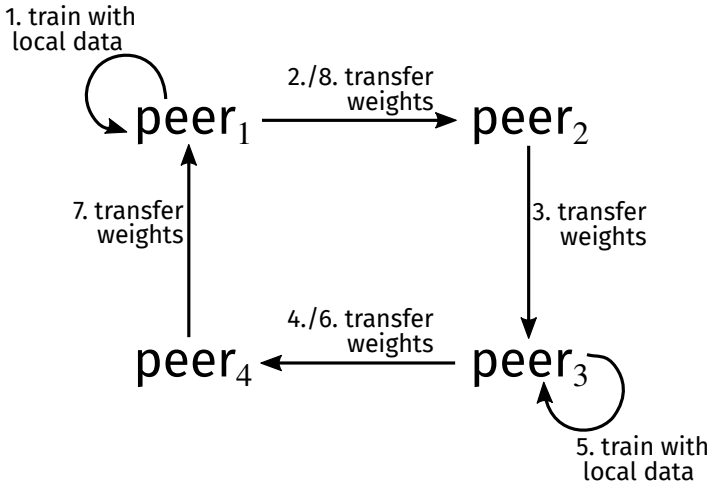

Figure 4.2: A schematic illustration of peer-to-peer learning with four peers. It consists of two phases: 1./5. One peer trains the currently valid common model locally. 2.-4./6.-8. The updated weights are propagated between all peers.

4.1.3 Analysis of decentralized version

By eliminating the communication between client and server, the training times should decrease significantly. The communication effort between the peers would increase compared to SL since now all weights must be communicated after training has been completed and previously, only those of the client-side model.

One way to ensure that not more than one training session can take place at the same time is token passing. A message, the so-called token, would always circulate between all peers, and the peer who currently "owns" the token would be allowed to train. In combination with the bucket chain distribution approach (Figure 4.2), it has the advantage that such a ring between peers must also be maintained so that the already established communication channels can be reused.

From a privacy perspective, this decentralized approach does not differ much from the original approach. In SL, the client-side weights were already shared with all other participating clients before, which has now been extended. In return, the necessary trust for the operator of the central server is eliminated. For this reason, attacks that attempt to reveal training data, such as Generative Adversarial Model INversion (GAMIN) [1], are still possible. Especially for the first training parties, there is an increased risk in the initial period. Other parties who have not yet trained could draw conclusions from the training data without having already exposed themselves to this risk.

4.2 Decentralized communication approach for federated optimization

4.2.1 Analysis of federated optimization

Figure 4.3 shows the abstract processes of FO, regardless of the technology or communication structure used. Each client[1] trains with its local data. With FO, the central server is used for steps 2 and 3; therefore, there is only one network connection between each client and the server. Besides, the clients do not know each other since they communicate exclusively with the server.

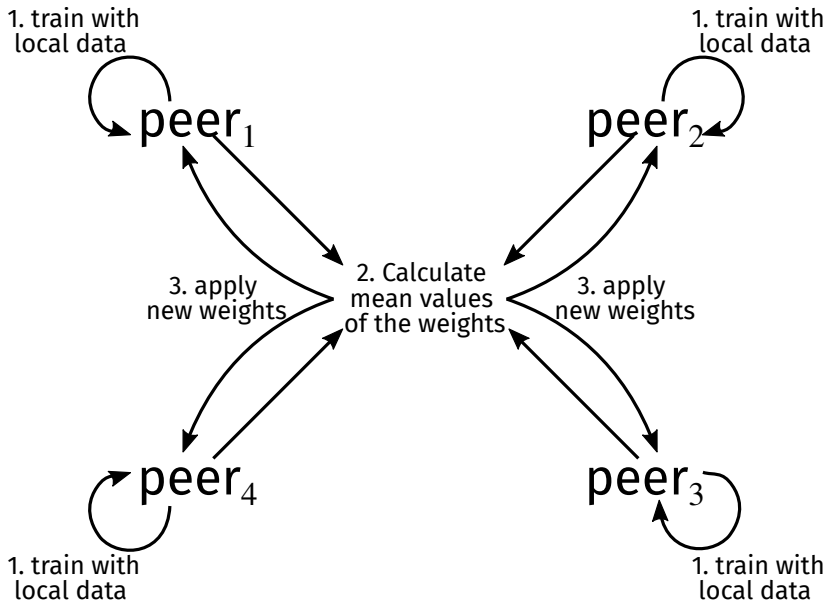

Figure 4.3: A schematic illustration of the abstract processes of FO with four peers. It consists of three phases: 1. All peers train the currently valid common model locally. 2. The mean values of the weights are calculated, *i. e.* Algorithm 4.1. 3. The updated weights are applied.

4.2.2 Design of decentralized version

In order to replace the function of the central server with a decentralized technology in FO, it is necessary to map the functionality of `FedAvg` differently. When converting to a decentralized communication model, steps 2 and 3 of Figure 4.3 must be implemented accordingly.

The easiest way to do this would be that all parties communicate their weight values after the training. This would allow each individual to calculate the average value independently. This method‘s problem is similar to the decentralized method described in Section 4.1 or SL in general. After the first communication round, attacks like GAMIN [1] are ideal because one gets all parties‘ pure weights.

[1] referred to as peer in the figure

The data privacy can be increased by incorporating MPC (Section 2.2). MPC allows a group to perform a common computation without one member's data becoming known to the other peers. Thus it fulfills precisely the use case desired here: The mean values of all weights are calculated without the concrete weights being known. For this reason, the algorithm `SecAvg` (Algorithm 4.1) was developed in the context of this work. It is applying a slight modification of the presented additive secret sharing scheme (Section 2.2.1).

To share a secret with p parties, the presented secret sharing scheme chooses $p-1$ field elements $s_1, \ldots, s_{p-1}$ randomly, and the last element s_p is defined as $s_p = s - \sum_{i=1}^{p-1} s_i$ [76]. Instead the chosen scheme here generates p random numbers $rn_1, \ldots, rn_p$ and normalizes them by dividing through their sum $p\_rn_i = \frac{rn_i}{\sum_{k=1}^{p} rn_k}, i \in \{1, \ldots, p\}$. The resulting percentage distribution is then used to generate the actual field elements $s_1, \ldots, s_p$, where $s_i = s * p\_rn_i, i \in \{1, \ldots, p\}$.

`SecAvg` now uses this secret sharing scheme to calculate the mean values. For distributing the computation, it benefits from the fact that a sum can be formed from the sum of its subtotals:

1. Each party divides its weights into p parts using the secret division scheme, sends one part to another party, and keeps one part.
2. Each party then totals the received parts and the retained part and sends this partial sum to all other parties.
3. Each party adds up all partial sums and divides the result by p.

In the following, it is now shown that the computations of the algorithm are correct.

Lemma 1
The function `Divide` *of Algorithm 4.1 divides the parameter w so that* $w = \sum_{i=1}^{n} par\_w_i$ *applies.*

Proof of lemma 1

$$
\begin{aligned}
w &= \sum_{i=1}^{n} par\_w_i \\
&= \sum_{i=1}^{n} w * p\_rn_i && \textit{(lines 27-29)} \\
&= \sum_{i=1}^{n} w * \frac{rn_i}{\sum_{j=1}^{n} rn_j} && \textit{(lines 23-25)} \\
&= w * \sum_{i=1}^{n} \frac{rn_i}{\sum_{j=1}^{n} rn_j} \\
&= w * \sum_{i=1}^{n} \frac{rn_i}{rn_1 + \ldots + rn_n} \\
&= w * \frac{rn_1 + \ldots + rn_n}{rn_1 + \ldots + rn_n} \\
&= w * \frac{1}{1} \\
&= w
\end{aligned}
$$

□

Algorithm 4.1 A MPC version for averaging values

Input: n = number of peers; wt = the weights tensor

function SecAvg(n, wt)
  par_wt = a tensor with dimension $dim(wt) + 1$
  for $w \in wt$ **do**
    par_wt.append($Divide(w, n)$) // Divide each weight into n randomly large fractions
  end for
  for $i = 2$ to n **do** // Send a fraction of all weights to a peer, assuming that you are $peer_1$
    Send par_wt_i to $peer_i$
  end for
  Save all received fractions of the other peers and par_wt_1 in f_wt
  avg_wt = a tensor with dimension $dim(wt) + 1$
  for $f\_w \in f\_wt$ **do**// Calculate the average value from the known fractions of each weight
    avg_wt.append($\frac{\sum_{i=1}^{n} f\_wt_i}{n}$)
  end for
  Send avg_wt to each peer
  Save all received average values of the other peers and avg_wt in a_wt
  for $i = 1$ to $|w|$ **do** // Sum up all average values received per weight
    $w_i = \sum_{j=1}^{n} a\_wt_j$
  end for
  return w
end function

function Divide(w, n)
  rn = Array of n random numbers $(rn_1, \ldots, rn_n)$
  p_rn = Array of length n $(p\_rn_1, \ldots, p\_rn_n)$
  for $i = 1$ to n **do**
    $p\_rn_i = \frac{rn_i}{\sum_{k=1}^{n} rn_k}$
  end for
  par_w = Array of length n $(par\_w_1, \ldots, par\_w_n)$
  for $i = 1$ to n **do**
    $par\_w_i = p\_rn_i * w$
  end for
  return par_w
end function

Theorem 2
Algorithm 4.1 calculates the arithmetic average of all given weights wt.

Proof of theorem 2 1
The loops in Algorithm 4.1 perform all operations on all weights. For this reason, only one weight w_i is discussed below.

The arithmetic average of n values $(w_{11}, \ldots, w_{1n})$ *is determined by* $\frac{1}{n} * \sum_{i=1}^{n} w_{1i}$.

$$
\begin{aligned}
w_i &= \sum_{j=1}^{n} a\_wt_j && \textit{(line 17)} \\
&= \sum_{j=1}^{n} \sum_{p=1}^{n} \left(\frac{\sum_{k=1}^{n} f\_wt_k}{n}\right)_{peer_p} && \textit{One fraction per peer (line 12)} \\
&= \sum_{j=1}^{n} \left(\left(\frac{\sum_{k=1}^{n} f\_wt_k}{n}\right)_{peer_1} + \ldots + \left(\frac{\sum_{k=1}^{n} f\_wt_k}{n}\right)_{peer_n}\right) \\
&= \frac{1}{n} * \left(\left(\sum_{k=1}^{n} f\_wt_k\right)_{peer_1} + \ldots + \left(\sum_{k=1}^{n} f\_wt_k\right)_{peer_n}\right) \\
&= \frac{1}{n} * \big(\left(par\_wt_{1_{p1}} + \ldots + par\_wt_{n_{p1}}\right) \\
&\qquad + \left(par\_wt_{1_{pn}} + \ldots + par\_wt_{n_{pn}}\right) \big) \\
&= \frac{1}{n} * \big(\left(par\_wt_{1_{p1}} + \ldots + par\_wt_{1_{pn}}\right) \\
&\qquad + \left(par\_wt_{n_{p1}} + \ldots + par\_wt_{n_{pn}}\right) \big) \\
&= \frac{1}{n} * \sum_{i=1}^{n} f\_wt_i && \textit{According to lemma 1} \quad \square
\end{aligned}
$$

4.2.3 Analysis of decentralized version

The communication effort is increased by the conversion to a decentralized communication model. A client in the centralized scenario must send all n weights to the server once after the training to calculate the average values. While in the decentralized scenario, the weights are sent $p - 1$ times per party, where p is the number of parties, in the simple case. In SecAvg, even $2 * (p - 1)$ because the computation of the mean value is split. On the receiving side, it is the same.

From a privacy perspective, the previously required trust for the operator of the central server is no longer needed. In the simple case[2], attacks such as GAMIN [1] have similar risks to SL because the pure weights are transferred. These risks do not apply when using SecAvg. Besides, all peers involved must now know each other.

If one recalls the Real World/Ideal World paradigm presented in Section 2.2.7, one will notice that FO's previous approaches describe the ideal world case. Basically, the central server corresponds to the trusted, neutral party in a distributed computation. The computations of the mean values are represented in the real world by SecAvg. Therefore, a security assessment using the Real World/Ideal World paradigm must highlight the additional information that can be obtained when using SecAvg. When considering the ideal world, each party receives only the weight averages, *i. e.* the computation results. When considering the real world with the help of SecAvg, the parties, of course, also receive the weight averages as results. They also receive a share of each weight value

[2]The operator of the central server has the same attack possibilities as long as no private FO [15] is used.

and the partial sums formed from each party. The secret sharing scheme used divides the weight values randomly, so the underlying weight value can only be reconstructed from all parts, which is also the essential requirement of any k-out-of-k secret sharing scheme. The summands involved cannot be reconstructed from a sum either, so the communicated subtotals do not provide any further information about a party's weight values. Although all parties receive more information than in the ideal world case, this additional information does not allow any further conclusions about the actual secrets (the weight values).

However, this statement is only valid for semi-honest security, *i. e.* when all involved parties follow the protocol. In the case of malicious adversaries, the presented MPC protocol is not secure enough. The distributed partial weights cannot be verified, and therefore neither can the computed partial sums. For this reason, there is not sufficient protection against such adversaries.

5 Implementation

The primary purpose of prototype development was to be able to test the considerations made so far. This means that the prototypes described here are not directly designed to learn together across network boundaries. However, instead, the focus is on good traceability of the results, including all intermediate results. The prototype was developed in Python. The complete source code can be found in Appendix B.2.

When implementing Algorithm 4.1, it had to be determined how the tensors would be transferred. The decision was made that any exchange should be file-based. This has the advantage, among other things, that all intermediate results can be traced directly as long as the files are not deleted. Hierarchical Data Format version 5 (HDF5) is to be used as the exchange format. HDF5 is a manufacturer-neutral, self-describing format that is also directly supported by TensorFlow. A further advantage of using file exchange is that it also works across PC boundaries if all peers use a common network share.

A naive implementation of Algorithm 4.1 would call the function `divide` for each weight individually. Therefore, for 1.6 million weights[1] and 4 peers, the function will generate 1.6 million times 4 random numbers to divide each weight into 4 parts. This implementation took 15 minutes on an Intel Core i7-4790 at 4GHz. With the simulation of all 4 peers on a single machine, it was several hours.

Listing 5.1 The implementation of `Divide` in Python

```
1 def divide(weights, peers, rnd_gen):
2     rn = [ rnd_gen.integers(1, 5, peers) for _ in range(len(weights))]
3     for i in range(len(rn)):
4         rn[i] = rn[i] / rn[i].sum()
5     rn = np.array(rn).transpose()
6     return rn * weights
```

For this reason, a modified form with equivalent result was chosen, which is shown in Listing 5.1. Instead of taking the described steps one by one, the function shown there is designed to deal directly with vectorized data, *i. e.* data in one-dimensional arrays. After refactoring the function `divide` to processing vectorized data, splitting a model with 1.6 million weights in 4 parts on an Intel Core i7-4790 at 4GHz took only about 20 seconds. First, a list of #peers random numbers is created in line 2; the list has #weights many elements. These random numbers are normalized in the following loop in such a way that the sum is 1, so they correspond to a percentage distribution. In line 5, this list is converted into a NumPy[2] array to transpose it directly. `rn` contains after line 5 a NumPy array with #weights elements in the first dimension with #peers sub-elements. In line

[1] 1.6 million weights are very few today, as described in Section 3.1

[2] NumPy is a Python library for efficient calculation with tensors and is very often used in the context of ML.

6, the actual division of the weights takes place, which looks like a simple multiplication thanks to NumPy. For a better understanding, the individual steps of `divide` are now illustrated with the following example:

Input parameters:

$$\begin{aligned} \text{weights} &= [1,\ 0.5,\ 0.25] \\ \text{peers} &= 2 \\ \text{rnd\_gen} &= 2^c, \text{ with initial } c = 0 \text{ and } c = c + 1 \text{ with each call.} \end{aligned}$$

After line 2:

$$\begin{aligned} \text{rn} = [&\text{array}([1,\ 2]), \\ &\text{array}([2,\ 4]), \\ &\text{array}([4,\ 8])] \end{aligned}$$

After lines 3-4:

$$\begin{aligned} \text{rn} = [&\text{array}([0.333,\ 0.667]), \\ &\text{array}([0.333,\ 0.667]), \\ &\text{array}([0.333,\ 0.667])] \end{aligned}$$

After line 5:

$$\begin{aligned} \text{rn} = \text{array}([&[0.333, 0.333, 0.333], \\ &[0.667, 0.667, 0.667]]) \end{aligned}$$

Output parameters:

$$\begin{aligned} \text{array}([&[0.333, 0.1667, 0.0833], \\ &[0.667, 0.3333, 0.1667]]) \end{aligned}$$

The weights in Keras models are arranged below the specific layers. Therefore, not all weights can be queried directly, as described in lines 3-5 of Algorithm 4.1, but must be queried for each layer individually. Also, the dimension of the weight tensors depends on the layer type and specific layer configuration. Keras uses NumPy arrays to store the tensors and returns the corresponding weights as a list of NumPy arrays when `get_weights()` on a layer is called. NumPy arrays have a `shape` property that specifies which dimension has how many associated elements. For example, a convolutional layer with 32 filters, a kernel size of $(5,\ 5)$, a step size of 1 with padding="same", and bias=True, and one input channel returns this list of NumPy array shapes $[(5,\ 5,\ 1,\ 32),\ (32,)]$. In contrast, a call to `get_weights()` for a MaxPooling layer returns an empty list because there are no weights. A dense layer with 3136 input channels and 512 output channels returns this list of NumPy array shapes $[(3136,\ 512),\ (512,)]$. For this reason, it is crucial that the implementation can handle such dynamic shapes.

Listing 5.2 shows the implementation for splitting the transferred weights. It is assumed that the parameter `layer_weights` contains a list of weights, as mentioned before. The first for-loop (lines 3-5) is executed for each element of the passed layer weights. First, in line 4, the possibly multidimensional array is converted into a one-dimensional array using the NumPy function `ravel`, so that the function `divide` (line 5) always gets one-dimensional data passed. The result of the function `divide` is an array of the shape (#flat_weights, #peers) and is also added to the list w in line 5. In lines 7-9, the `final_weights` list is pre-initialized so that it then has #peers elements,

Listing 5.2 Implementation for dividing the layer weights in Python

```
1   def divide_layer(layer_weights, peers, rnd_gen=np.random.default_rng(time.time_ns())):
2     w = []
3     for weights in layer_weights:
4       flat_weights = np.ravel(weights)
5       w.append(divide(flat_weights, peers, rnd_gen))
6
7     final_weights = []
8     for _ in range(peers):
9       final_weights.append(layer_weights)
10
11    for p in range(peers):
12      for i in range(len(layer_weights)):
13        final_weights[p][i] = w[i][p].reshape(layer_weights[i].shape)
14
15    return final_weights
```

which all have the same shape as `layer_weights`. In lines 11-13, the values of `final_weights` are overwritten with the values previously calculated by `divide`. To ensure that the values are compatible, the conversion to a one-dimensional array must be reversed. This is done with the Numpy function `reshape`, which converts the shape of the array on which the function is called into the given form. For this to succeed, the shapes must be compatible, which means that the total number of elements of both shapes must be identical. During the copy process, a restructuring of the data also takes place, while the form of reshaped w is (#layer_weights, #peers), *i. e.* each weight consists of #peers elements, `final_weight` has the shape (#peers, layer_weights), each peer has #layer_weights elements.

Listing 5.3 shows a condensed version of Listing 5.2, which uses list comprehensions to maximize performance.

Listing 5.3 Minimized version of `divide_layer`

```
1  def divide_layer(layer_weights, peers, rnd_gen=np.random.default_rng(time.time_ns())):
2      w = [divide(np.ravel(weight), peers, rnd_gen) for weight in layer_weights]
3      final_weights = [layer_weights for _ in range(peers)]
4      for p in range(peers):
5          for i in range(len(layer_weights)):
6              final_weights[p][i] = w[i][p].reshape(layer_weights[i].shape)
7      return final_weights
```

6 Experiments

This chapter will show how well the discussed approaches work in practice. The experiments were performed with three different image classification datasets. Accordingly, three different models were designed for this purpose. The dataset-specific subchapters are structured identically. First, the dataset and the model are presented, and then, the experiments are analyzed. The corresponding source code can be found in Appendix B.1.

Comparing the three datasets and the associated tasks with each other, Section 6.1 is a simple experiment where each party has a relatively large amount of training data consisting relatively simple images. Section 6.2 is a bit more challenging than Section 6.1, the now colored images are slightly bigger, but the amount of training data is similar. In contrast, the task in Section 6.3 is much more challenging because compared to Section 6.1, the parties do not even have 1/6 of the training data available, and the data is much more complex.

In general, all experiments were performed with balanced data, *i. e.* all parties involved had similarly large datasets. The experiments were performed with both approaches presented in Chapter 4. To be able to compare the effort of the approaches, the respective communication effort is calculated, which describes how many weights must be communicated in total.

Decentralized split learning uses the approach from Section 4.1, which was simulated by training a model locally, alternating with different training data. A communication round using this approach means that one party has trained the model locally and then communicates it to all other parties involved. This results in a communication effort of $(p-1)*n$ weights, where p is the number of parties involved, and n is the number of weights.

Decentralized federated optimization uses the approach from Section 4.2. One communication round using this approach means that all parties involved have trained the model locally and then calculated the weights‘ mean values. The communication effort per communication round depends on the selected security level. Without data protection, the communication effort is $p*(p-1)*n = n(p^2-p)$ weights, since each party must communicate all weight values to each other. With `SecAvg` (semi-honest security), the communication effort is $p*(p-1)*n+p*(p-1)*n = 2n(p^2-p)$ weights, since the weight parts must be distributed first and then the subtotals.

6.1 Experiments on MNIST

All experiments aimed to achieve 99% accuracy in the test dataset, whereby the experiments were stopped after 3,000 training rounds at the latest. Each experiment was performed five times, which are then called Run1 - Run5. Additionally, the following parameters were used over all experiments:

- Party count: 5
- Optimizer: Adam
- Learning rate: 0.001
- Loss function: Categorical cross-entropy
- Number of epochs: 1
- Batch size: 64
- Validation split[1]: 0.2

6.1.1 MNIST dataset

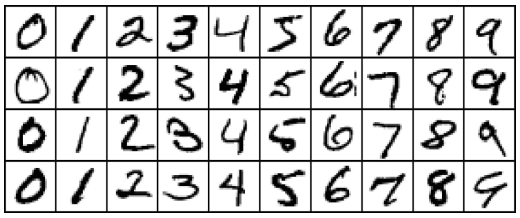

Figure 6.1: Example digits from the MNIST dataset

The Modified NIST dataset is a well-known set of images (28x28 pixels) showing handwritten digits. It was first introduced by LeCun et al. in [71] and is a subset of a larger set available from NIST. Today it is often called the "hello world" of ML. The dataset consists of a training set of 60,000 annotated images and a test set of 10,000 annotated images. Figure 6.1 shows some digits from the dataset.

The MNIST dataset is used because it is well known, and at the same time, it is small enough to train models quickly even with commodity hardware.

6.1.2 The applied model

The model applied here is a small DNN for image recognition. It consists of 8 layers and 1,663,370 weights in total. This model was chosen because it corresponds to one of the two models chosen for the MNIST dataset by [81], so the results should be comparable.

[1] *"Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch."* [104]

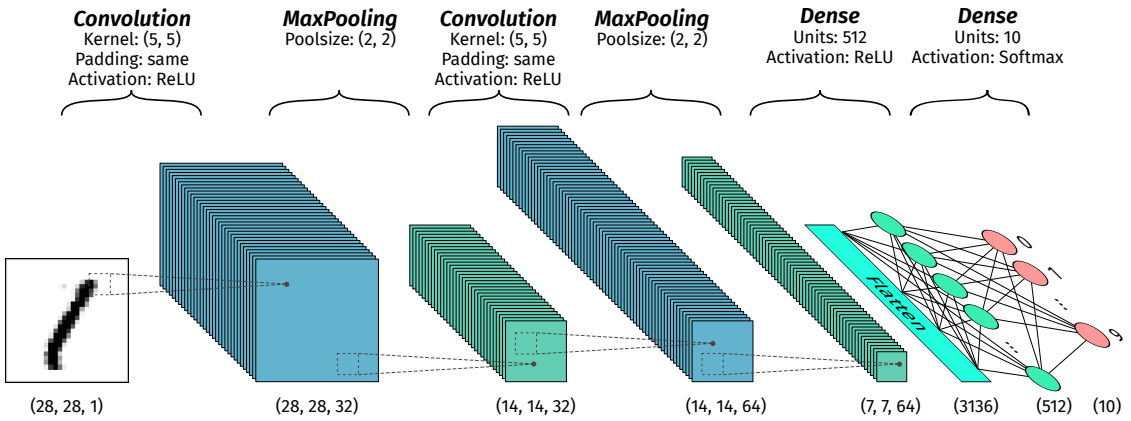

Figure 6.2: The applied model

Figure 6.2 shows the applied model as a schematic representation. The different types of layers have already been discussed in Section 2.1. The layers are written in bold on the top. Below, the values describe the applied hyperparameters. The corresponding tensor dimensions are shown at the bottom.

6.1.3 Experiment M1: IID data

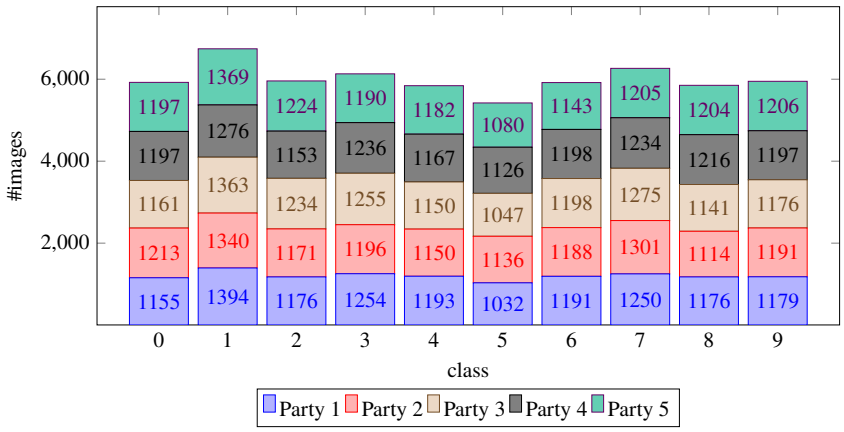

Figure 6.3: Dataset distribution for experiment M1.

This experiment represents the simplest and most optimal case. The data distribution of the five involved parties is IID. This distribution was achieved by shuffling the training data and then distributing it among the parties. Each party received $1/5$ of the total training data, *i. e.* $12,000$ annotated training images for each party. The concrete distribution is shown in Figure 6.3, where it can be seen that the training images of the digits were similarly distributed between the parties.

It turned out that a single party under the boundary conditions described before with $12,000$ annotated training images was only able to achieve 99% accuracy in three of 25 experiments (5 runs with 5 different datasets each). The accuracy of 99% was achieved in the three cases after 34,

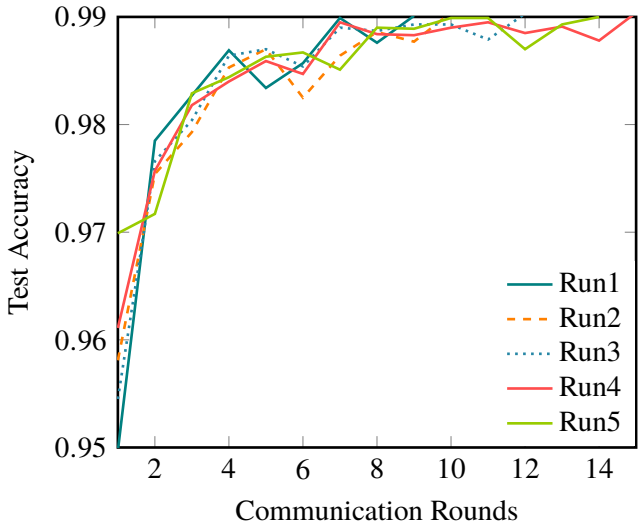

Figure 6.4: Test accuracy over communication rounds for decentralized SL of experiment M1.

278, and 365 rounds, respectively. Presumably, it is possible to get more out of the 12,000 training images by changing the boundary conditions, but this result should only serve as a lower limit for federated learning.

Figure 6.4 shows the accuracy curve for five runs with decentralized SL. As can be seen, this scenario‘s approach is appropriate, as all runs have reached the set target of 99% accuracy. The minimum number of communication rounds was 9 (Run1); the maximum number was 15 (Run4), and the median was 12.

Figure 6.5 shows the results of experiments with decentralized FO. This approach is also suitable for the scenario described since all runs have reached the set target of 99%. The minimum number of communication rounds was 4 (Run2, Run4, Run5), and the maximum was 5 (Run1, Run3).

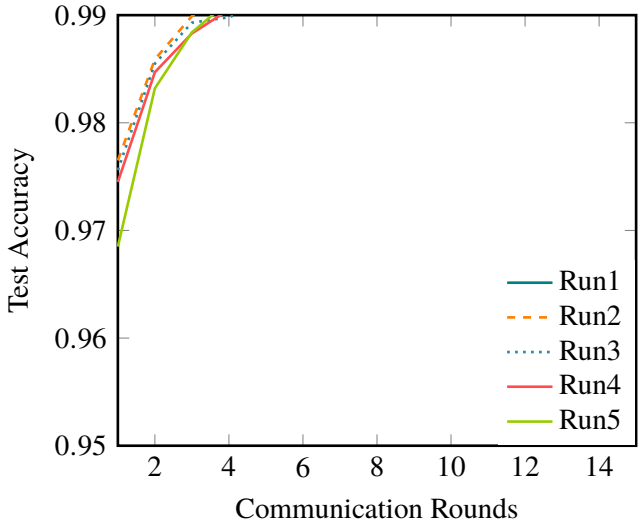

Figure 6.5: Test accuracy over communication rounds for decentralized FO of experiment M1.

If one compares both approaches‘ results, the results of decentralized FO look better at first glance. However, on closer inspection one has to make two distinctions: The number of all training rounds completed and the required communication effort per round. In the case of decentralized SL, the number of all completed training rounds corresponds to the number of communication rounds. In the case of decentralized FO, the factor p lies between the number of training rounds and the number of communication rounds. In the case of decentralized FO, the communication effort per communication round depends on the security level selected. The exact values of the respective efforts are shown in Table 6.1. According to this, the effort of the data privacy offered by `SecAvg` is a good three times higher than with decentralized SL.

	#communication rounds	#training rounds	communication effort per round (in weights)	total communication effort (in weights)
Decentralized SL	12	12	6,653,480	79,841,760
Decentralized FO (no privacy)	4	20	33,267,400	133,069,600
Decentralized FO (`SecAvg`)	4	20	66,534,800	266,139,200

Table 6.1: Comparison of the required effort of experiment M1, based on median results.

6.1.4 Experiment M2: non-IID data (5%)

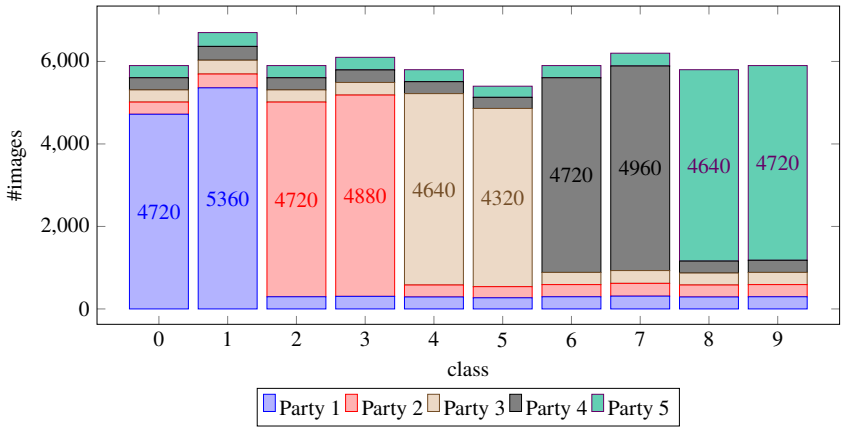

Figure 6.6: Dataset distribution for experiment M2.

In contrast to experiment M1, the data distribution is not IID. As shown in Figure 6.6, each party has two main classes, of which they have by far the most training images. Of the remaining digits, they have only 1/4 of the data from experiment M1. More precisely, each party has 80% of its main digits‘ training data and 5% of the remaining digits‘ training data. With such a distribution, in contrast to experiment M1, it is no longer possible for a single party alone to achieve 99% accuracy.

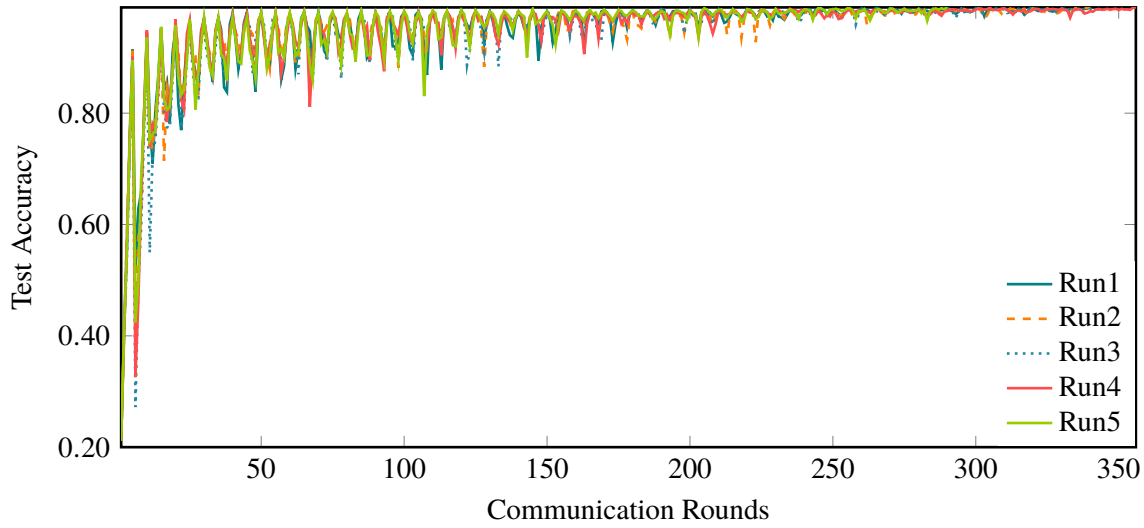

Figure 6.7: Test accuracy over communication rounds for decentralized SL of experiment M2.

Figure 6.7 shows the accuracy curve for five experiments with decentralized SL. As can be seen, scenario‘s approach is appropriate, as all runs have reached the set target of 99% accuracy. The minimum number of communication rounds was 290 (Run5), the maximum number was 356 (Run4), and the median was 340.

Figure 6.8 shows the results of experiments with decentralized FO. This approach is also suitable for the scenario described since all runs have reached the set target. The minimum number of communication rounds was 155 (Run3), the maximum was 185 (Run2), and the median was 158.

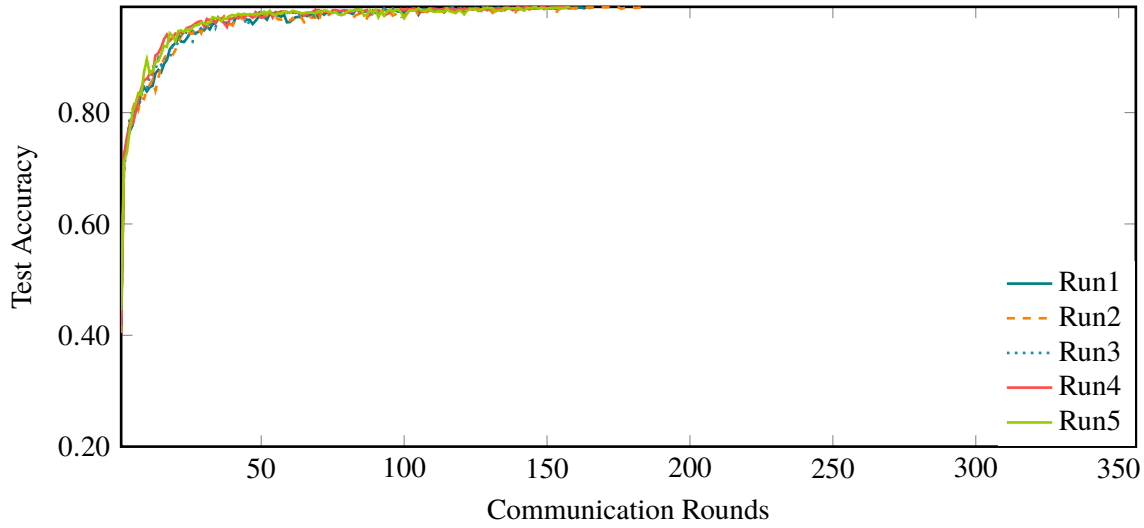

Figure 6.8: Test accuracy over communication rounds forr decentralized FO of experiment M2.

When comparing the two approaches, the results of decentralized FO also look better at first glance, so we now examine the results again according to the total number of training rounds and the communication effort required. In the case of decentralized SL, the number of all completed training rounds corresponds to the number of communication rounds. In the case of decentralized

FO, the factor p lies between the number of training rounds and the number of communication rounds. In the case of decentralized FO, the communication effort per communication round depends on the security level selected. The exact values of the respective efforts are shown in Table 6.2. According to this, the effort of the data privacy offered by `SecAvg` is a good 4.5 times higher than with decentralized SL.

	#communication rounds	#training rounds	communication effort per round (in weights)	total communication effort (in weights)
Decentralized SL	340	340	6,653,480	2,262,183,200
Decentralized FO (no privacy)	158	790	33,267,400	5,256,249,200
Decentralized FO (`SecAvg`)	158	790	66,534,800	10,512,498,400

Table 6.2: Comparison of the required effort of experiment M2, based on median results.

The experiment was also performed with modified distributions, so that each party's training data became more specialized, leaving each party with only 3% and 1% of the training data of the non-main digits, respectively. Interestingly, the results for these distributions do not differ much from those described here. Decentralized SL required 340 and 387 communication rounds in the case of 3% and 1% in the median. Decentralized FO required 198 and 184 communication rounds for 3% and 1%, respectively. The corresponding graphs can be found in Appendix A.1 and Appendix A.2.

6.1.5 Experiment M5: non-IID data (0%)

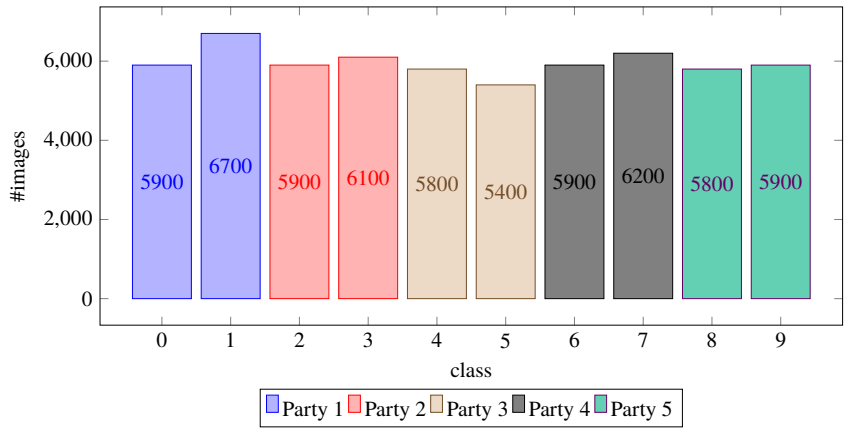

Figure 6.9: Dataset distribution for experiment M5.

In this experiment, the most extreme distribution was chosen. As shown in Figure 6.9, the datasets are now completely disjointed. Thus, each party has only the training data of the two main digits available, and the other digits' training data is no longer available.

Figure 6.10 shows the accuracy curve for five experiments with decentralized SL. As can be seen, this scenario‘s approach is appropriate, as all runs have reached the set target of 99% accuracy. The minimum number of communication rounds was 451 (Run1), the maximum number was 527 (Run4), and the median was 488.

Figure 6.11 shows the results of experiments with decentralized FO. Surprisingly, the results show that only 2 of 5 runs reached the set target of 99% accuracy. The best result was achieved in Run1 with 872 communication rounds, and in Run2, there were 933 communication rounds, giving an average of 902.5.

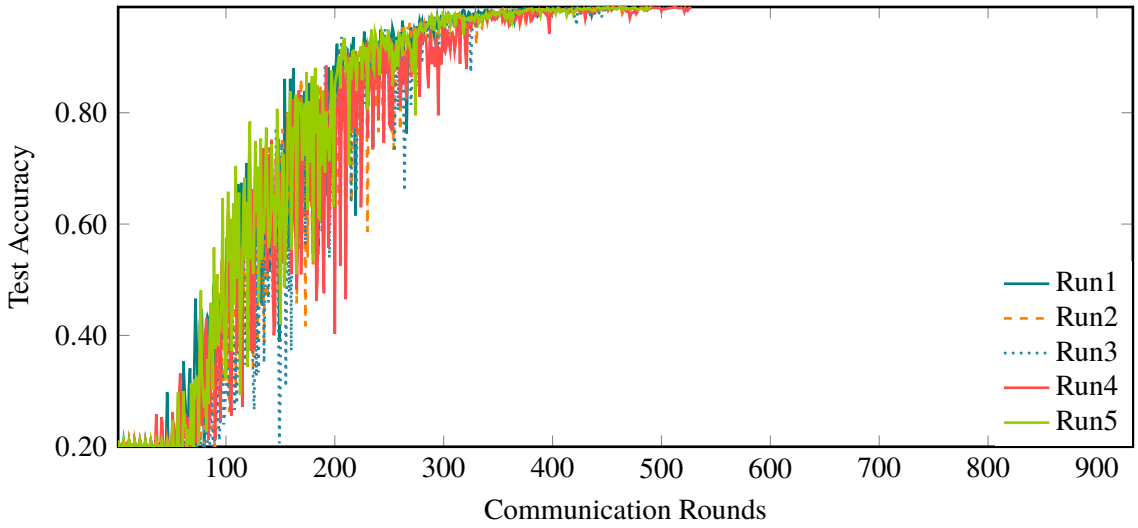

Figure 6.10: Test accuracy over communication rounds for decentralized SL of experiment M5.

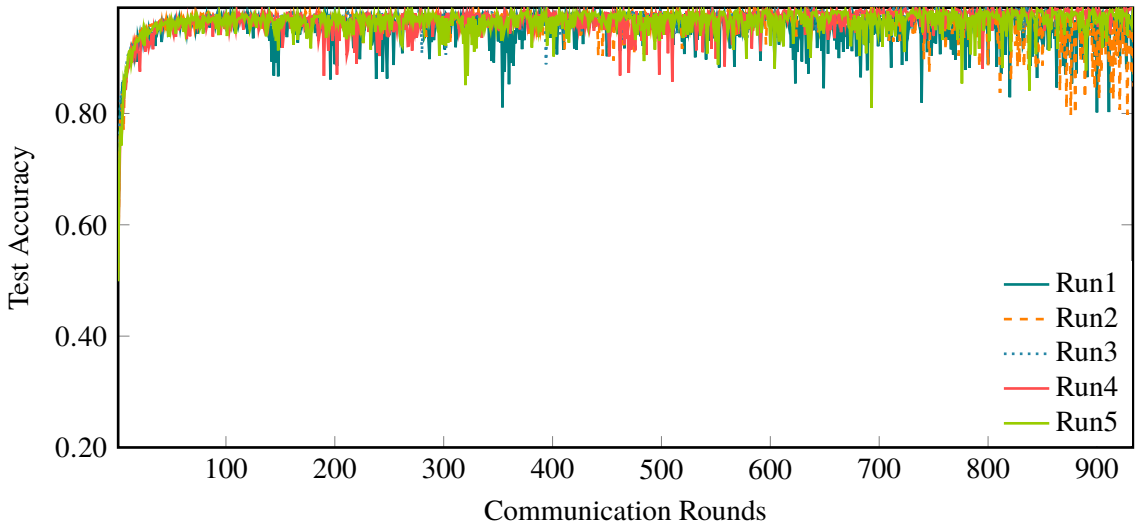

Figure 6.11: Test accuracy over communication rounds for decentralized FO of experiment M5.

When comparing the results of both approaches, two things stand out. On the one hand, decentralized FO delivers very stable results early on and misses the target by a small margin. On the other hand, decentralized SL performs significantly better overall, even if the intermediate results are unstable.

In the case of decentralized SL, the experiment was successful in all rounds, and only about half as many communication rounds were required at the same time (488 compared to 902.5). For the sake of completeness, Table 6.3 shows the corresponding communication efforts. The effort for `SecAvg` is a good 18 times higher than for decentralized SL.

	#communication rounds	#training rounds	communication effort per round (in weights)	total communication effort (in weights)
Decentralized SL	488	488	6,653,480	3,246,898,240
Decentralized FO (no privacy)	902.5	4512.5	33,267,400	30,023,828,500
Decentralized FO (`SecAvg`)	902.5	4512.5	66,534,800	60,047,657,000

Table 6.3: Comparison of the required effort of experiment M2, based on median results.

6.2 Experiments on CIFAR-10

The goal of all experiments was to achieve at least 70%[2] accuracy within 500 communication rounds. Each experiment was performed five times, which are then called Run1-Run5. Additionally, the following parameters were used for all experiments:

- Party count: 5
- Optimizer: Adam
- Loss function: Categorical cross-entropy
- Learning rate: 0.0001
- Number of epochs: 1
- Batch size: 64

[2] 70% accuracy was achieved with the complete dataset and the applied model after 60 epochs on average.

6.2.1 CIFAR-10 dataset

Figure 6.12: Upscaled example images from the CIFAR-10 dataset

The CIFAR-10 dataset was introduced in 2009 and is named after the Canadian Institute for Advanced Research [70]. It consists of 60, 000 images (50, 000 for training and 10, 000 for testing). Each image is 32x32 pixels in size, and, unlike MNIST, these are RGB images and, therefore, each image has 3 channels. The number in CIFAR-10 describes the number of classes. There is also another variation with 100 classes, the CIFAR-100.

6.2.2 The applied model

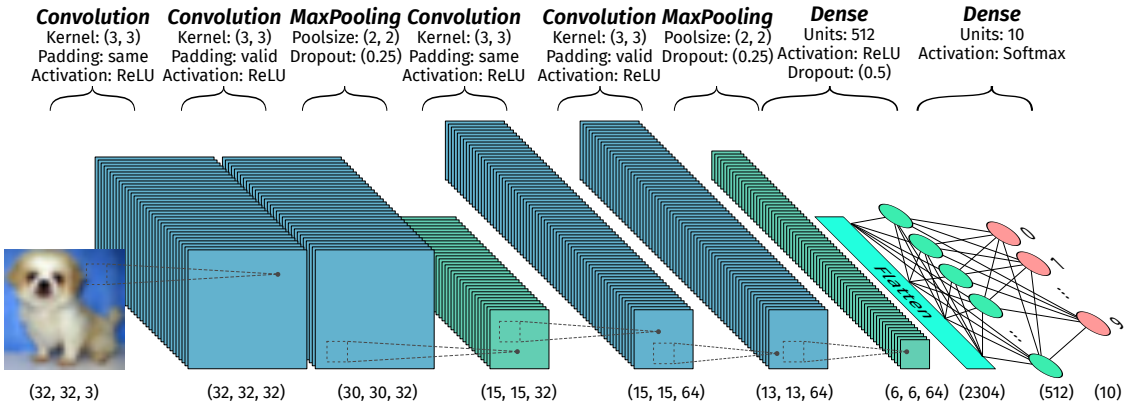

Figure 6.13: The applied model

The model applied here is a small DNN for image recognition. It was taken from an example[3] for Keras. Strictly speaking, it consists of 18 layers, since the activation functions were modeled as individual layers. Figure 6.13 shows a schematic representation of the model. The structure is similar to the model in Section 6.1.2. Instead of one convolutional layer followed by a MaxPooling-layer, two convolutional layers are combined with one MaxPooling-layer. Additionally, three Dropout-layers, one after each MaxPooling-layer and one before the output layer, avoid overfitting. Even though the four convolutional layers make it more complex than the model from Section 6.1, it has good 400, 000 weights less with only 1, 250, 858 weights.

[3] https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py

6.2.3 Experiment C1: IID data

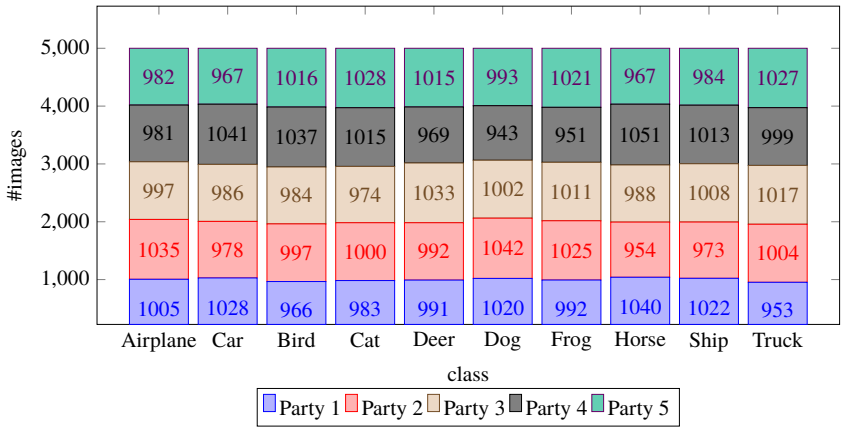

Figure 6.14: Dataset distribution for experiment C1.

This experiment again represents the simplest and optimal case. As shown in Figure 6.14, the data distribution of the five involved parties is almost IID. It was achieved by shuffling the training data and then distributing it among the parties. Each party received $1/5$ of the total training data, *i. e.* $5,000$ annotated training images for each party.

A single party with $5,000$ images cannot achieve the goal of 70% accuracy under the described conditions. Each party achieved about 58% accuracy during tests.

Figure 6.15 shows the accuracy curve for five experiments with decentralized SL. As can be seen, this scenario's approach is appropriate, as all runs have reached the set target of 70% accuracy. The individual experiment runs proceeded similarly, even if there were differences in the results. The minimum number of communication rounds was 262 (Run2), the maximum was 318 (Run5), and the median was 269.

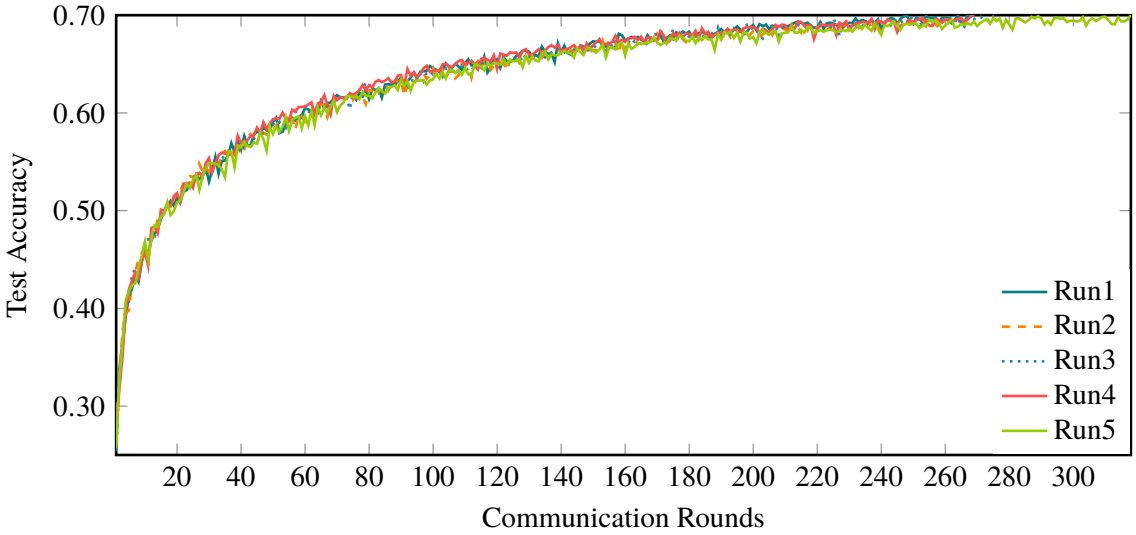

Figure 6.15: Test accuracy over communication rounds for decentralized SL of experiment C1.

Figure 6.16 shows the results of experiments with decentralized FO. This approach is also suitable for the described scenario since all runs have reached the set target. The minimum number of communication rounds was 220 (Run3), the maximum was 255 (Run1), and the median was 234.

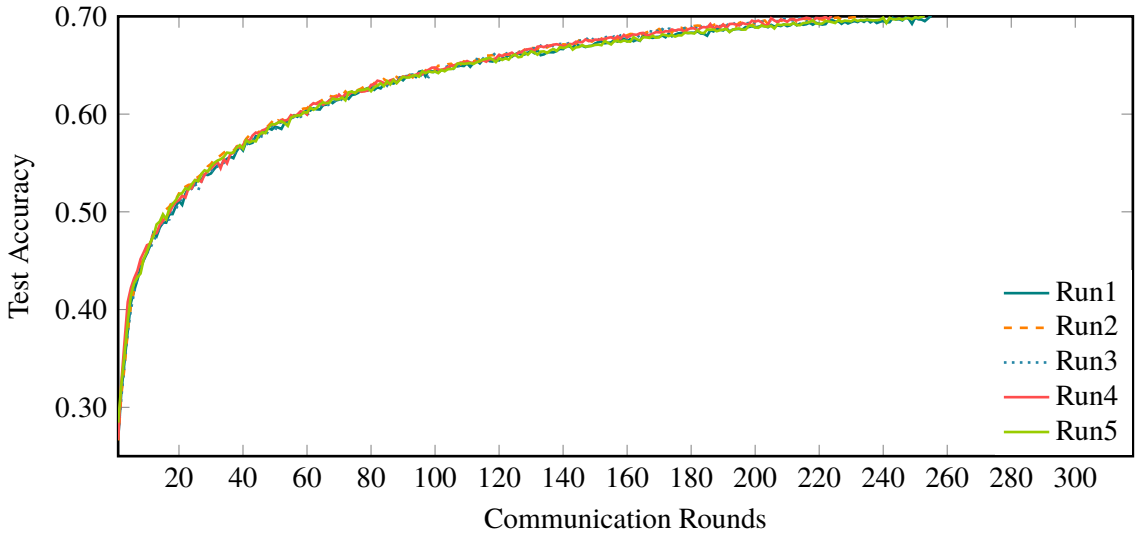

Figure 6.16: Test accuracy over communication rounds for decentralized FO of experiment C1.

When comparing both approaches‘ results, it turns out that decentralized FO needs, on average, 35 communication rounds less. This difference is put into perspective when you consider the communication effort (Table 6.4). The communication effort of one round with decentralized FO is 5 or 10 times as high as that of decentralized SL, depending on whether increased data privacy is desired.

	#communication rounds	#training rounds	communication effort per round (in weights)	total communication effort (in weights)
Decentralized SL	269	269	5,003,432	1,345,923,208
Decentralized FO (no privacy)	234	1,170	25,017,160	5,854,015,440
Decentralized FO (`SecAvg`)	234	1,170	50,034,320	11,708,030,880

Table 6.4: Comparison of the required effort of experiment C1, based on median results.

6.2.4 Experiment C2: non-IID data (5%)

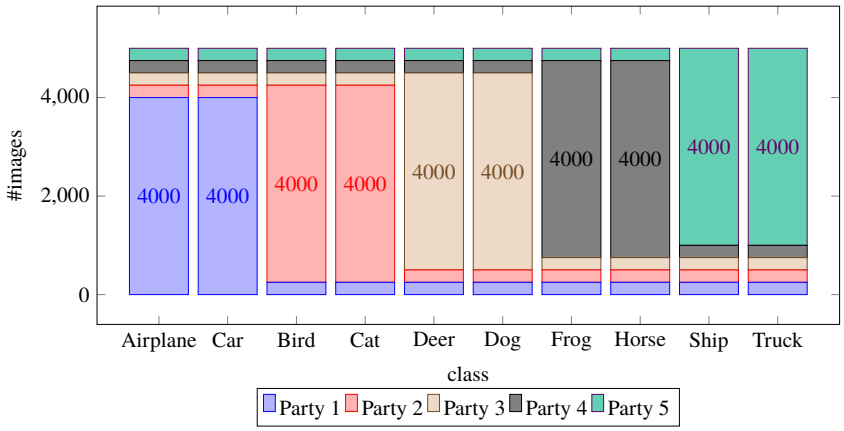

Figure 6.17: Dataset distribution for experiment C2.

In this experiment, the data distribution is no longer IID. Each party has two main classes, of which it has 4,000 training images each and 250 training images (5%) from all other classes, as shown in Figure 6.17.

Figure 6.18 shows the accuracy curve for the runs with decentralized SL. As can be seen, this scenario's approach has reached about 60% accuracy, and therefore the goal of 70% accuracy was not achieved. Each run performed almost identical.

Figure 6.18 shows the results of experiments with decentralized FO. The different runs performed almost identically and achieved about 68% accuracy, narrowly missing the aimed 70% accuracy.

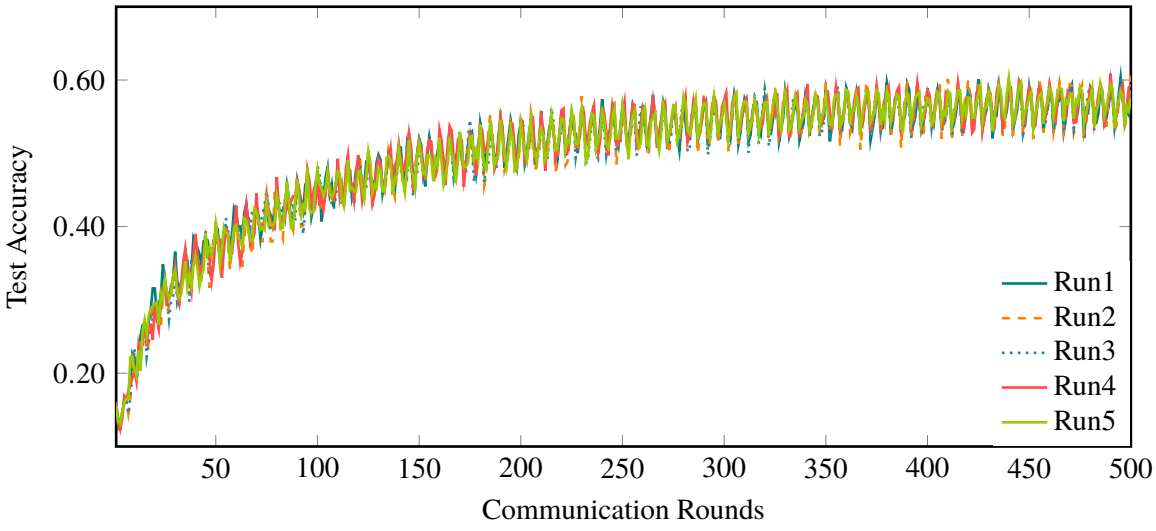

Figure 6.18: Test accuracy over communication rounds for decentralized SL of experiment C2.

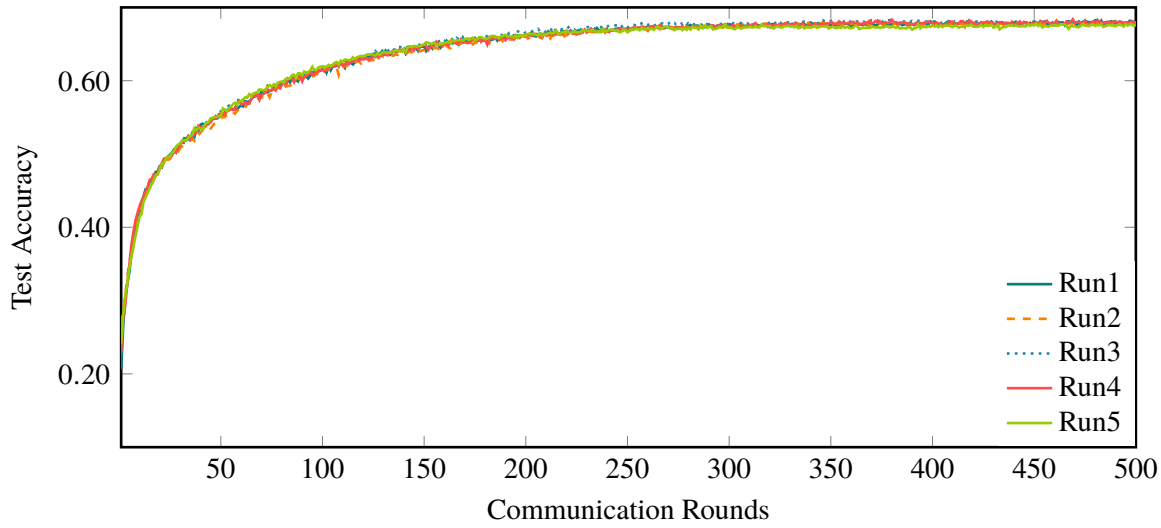

Figure 6.19: Test accuracy over communication rounds for decentralized FO of experiment C2.

6.2.5 Experiment C3: non-IID data (0%)

In this experiment, the most extreme distribution was chosen. The datasets are completely disjointed. Thus, each party has the training data for only two classes and no information about the other classes. With such a distribution, one party alone can reach a maximum of 20%.

Figure 6.20 shows the accuracy curve for five experiments with decentralized SL. As can be seen, this approach is not suitable for the scenario, as the results show that the accuracy varies between 10% and 16%, and no real improvements can be achieved over time.

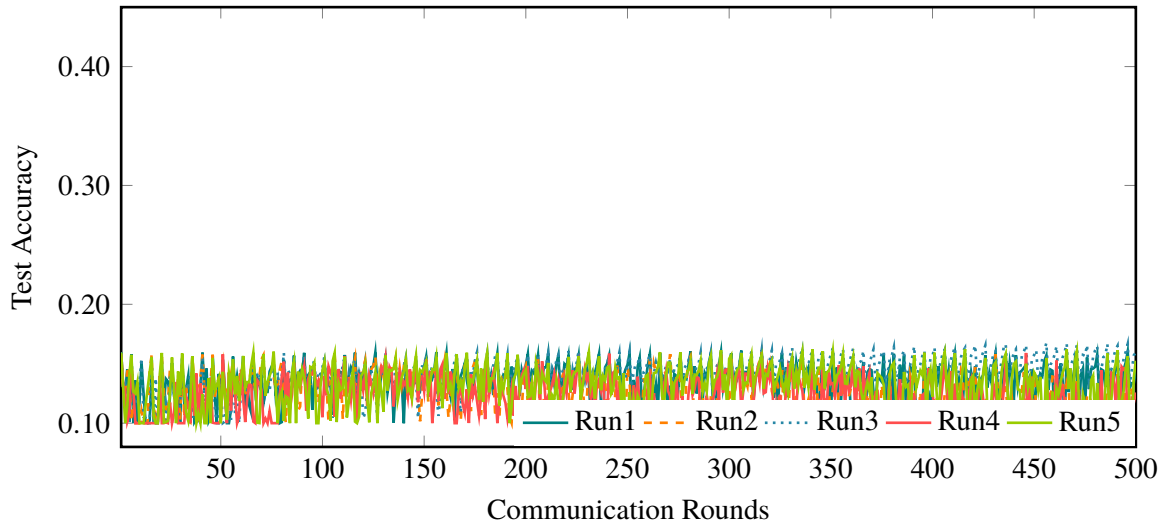

Figure 6.20: Test accuracy over communication rounds for decentralized SL of experiment C3.

Figure 6.21 shows the results of the experiments with decentralized FO. The results show that the goal is not reached, but there are improvements over time. The accuracy improves from around 10% initially to over 40%, which is at least better than what one party could achieve on its own.

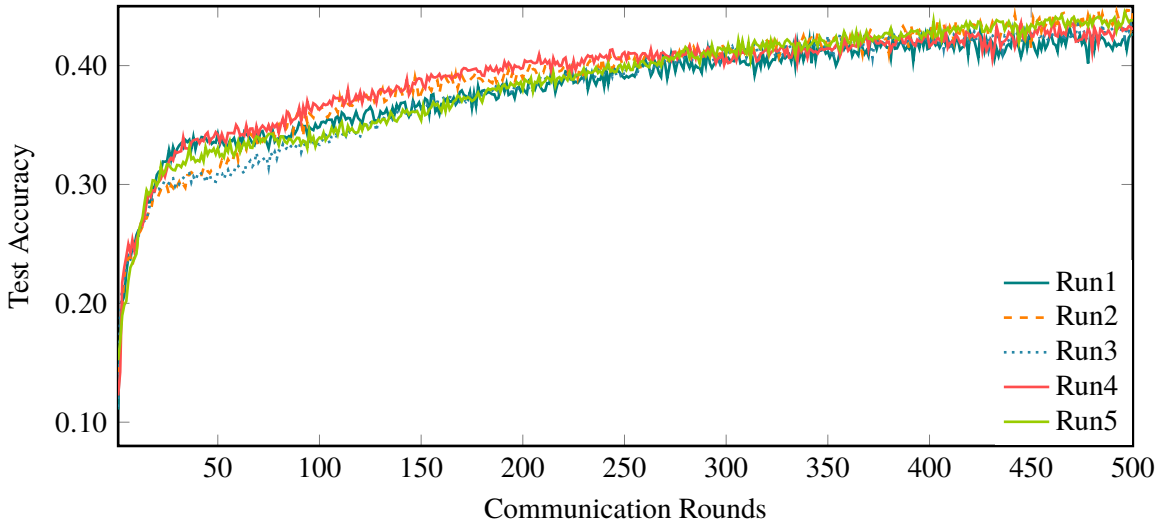

Figure 6.21: Test accuracy over communication rounds for decentralized FO of experiment C3.

6.3 Experiments on Imagenette

This experiment‘s setup should show how well both approaches work with more complex models and datasets. All experiments aimed to reach 75% accuracy as quickly as possible, with an upper limit of 200 for communication rounds. To make the task more challenging, the difference in accuracy between two consecutive (communication) rounds must be greater than 0.000001. Each experiment was performed three times, which are then called Run1 - Run3. Additionally, the following parameters were used for all experiments:

- Party count: 5
- Optimizer: Adam
- Learning rate: 0.001
- Loss function: Categorical cross-entropy
- Number of epochs: 1
- Batch size: 64

6.3.1 Imagenette dataset

"The explosion of image data on the Internet has the potential to foster more sophisticated and robust models and algorithms to index, retrieve, organize and interact with images and multimedia data. But exactly how such data can be harnessed and organized remains a critical problem. We introduce here a new database called "ImageNet", a largescale ontology of images built upon the backbone of the WordNet structure. ImageNet aims to populate the majority of the 80,000 *synsets*[4] *of WordNet with an average of* 500-1,000 *clean and full resolution images. This will result in tens of millions of annotated images organized by the semantic hierarchy of WordNet."* [35]

Figure 6.22: Example images from the Imagenette dataset

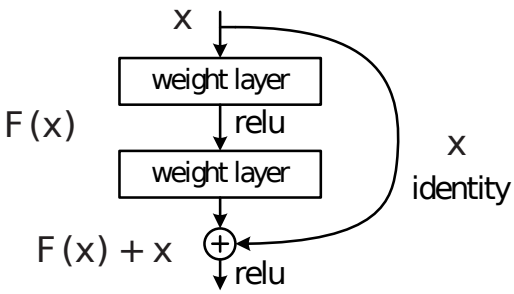

Figure 6.23: Residual learning [60]

Currently, ImageNet contains more than 14 million images in over 20,000 synsets. Since the processing of such data volumes takes a long time, Imagenette [63] was created. *"Imagenette is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute)."* [63] The version used here consists of 13,394 color photos, divided into 9,469 training images and 3,925 validation images. This input data is scaled so that the shortest side is 160 pixels long. Figure 6.22 shows one example of each category.

6.3.2 The applied model

The model used in this experiment is the so-called ResNet50. ResNet [60] stands for *residual (neural) network* and is a so-called advanced CNN architecture, which refers to ready-made models. The number at the end specifies the number of convolutional layers. Deeper nets have more weights and potentially perform better.

[4] *"Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset""* [35]

As mentioned in Section 2.1.3, the vanishing gradient problem in deep neural networks causes the earlier layers to learn more and more slowly. ResNet uses *deep residual learning* as a countermeasure. As shown in Figure 6.23, residual learning means that the input x will be forwarded and added to the processed output $F(x)$. Such skipped connections or shortcut connections mitigate the vanishing gradient problem and ensure that the higher layers work at least as well as the lower layers [38].

The here used ResNet50 is not entirely identical to the original. It is the version provided by Keras, which includes some additional newer insights. The model basically consists of two blocks, an *identity block (IdBlock)* and a *convolutional block (ConvBlock)*, configured with different numbers of filters. From stage to stage, the used filters of the convolutional layers double.

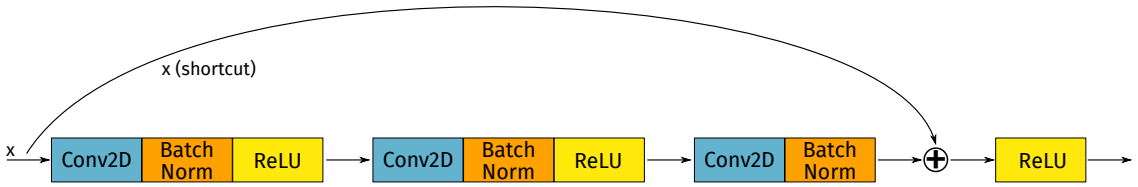

Figure 6.24: ResNet50 identity block [38]

Figure 6.24 shows such an IdBlock. As you can see, there is a short way and a main way. The main path‘s convolutional layers use a 1x1 kernel for the first and last layer and a 3x3 kernel in the second layer, where each layer has a stride of $(1, 1)$. The value of x is added to the result before the last ReLU function.

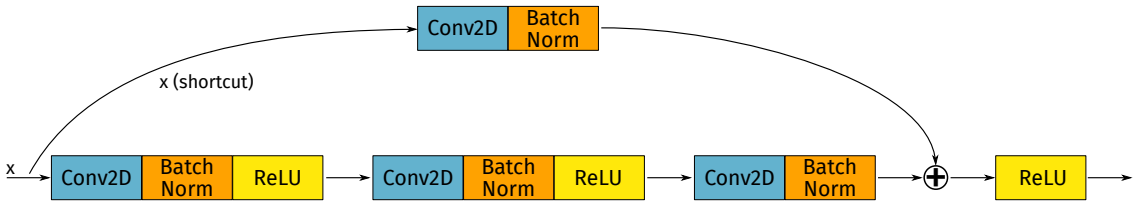

Figure 6.25: ResNet50 convolution block [38]

The ConvBlock is similar to the IdBlock, but as shown in Figure 6.25, the shortcut also has a convolutional layer and batch normalization. This block is needed whenever the dimensions of input and output tensor are not equal, which is then adjusted by the convolution. Additionally, it is noticeable that the shortcut does not contain any non-linear activation function. Therefore, the shortcut‘s main task is to apply a learned linear function, which reduces the input tensor‘s dimension to the output tensor‘s dimension. The used kernel sizes are identical to those of the IdBlock, only the used stride of the first layer is $(2, 2)$, which results in a halving of the dimension. The parameterization of the shortcut is identical to the first layer of the main path.

With these block‘ help, the used model can now be represented schematically, as shown in Figure 6.26. When recounting, one will notice that this model has 49 or 53 convolutional layers, depending on whether one considers only the main path or all paths. This is because the original ResNet in stage 1 contained another convolutional layer, and only IdBlocks were used.

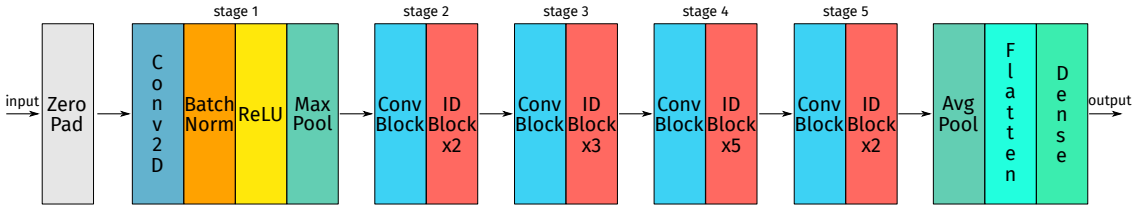

Figure 6.26: ResNet50 [38]

The original ResNet was designed for the ImageNet dataset, which resulted in the last layer being a Dense layer with $1,000$ units since ImageNet has $1,000$ classes. Since Imagenette contains only 10 classes, a dense layer with 10 units is used. The version of ResNet50 described here has $23,608,202$ weights, of which $23,555,082$ can be trained.

The ResNet expects an input tensor of the form $(224, 224, 3)$, Imagenette images were resized accordingly. This can cause image distortions.

6.3.3 Experiment I1: IID data

This experiment again represents the simplest and most optimal case. The data distribution of the five involved parties is IID. This distribution again was achieved by shuffling the training data and then distributing it among the parties. Each party received 1/5 of the total training data, *i. e.* about $1,900$ annotated training images for each party.

It has been found that a single party is unable to achieve the goal of 75% accuracy under the described conditions. In the best case, the accuracy was 63%, and on average, about 59% accuracy. The experiments were terminated after an average of 49 rounds, as the difference between two results was then too small.

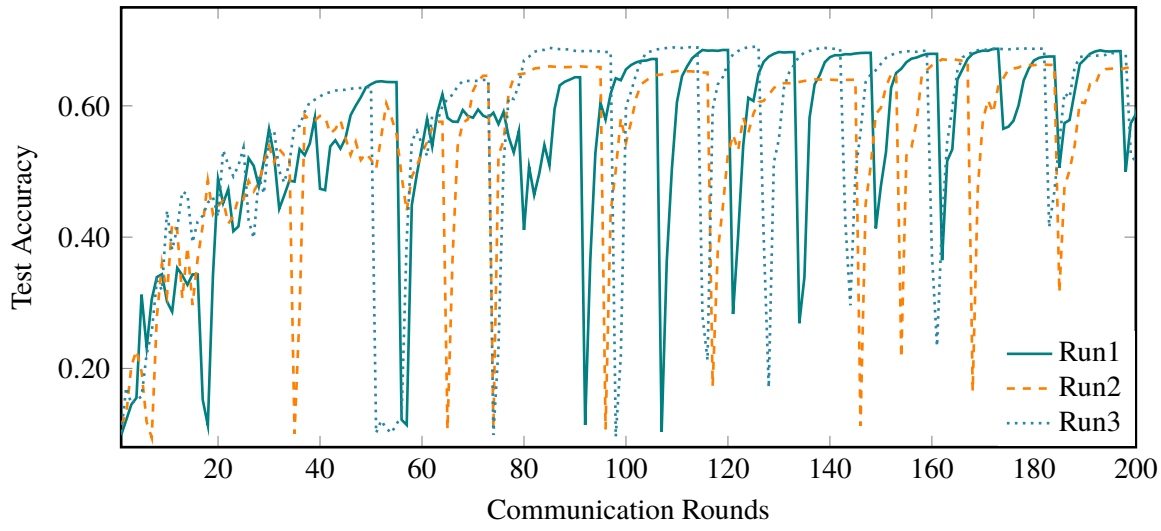

Figure 6.27: Test accuracy over communication rounds for decentralized SL of experiment I1.

Figure 6.27 shows the accuracy curve for three experiments with decentralized SL. As can be seen, this scenario‘s approach is not appropriate, as no run has reached the set target of 75% accuracy. The best result was just below 70% accuracy, which is at least better than one party alone could achieve.

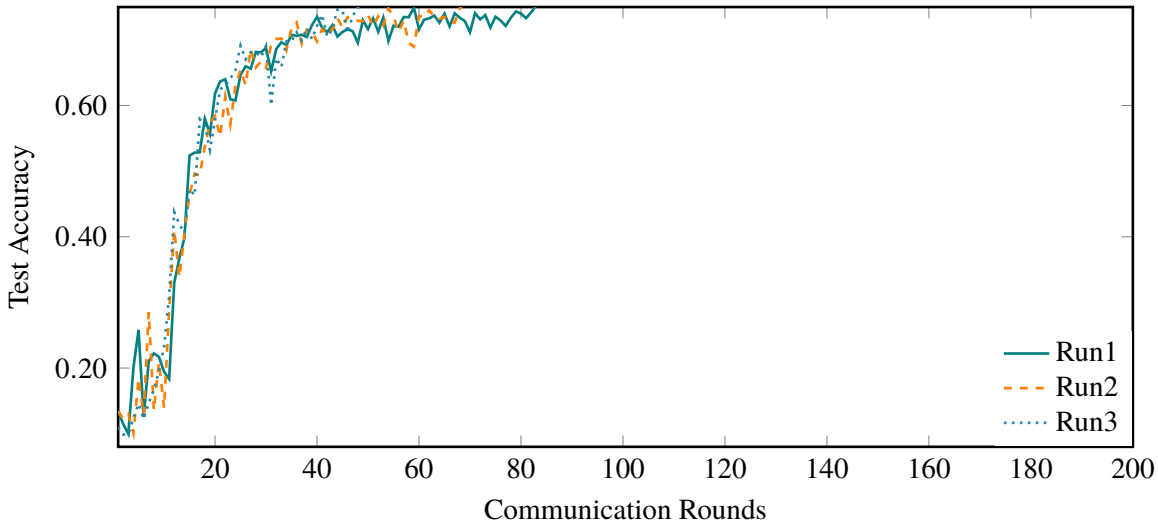

Figure 6.28: Test accuracy over communication rounds for decentralized FO of experiment I1.

Figure 6.28 shows the results of the three experiment runs with decentralized FO. This approach is appropriate in this scenario, as all runs have reached the set target. It even turned out that all runs were over after 83 communication rounds at the latest, so the set goal was achieved after less than half of the available communication rounds, in the best case, even in a quarter.

The comparison of both approaches is not necessary for these results. Decentralized FO has achieved the set goal, in contrast to decentralized SL.

6.3.4 Experiment I2: non-IID data (0%)

In this experiment, the most extreme distribution was chosen. The datasets are completely disjointed. Thus, each party has only two classes‘ training data and no other classes‘ information.

Since decentralized SL could not achieve satisfactory results, even with optimal distribution of the training data, this experiment was performed exclusively with decentralized FO. Due to technical difficulties, the experiment was performed only once. As shown in Figure 6.29, the results obtained ranged from 8% to 14.7%, although they were mostly in the region of about 10%.

Thus decentralized FO and probably every other known approach for federated learning is not able to achieve the set goal under the conditions chosen here.

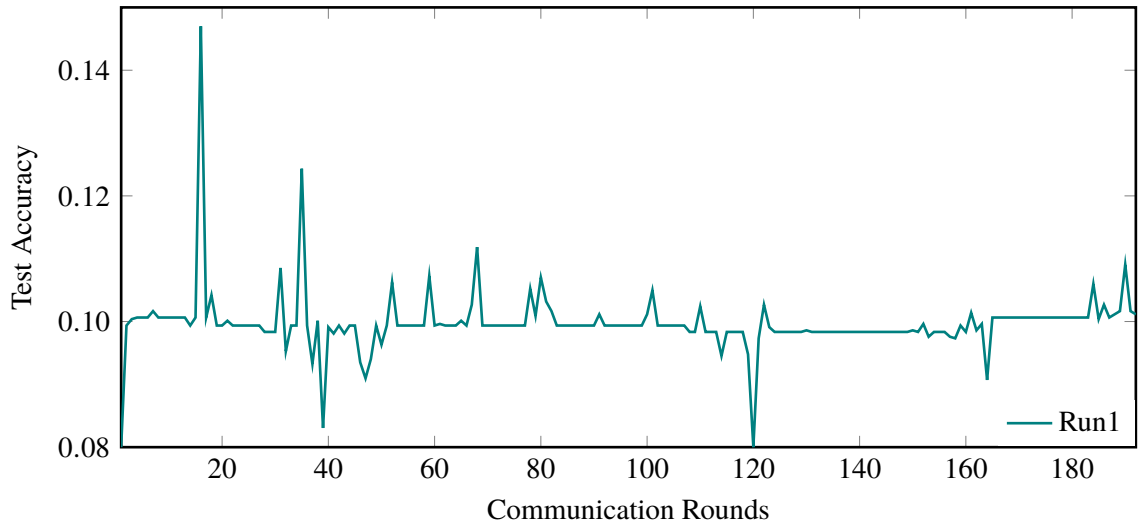

Figure 6.29: Test accuracy over communication rounds for decentralized FO of experiment I2.

7 Conclusion and Outlook

Federated learning with a decentralized communication model is possible, as was shown in this thesis with two different approaches. The experiments showed that even with extreme input data distribution, good results were achieved. Compared to the server-based approaches, it is not the best choice for hundreds or even thousands of parties due to the inherently higher communication efforts for decentralized communication models. However, for a low or medium double-digit number of parties, it could be the right solution, because then all parties are equal and no one has advantages over the others.

Besides private FO[1], `SecAvg` provides a much higher privacy level than other federated learning approaches. Thus all requirements for the initially mentioned application scenario, banks training a common model for credit card fraud detection, are fulfilled when using `SecAvg`.

Although semi-honest security as the chosen security model for SecAvg offers no protection against malicious adversaries who send fake values, this poses no threat in the intended main application scenario, creating a common model by parties who have a high interest in a valid result. An attack on SecAvg would not give the attacker any advantage but will instead sabotage the entire project. However, this sabotage could be noted since the results of the created model would not be satisfying.

The experiments show that decentralized SL, *i. e*. the successive training of a model with different datasets, can achieve excellent and communication-efficient results with IID data and little data privacy compared to `SecAvg`. Perhaps it is possible to combine both methods for this application by creating a kind of equality of weapons in the first rounds with the help of `SecAvg`, and then to train more communication-efficiently with decentralized SL afterward.

Only the basic version of federated optimization has been considered. Recently FO has been extended by adaptive federated optimization [90]. For aggregation, not only the mean value is used there, but, depending on which optimization method is used, other aggregate functions. Adaptive optimization can lead to better results in less time, and therefore, it is of interest for the decentralized case to convert the algorithms into corresponding MPC versions.

To compensate for the loss of a few parties during a training round, a change to a (k, n) secret sharing scheme could be advantageous in the future. Another possibility for future developments is data distribution. Only balanced datasets, *i. e*. datasets with comparable scope and quality, have been considered. When looking at unbalanced data, one could work with weighted sums to determine the mean values. Nevertheless, for this to be possible, a trustworthy procedure is needed to determine a measure of quality and scope.

[1] Differential privacy has further process-related disadvantages, especially with small datasets [86].

In this thesis, possible approaches were primarily considered as core elements for decentralized communication protocols for federated learning. However, it has to be determined how the first model for the training is created and distributed for the specification of such protocols. Furthermore, mechanisms for changing the group‘s composition and possibilities for commitment to participate in a round are missing. The number of participant parties per round must be known before a round starts due to the MPC protocol.

Bibliography

[1] U. Aïvodji, S. Gambs, T. Ther. *GAMIN: An Adversarial Approach to Black-Box Model Inversion*. 2019. arXiv: 1909.11835 [cs.LG] (cit. on pp. 45, 46, 49).

[2] G. Andrew, S. Chien, N. Papernot. *tenorflow/privacy: Library for training machine learning models with privacy for training data*. 2019. URL: https://github.com/tensorflow/privacy (cit. on p. 42).

[3] B. Applebaum, Y. Ishai, E. Kushilevitz. "How to Garble Arithmetic Circuits". In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, Oct. 2011. DOI: 10.1109/focs.2011.40 (cit. on p. 29).

[4] M. Awad, R. Khanna. "Machine Learning". In: *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*. Berkeley, CA: Apress, 2015, pp. 1–18. ISBN: 978-1-4302-5990-9. DOI: 10.1007/978-1-4302-5990-9_1. URL: https://doi.org/10.1007/978-1-4302-5990-9_1 (cit. on p. 16).

[5] H. Azizpour, A. S. Razavian, J. Sullivan, A. Maki, S. Carlsson. "From generic to specific deep representations for visual recognition". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, June 2015. DOI: 10.1109/cvprw.2015.7301270 (cit. on p. 44).

[6] L. J. Ba, R. Caruana. "Do Deep Nets Really Need to be Deep?" In: *CoRR* abs/1312.6184 (2013). arXiv: 1312.6184. URL: http://arxiv.org/abs/1312.6184 (cit. on p. 20).

[7] M. Backes, B. Pfitzmann, M. Waidner. "A General Composition Theorem for Secure Reactive Systems". In: *Theory of Cryptography*. Springer Berlin Heidelberg, 2004, pp. 336–354. DOI: 10.1007/978-3-540-24638-1_19 (cit. on p. 33).

[8] S. Badrinarayanan, A. Jain, N. Manohar, A. Sahai. *Threshold Multi-Key FHE and Applications to Round-Optimal MPC*. Cryptology ePrint Archive, Report 2018/580. https://eprint.iacr.org/2018/580. 2018 (cit. on p. 32).

[9] D. Beaver, S. Micali, P. Rogaway. "The round complexity of secure protocols". In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing - STOC '90*. ACM Press, 1990. DOI: 10.1145/100216.100287 (cit. on p. 29).

[10] Z. Beerliová-Trubíniová, M. Fitzi, M. Hirt, U. Maurer, V. Zikas. "MPC vs. SFE: Perfect Security in a Unified Corruption Model". In: *Theory of Cryptography*. Ed. by R. Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 231–250. ISBN: 978-3-540-78524-8 (cit. on p. 27).

[11] R. Bekkerman, M. Bilenko, J. Langford, eds. *Scaling Up Machine Learning*. Cambridge University Press, 2009. DOI: 10.1017/cbo9781139042918 (cit. on p. 36).

[12] M. Bellare, P. Rogaway. "Optimal asymmetric encryption". In: *Advances in Cryptology — EUROCRYPT'94*. Ed. by A. De Santis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 92–111. ISBN: 978-3-540-44717-7 (cit. on p. 31).

[13] T. Ben-Nun, T. Hoefler. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis*. 2018. arXiv: 1802.09941 [cs.LG] (cit. on p. 38).

[14] P. Bhavsar, I. Safro, N. Bouaynaya, R. Polikar, D. Dera. "Chapter 12 - Machine Learning in Transportation Data Analytics". In: *Data Analytics for Intelligent Transportation Systems*. Ed. by M. Chowdhury, A. Apon, K. Dey. Elsevier, 2017, pp. 283–307. ISBN: 978-0-12-809715-1. DOI: https://doi.org/10.1016/B978-0-12-809715-1.00012-2. URL: http://www.sciencedirect.com/science/article/pii/B9780128097151000122 (cit. on p. 16).

[15] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, R. Rogers. *Protection Against Reconstruction and Its Applications in Private Federated Learning*. 2018. arXiv: 1812.00984 [stat.ML] (cit. on pp. 42, 49).

[16] G. R. Blakley. "Safeguarding cryptographic keys". In: *Managing Requirements Knowledge, International Workshop on*. Los Alamitos, CA, USA: IEEE Computer Society, June 1979, p. 313. DOI: 10.1109/AFIPS.1979.98. URL: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1979.98 (cit. on p. 27).

[17] D. Boneh, A. Sahai, B. Waters. "Functional Encryption: Definitions and Challenges". In: *Theory of Cryptography*. Ed. by Y. Ishai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 253–273. ISBN: 978-3-642-19571-6 (cit. on p. 32).

[18] J. W. Bos, K. Lauter, J. Loftus, M. Naehrig. *Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Report 2013/075. https://eprint.iacr.org/2013/075. 2013 (cit. on p. 31).

[19] B. E. Boser, I. M. Guyon, V. N. Vapnik. "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*. ACM Press, 1992. DOI: 10.1145/130385.130401 (cit. on p. 17).

[20] Z. Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP*. Cryptology ePrint Archive, Report 2012/078. https://eprint.iacr.org/2012/078. 2012 (cit. on p. 31).

[21] Z. Brakerski, C. Gentry, V. Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. https://eprint.iacr.org/2011/277. 2011 (cit. on p. 31).

[22] Z. Brakerski, V. Vaikuntanathan. *Efficient Fully Homomorphic Encryption from (Standard) LWE*. Cryptology ePrint Archive, Report 2011/344. https://eprint.iacr.org/2011/344. 2011 (cit. on p. 31).

[23] G. Brassard, D. Chaum, C. Crépeau. "Minimum disclosure proofs of knowledge". In: *Journal of Computer and System Sciences* 37.2 (Oct. 1988), pp. 156–189. DOI: 10.1016/0022-0000(88)90005-0 (cit. on p. 28).

[24] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone. *Classification and regression trees*. The Wadsworth statistics/probability series. Monterey, CA: Wadsworth & Brooks/Cole Advanced Books & Software, 1984. URL: https://cds.cern.ch/record/2253780 (cit. on p. 17).

[25] S. S. Burra, E. Larraia, J. B. Nielsen, P. S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, N. P. Smart. *High Performance Multi-Party Computation for Binary Circuits Based on Oblivious Transfer*. Cryptology ePrint Archive, Report 2015/472. https://eprint.iacr.org/2015/472. 2015 (cit. on p. 28).

[26] M. Chen, R. Mathews, T. Ouyang, F. Beaufays. *Federated Learning Of Out-Of-Vocabulary Words*. 2019. arXiv: 1903.10635 [cs.CL] (cit. on p. 41).

[27] Y. Chen, X. Sun, Y. Jin. "Communication-Efficient Federated Deep Learning with Asynchronous Model Update and Temporally Weighted Aggregation". In: *CoRR* abs/1903.07424 (2019). arXiv: 1903.07424. URL: http://arxiv.org/abs/1903.07424 (cit. on p. 41).

[28] J. H. Cheon, A. Kim, M. Kim, Y. Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, 2017, pp. 409–437. DOI: 10.1007/978-3-319-70694-8_15 (cit. on p. 31).

[29] F. Chollet. *Deep Learning with Python*. Manning, 2017 (cit. on pp. 15, 17, 18, 26).

[30] B. Chor, S. Goldwasser, S. Micali, B. Awerbuch. "Verifiable secret sharing and achieving simultaneity in the presence of faults". In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 1985. DOI: 10.1109/sfcs.1985.64 (cit. on p. 28).

[31] J. Chotard, E. D. Sans, R. Gay, D. H. Phan, D. Pointcheval. "Decentralized multi-client functional encryption for inner product". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2018, pp. 703–732 (cit. on p. 32).

[32] T. Chou, C. Orlandi. "The Simplest Protocol for Oblivious Transfer". In: *Progress in Cryptology – LATINCRYPT 2015*. Springer International Publishing, 2015, pp. 40–58. DOI: 10.1007/978-3-319-22174-8_3 (cit. on pp. 28–30).

[33] C.-t. Chu, S. K. Kim, Y.-a. Lin, Y. Yu, G. Bradski, K. Olukotun, A. Y. Ng. "Map-Reduce for Machine Learning on Multicore". In: *Advances in Neural Information Processing Systems 19*. Ed. by B. Schölkopf, J. C. Platt, T. Hoffman. MIT Press, 2007, pp. 281–288. URL: http://papers.nips.cc/paper/3150-map-reduce-for-machine-learning-on-multicore.pdf (cit. on p. 37).

[34] DeepSpeed. *DeepSpeed*. 2020. URL: https://www.deepspeed.ai/ (cit. on p. 38).

[35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2009. DOI: 10.1109/cvpr.2009.5206848 (cit. on p. 70).

[36] W. Diffie, M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. DOI: 10.1109/tit.1976.1055638 (cit. on pp. 29, 32).

[37] P. Domingos. "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10 (Oct. 2012), pp. 78–87. DOI: 10.1145/2347736.2347755 (cit. on p. 13).

[38] P. Dwivedi. *Understanding and Coding a ResNet in Keras*. Apr. 1, 2019. URL: https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33 (cit. on pp. 71, 72).

[39] C. Dwork. "Differential Privacy". In: *Automata, Languages and Programming*. Ed. by M. Bugliesi, B. Preneel, V. Sassone, I. Wegener. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12. ISBN: 978-3-540-35908-1 (cit. on p. 42).

[40] C. Dwork, F. McSherry, K. Nissim, A. Smith. "Calibrating Noise to Sensitivity in Private Data Analysis". In: *Theory of Cryptography*. Springer Berlin Heidelberg, 2006, pp. 265–284. DOI: 10.1007/11681878_14 (cit. on p. 42).

[41] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. DOI: 10.1109/tit.1985.1057074 (cit. on p. 31).

[42] D. Evans, V. Kolesnikov, M. Rosulek. “A Pragmatic Introduction to Secure Multi-Party Computation”. In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), pp. 70–246. DOI: 10.1561/3300000019 (cit. on pp. 30, 32, 33).

[43] S. Even, O. Goldreich, A. Lempel. “A Randomized Protocol for Signing Contracts”. In: *Advances in Cryptology*. Ed. by D. Chaum, R. L. Rivest, A. T. Sherman. Boston, MA: Springer US, 1983, pp. 205–210. ISBN: 978-1-4757-0602-4 (cit. on p. 28).

[44] Facebook. *PyTorch*. 2020. URL: https://pytorch.org/ (cit. on p. 38).

[45] J. Fan, F. Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144. 2012 (cit. on p. 31).

[46] E. W. Forgy. “Cluster analysis of multivariate data: efficiency versus interpretability of classifications.” In: *Biometrics* 21.3 (1965), pp. 761–777. ISSN: 0006341X, 15410420. URL: http://www.jstor.org/stable/2528559 (cit. on p. 16).

[47] J. Freudiger, M. Xue. *Designing for Privacy - WWDC 2019*. June 5, 2019. URL: https://developer.apple.com/videos/play/wwdc2019/708/ (cit. on p. 13).

[48] C. Gentry. “Computing arbitrary functions of encrypted data”. In: *Communications of the ACM* 53.3 (Mar. 2010), pp. 97–105. DOI: 10.1145/1666420.1666444 (cit. on p. 31).

[49] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC ’09*. ACM Press, 2009. DOI: 10.1145/1536414.1536440 (cit. on p. 31).

[50] C. Gentry, S. Halevi. *Implementing Gentry’s Fully-Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Report 2010/520. https://eprint.iacr.org/2010/520. 2010 (cit. on p. 31).

[51] O. Goldreich, S. Micali, A. Wigderson. “How to play ANY mental game”. In: *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC ’87*. ACM Press, 1987. DOI: 10.1145/28395.28420 (cit. on p. 29).

[52] O. Goldreich. “Cryptography and cryptographic protocols”. In: *Distributed Computing* 16.2-3 (Sept. 2003), pp. 177–199. DOI: 10.1007/s00446-002-0077-1 (cit. on p. 29).

[53] S. Goldwasser, V. Goyal, A. Jain, A. Sahai. *Multi-Input Functional Encryption*. Cryptology ePrint Archive, Report 2013/727. https://eprint.iacr.org/2013/727. 2013 (cit. on p. 32).

[54] S. Goldwasser, S. Micali. “Probabilistic encryption”. In: *Journal of Computer and System Sciences* 28.2 (Apr. 1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9 (cit. on p. 33).

[55] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 22, 23).

[56] S. D. Gordon, J. Katz, F.-H. Liu, E. Shi, H.-S. Zhou. *Multi-Input Functional Encryption*. Cryptology ePrint Archive, Report 2013/774. https://eprint.iacr.org/2013/774. 2013 (cit. on p. 32).

[57] O. Gupta, R. Raskar. “Distributed learning of deep neural network over multiple agents”. In: *Journal of Network and Computer Applications* 116 (Aug. 2018), pp. 1–8. DOI: 10.1016/j.jnca.2018.05.003 (cit. on p. 39).

[58] A. Hard, K. Rao, R. Mathews, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, D. Ramage. "Federated Learning for Mobile Keyboard Prediction". In: *CoRR* abs/1811.03604 (2018). arXiv: 1811.03604. URL: http://arxiv.org/abs/1811.03604 (cit. on pp. 13, 41).

[59] L. Harn, C. Lin. "Strong (n, t, n) verifiable secret sharing scheme". In: *Information Sciences* 180.16 (2010), pp. 3059–3064 (cit. on p. 28).

[60] K. He, X. Zhang, S. Ren, J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV] (cit. on p. 70).

[61] R. Hecht-Nielsen. "Kolmogorov"s Mapping Neural Network Existence Theorem". In: 1987 (cit. on p. 20).

[62] J. Hempel. *Melinda Gates and Fei-Fei Li Want to Liberate AI from "Guys With Hoodies"*. May 4, 2017. URL: https://www.wired.com/2017/05/melinda-gates-and-fei-fei-li-want-to-liberate-ai-from-guys-with-hoodies/ (cit. on p. 13).

[63] J. Howard. *Imagenette*. URL: https://github.com/fastai/imagenette/ (cit. on p. 70).

[64] S. Ioffe, C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG] (cit. on p. 25).

[65] J. Kilian. "Founding crytpography on oblivious transfer". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing - STOC '88*. ACM Press, 1988. DOI: 10.1145/62212.62215 (cit. on p. 28).

[66] A. N. Kolmogorov. "On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition". In: *Dokl. Akad. Nauk SSSR* (June 20, 1957) (cit. on p. 20).

[67] J. Konecný, H. B. McMahan, D. Ramage, P. Richtárik. "Federated Optimization: Distributed Machine Learning for On-Device Intelligence". In: *CoRR* abs/1610.02527 (2016). arXiv: 1610.02527. URL: http://arxiv.org/abs/1610.02527 (cit. on p. 41).

[68] J. Konecný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, D. Bacon. "Federated Learning: Strategies for Improving Communication Efficiency". In: *CoRR* abs/1610.05492 (2016). arXiv: 1610.05492. URL: http://arxiv.org/abs/1610.05492 (cit. on p. 41).

[69] J. Konečný, B. McMahan, D. Ramage. *Federated Optimization:Distributed Optimization Beyond the Datacenter*. 2015. arXiv: 1511.03575 [cs.LG] (cit. on pp. 13, 41).

[70] A. Krizhevsky, G. Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009) (cit. on p. 64).

[71] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on p. 56).

[72] Y. LeCun. "Generalization and network design strategies". In: *Technical Report CRG-TR-89-4, University of Toronto* (1989) (cit. on p. 22).

[73] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. DOI: 10.1109/tit.1982.1056489 (cit. on p. 16).

[74] A. Lopez-Alt, E. Tromer, V. Vaikuntanathan. *On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2013/094. https://eprint.iacr.org/2013/094. 2013 (cit. on pp. 31, 32).

[75] J. MacQueen. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–297. URL: https://projecteuclid.org/euclid.bsmsp/1200512992 (cit. on p. 16).

[76] E. Makri. *CO6GC: LINEAR SECRET SHARING SCHEMES – LSSS | COSIC*. Apr. 17, 2020. URL: https://www.esat.kuleuven.be/cosic/blog/lsss/ (cit. on pp. 28, 47).

[77] G. Mann, R. McDonald, M. Mohri, N. Silberman, D. W. IV. “Efficient Large-Scale Distributed Training of Conditional Maximum Entropy Models”. In: *Neural Information Processing Systems (NIPS)*. 2009 (cit. on pp. 36, 37).

[78] J. McCarthy, M. L. Minsky, N. Rochester, C. E. Shannon. *A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE*. Aug. 31, 1955. URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html (cit. on p. 15).

[79] W. S. McCulloch, W. Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. DOI: 10.1007/bf02478259 (cit. on pp. 16–18).

[80] H. B. McMahan, G. Andrew. “A General Approach to Adding Differential Privacy to Iterative Training Procedures”. In: *CoRR* abs/1812.06210 (2018). arXiv: 1812.06210. URL: http://arxiv.org/abs/1812.06210 (cit. on p. 42).

[81] H. B. McMahan, E. Moore, D. Ramage, B. A. y Arcas. “Federated Learning of Deep Networks using Model Averaging”. In: *CoRR* abs/1602.05629 (2016). arXiv: 1602.05629. URL: http://arxiv.org/abs/1602.05629 (cit. on pp. 41, 56).

[82] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, B. A. y Arcas. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. 2016. arXiv: 1602.05629 [cs.LG] (cit. on p. 41).

[83] Merriam-Webster.com. *Cluster Analysis | Definition of Cluster Analysis by Merriam-Webster*. Aug. 1, 2020. URL: https://www.merriam-webster.com/dictionary/cluster%20analysis (cit. on p. 16).

[84] S. Micali, P. Rogaway. “Secure Computation”. In: *Advances in Cryptology — CRYPTO ’91*. Springer Berlin Heidelberg, 1991, pp. 392–404. DOI: 10.1007/3-540-46766-1_32 (cit. on p. 27).

[85] J. B. Nielsen, P. S. Nordholt, C. Orlandi, S. S. Burra. “A New Approach to Practical Active-Secure Two-Party Computation”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by R. Safavi-Naini, R. Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 681–700. ISBN: 978-3-642-32009-5 (cit. on p. 28).

[86] N. H. Phan, X. Wu, D. Dou. “Preserving Differential Privacy in Convolutional Deep Belief Networks”. In: (June 2017) (cit. on p. 75).

[87] M. G. Poirot, P. Vepakomma, K. Chang, J. Kalpathy-Cramer, R. Gupta, R. Raskar. *Split Learning for collaborative deep learning in healthcare*. 2019. arXiv: 1912.12115 [cs.LG] (cit. on p. 39).

[88] S. Rajbhandari, J. Rasley, O. Ruwase, Y. He. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. ArXiv. May 2020. URL: https://www.microsoft.com/en-us/research/publication/zero-memory-optimizations-toward-training-trillion-parameter-models/ (cit. on pp. 37, 38).

[89] S. Ramaswamy, R. Mathews, K. Rao, F. Beaufays. "Federated Learning for Emoji Prediction in a Mobile Keyboard". In: *CoRR* abs/1906.04329 (2019). arXiv: 1906.04329. URL: http://arxiv.org/abs/1906.04329 (cit. on pp. 13, 41).

[90] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečný, S. Kumar, H. B. McMahan. *Adaptive Federated Optimization*. 2020. arXiv: 2003.00295 [cs.LG] (cit. on pp. 42, 75).

[91] R. L. Rivest, L. Adleman, M. L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation, Academia Press* (1978), pp. 169–179 (cit. on p. 31).

[92] R. L. Rivest, A. Shamir, L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. DOI: 10.1145/359340.359342 (cit. on pp. 31, 32).

[93] G. Rometty, J. W. Owens, R. N. Haass. *A Conversation with Ginni Rometty*. Mar. 7, 2013. URL: https://www.youtube.com/watch?v=SUoCHC-i7_o (cit. on p. 13).

[94] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519 (cit. on pp. 16, 18).

[95] C. Rosset. *Turing-NLG: A 17-billion-parameter language model by Microsoft*. Feb. 13, 2020. URL: https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/ (cit. on p. 35).

[96] D. E. Rumelhart, G. E. Hinton, R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0 (cit. on p. 19).

[97] A. L. Samuel. "Some studies in machine learning using the game of Checkers". In: *IBM JOURNAL OF RESEARCH AND DEVELOPMENT* (1959), pp. 71–105 (cit. on p. 16).

[98] A. Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (Nov. 1979), pp. 612–613. DOI: 10.1145/359168.359176 (cit. on pp. 27, 28).

[99] M. Simon, E. Rodner, J. Denzler. *ImageNet pre-trained models with batch normalization*. 2016. arXiv: 1612.01452 [cs.CV] (cit. on p. 44).

[100] A. Singh, P. Vepakomma, O. Gupta, R. Raskar. *Detailed comparison of communication efficiency of split learning and federated learning*. 2019. arXiv: 1909.09145 [cs.LG] (cit. on p. 40).

[101] H. Steinhaus. "Sur la division des corps matériels en parties". In: *Bulletin de l'Académie Polonaise des Sciences* Cl. III — Vol. IV.12 (1956), pp. 801–804 (cit. on p. 16).

[102] G. J. Székely, M. L. Rizzo, N. K. Bakirov. "Measuring and testing dependence by correlation of distances". In: *The Annals of Statistics* 35.6 (Dec. 2007), pp. 2769–2794. ISSN: 0090-5364. DOI: 10.1214/009053607000000505. URL: http://dx.doi.org/10.1214/009053607000000505 (cit. on p. 40).

[103] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, C. Liu. *A Survey on Deep Transfer Learning*. 2018. arXiv: 1808.01974 [cs.LG] (cit. on p. 44).

[104] tensorflow. *tf.keras.Model | TensorFlow Core v2.3.0*. 2020. URL: https://www.tensorflow.org/api_docs/python/tf/keras/Model (cit. on p. 56).

[105] P. Vepakomma, O. Gupta, A. Dubey, R. Raskar. "REDUCING LEAKAGE IN DISTRIBUTED DEEP LEARNING FOR SENSITIVE HEALTH DATA". In: 2019 (cit. on pp. 39, 40).

[106] P. Vepakomma, O. Gupta, T. Swedish, R. Raskar. "Split learning for health: Distributed deep learning without sharing raw patient data". In: *CoRR* abs/1812.00564 (2018). arXiv: 1812.00564. URL: http://arxiv.org/abs/1812.00564 (cit. on pp. 39, 40).

[107] M. M. Waldrop. "News Feature: What are the limits of deep learning?" In: *Proceedings of the National Academy of Sciences* 116.4 (Jan. 2019), pp. 1074–1077. DOI: 10.1073/pnas.1821594116 (cit. on p. 20).

[108] J. Weizenbaum. "ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine". In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. ISSN: 0001-0782. DOI: 10.1145/365153.365168. URL: https://doi.org/10.1145/365153.365168 (cit. on p. 15).

[109] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, F. Beaufays. *Applied Federated Learning: Improving Google Keyboard Query Suggestions*. 2018. arXiv: 1812.02903 [cs.LG] (cit. on p. 13).

[110] A. C. Yao. "Protocols for secure computations". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE, Nov. 1982. DOI: 10.1109/sfcs.1982.38 (cit. on p. 27).

[111] A. C.-C. Yao. "How to generate and exchange secrets". In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, Oct. 1986. DOI: 10.1109/sfcs.1986.25 (cit. on pp. 29, 30).

List of Figures

List of Tables

List of Listings

List of Algorithms

A Further experiments on MNIST

In this section, further experiments are shown. The results do not provide any additional insights, but the data distribution is slightly different from those already shown.

A.1 Experiment M3: non-IID (3%)

As shown in Figure A.1, each party has two main digits, of which they have by far the most training images. Of the remaining digits, each party has 88% of its main digits‘ training data and 3% of the remaining digits‘ training data. Each experiment was performed five times. The corresponding results are summarized in Table A.1.

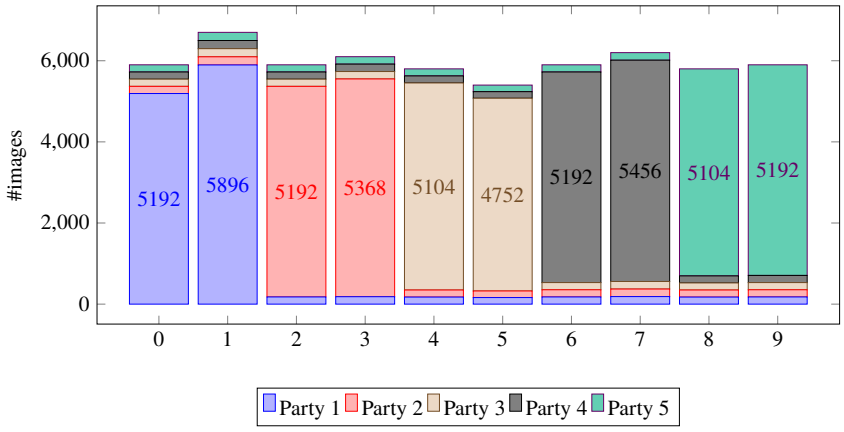

Figure A.1: Dataset distribution for experiment M3

approach	best result	worst result	median
SL	258	410	340
FO	164	231	198

Table A.1: Comparison of the results experiment M3.

A.2 Experiment M4: non-IID (1%)

As shown in Figure A.2, each party has two main digits, of which they have by far the most training images. Of the remaining digits, each party has 96% of its main digits‘ training data and 1% of the remaining digits‘ training data. Each experiment was performed five times. The corresponding results are summarized in Table A.2.

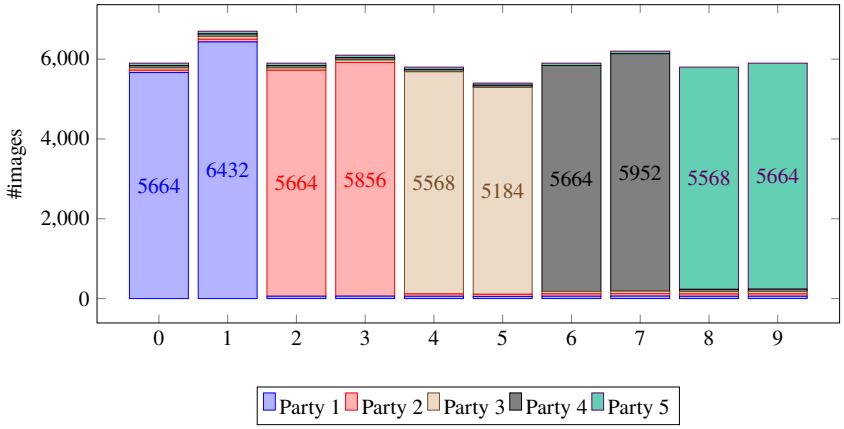

Figure A.2: Dataset distribution for experiment M4.

approach	best result	worst result	median
SL	370	478	387
FO	145	233	184

Table A.2: Comparison of the results experiment M4.

B Python implementations

B.1 Source code of the experiments

`transform_mnist.py` – Transform MNIST dataset to NumPy array:

```
import os
import struct
import numpy as np

path='./work_data'
train_labels_path = os.path.join(path, 'train-labels-idx1-ubyte')
train_images_path = os.path.join(path, 'train-images-idx3-ubyte')
test_labels_path = os.path.join(path, 't10k-labels-idx1-ubyte')
test_images_path = os.path.join(path, 't10k-images-idx3-ubyte')

with open(train_labels_path, 'rb') as lbpath:
  magic, n = struct.unpack('>II', lbpath.read(8))
  y_train = np.fromfile(lbpath, dtype=np.uint8)

with open(train_images_path, 'rb') as imgpath:
  magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
  x_train = np.fromfile(imgpath, dtype=np.uint8).reshape(len(y_train), 784)

with open(test_labels_path, 'rb') as lbpath:
  magic, n = struct.unpack('>II', lbpath.read(8))
  y_test = np.fromfile(lbpath, dtype=np.uint8)

with open(test_images_path, 'rb') as imgpath:
  magic, num, rows, cols = struct.unpack(">IIII", imgpath.read(16))
  x_test = np.fromfile(imgpath, dtype=np.uint8).reshape(len(y_test), 784)

np.savez('./work_data/mnist.npz', x_train=x_train, y_train=y_train, x_test=x_test,
y_test=y_test)
```

`transform_cifar10.py` – Transform CIFAR-10 dataset to NumPy array.

```
import os
import numpy as np

# https://www.cs.toronto.edu/~kriz/cifar.html
def unpickle(file):
  import pickle
  with open(file, 'rb') as fo:
    dict = pickle.load(fo, encoding='bytes')
  return dict


path = './work_data/cifar-10-batches-py'
train = [unpickle(os.path.join(path, f'data_batch_{i}')) for i in range(1, 6)]

labels = train[0][b'labels'] + train[1][b'labels'] + train[2][b'labels'] + train[3][b'labels']
 + train[4][b'labels']
data = np.concatenate([train[0][b'data'], train[1][b'data'], train[2][b'data'], train[3][b'
data'], train[4][b'data']])

test = unpickle(os.path.join(path, 'test_batch'))

np.savez('./work_data/cifar10.npz', x_train=data, y_train=labels, x_test=test[b'data'], y_test
=test[b'labels'])
```

common.py – Common functions for MNIST and CIFAR-10:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models

def setup_tensorflow():
# https://github.com/tensorflow/tensorflow/issues/24496#issuecomment-592179648
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
  try:
    for gpu in gpus:
      tf.config.experimental.set_memory_growth(gpu, True)
      logical_gpus = tf.config.experimental.list_logical_devices('GPU')
      print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
  except RuntimeError as e:
    print(e)


def get_dataset_iid(party_count, seed, dataset='mnist'):
  if dataset == 'mnist':
    (train_images, train_labels), (test_images, test_labels) = load_mnist()
  elif dataset == 'cifar10':
    (train_images, train_labels), (test_images, test_labels) = load_cifar10()

  # Shuffle images and labels with same permutation
  rnd_state = np.random.RandomState(seed=seed)
  permutation = rnd_state.permutation(len(train_images))
  train_images1 = train_images[permutation]
  train_labels1 = train_labels[permutation]
  data = np.split(train_images1, party_count)
  labels = np.split(train_labels1, party_count)

  # Convert labels to categorical to be conform with the produced model results
  test_labels = tf.keras.utils.to_categorical(test_labels)
  for i in range(0, len(labels)):
  labels[i] = tf.keras.utils.to_categorical(labels[i])

  return (data, labels), (test_images, test_labels)


def get_dataset_non_iid(party_count, min_percentage, dataset='mnist', input_shape=[1, 28, 28,
1]):
  if dataset == 'mnist':
    (train_images, train_labels), (test_images, test_labels) = load_mnist()
  elif dataset == 'cifar10':
    (train_images, train_labels), (test_images, test_labels) = load_cifar10()

  data = []
  labels = []
```

```
51
52    main_classes_per_party = 10 // party_count
53    main_class_percentage = 100 - ((party_count - 1) * min_percentage)
54
55    begin_classes = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
56    max_classes = [ len(train_images[train_labels == 0])
57    , len(train_images[train_labels == 1])
58    , len(train_images[train_labels == 2])
59    , len(train_images[train_labels == 3])
60    , len(train_images[train_labels == 4])
61    , len(train_images[train_labels == 5])
62    , len(train_images[train_labels == 6])
63    , len(train_images[train_labels == 7])
64    , len(train_images[train_labels == 8])
65    , len(train_images[train_labels == 9])
66    ]
67    for party in range(party_count):
68      dataset = np.empty(input_shape)
69      labelset = []
70      for f in range(10):
71        if (f >= party * main_classes_per_party) and (f < (party + 1) * main_classes_per_party):
72          end_class = begin_classes[f] + main_class_percentage
73        else:
74          end_class = begin_classes[f] + min_percentage
75        dataset = np.concatenate((dataset, train_images[train_labels == f][max_classes[f] //
    100 * begin_classes[f]:max_classes[f] // 100 * end_class]))
76        labelset = np.concatenate((labelset, train_labels[train_labels == f][max_classes[f] //
    100 * begin_classes[f]:max_classes[f] // 100 * end_class]))
77        begin_classes[f] = end_class
78      data.append(dataset[1:])
79      labels.append(labelset)
80
81    # Convert labels to categorical to be conform with the produced model results
82    test_labels = tf.keras.utils.to_categorical(test_labels)
83    for i in range(0, len(labels)):
84      labels[i] = tf.keras.utils.to_categorical(labels[i], num_classes=10)
85
86    return (data, labels), (test_images, test_labels)
87
88
89  def get_mnist_model(input_shape):
90    model = models.Sequential()
91    model.add(layers.Conv2D(32, (5, 5), activation='relu', padding='same',
92    input_shape=input_shape))
93    model.add(layers.MaxPooling2D((2, 2)))
94    model.add(layers.Conv2D(64, (5, 5), activation='relu', padding='same'))
95    model.add(layers.MaxPooling2D((2, 2)))
96
97    model.add(layers.Flatten())
98    model.add(layers.Dense(512, activation='relu'))
99    model.add(layers.Dense(10, activation='softmax'))
100   model.summary()
101   return model
```

```
102
103
104 def get_cifar10_model(input_shape):
105   model = models.Sequential()
106   model.add(layers.Conv2D(32, (3, 3), padding='same', input_shape=input_shape))
107   model.add(layers.Activation('relu'))
108   model.add(layers.Conv2D(32, (3, 3)))
109   model.add(layers.Activation('relu'))
110   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
111   model.add(layers.Dropout(0.25))
112
113   model.add(layers.Conv2D(64, (3, 3), padding='same'))
114   model.add(layers.Activation('relu'))
115   model.add(layers.Conv2D(64, (3, 3)))
116   model.add(layers.Activation('relu'))
117   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
118   model.add(layers.Dropout(0.25))
119
120   model.add(layers.Flatten())
121   model.add(layers.Dense(512))
122   model.add(layers.Activation('relu'))
123   model.add(layers.Dropout(0.5))
124   model.add(layers.Dense(10))
125   model.add(layers.Activation('softmax'))
126   return model
127
128
129 def load_mnist():
130   with np.load('../work_data/mnist.npz') as f:
131     x_train, y_train = f['x_train'], f['y_train']
132     x_test, y_test = f['x_test'], f['y_test']
133   x_train = x_train.reshape(60000, 28, 28, 1)
134   x_train = x_train.astype('float32') / 255
135   x_test = x_test.reshape(10000, 28, 28, 1)
136   x_test = x_test.astype('float32') / 255
137
138   return (x_train, y_train), (x_test, y_test)
139
140
141 def load_cifar10():
142   with np.load('../work_data/cifar10.npz') as f:
143     x_train, y_train = f['x_train'], f['y_train']
144     x_test, y_test = f['x_test'], f['y_test']
145   x_train = x_train.reshape(50000, 32, 32, 3)
146   x_train = x_train.astype('float32') / 255
147   x_test = x_test.reshape(10000, 32, 32, 3)
148   x_test = x_test.astype('float32') / 255
149
150   return (x_train, y_train), (x_test, y_test)
```

`decentralized_sl.py` – Decentralized SL for MNIST and CIFAR-10:

```
import tensorflow as tf

import lib.common as common

PARAMS = {'learning_rate': 0.0001,
  'seed': 1234567890,
  'party_count': 5,
  'epochs': 1,
  'batch_size': 64,
  'validation_split': 0.2,
  'distribution': 'non-iid',
  'optimizer': 'adam',
  'loss': 'categorical_crossentropy',
  'target_acc': 0.70,
  'max_round_count': 125,
  'min_percentage': 5,
  'dataset': 'cifar10',
  'input_shape': [1,32,32,3]
}

common.setup_tensorflow()

if PARAMS['distribution'] == 'non-iid':
  (train_x, train_y), (validate_x, validate_y) =
common.get_dataset_non_iid(PARAMS['party_count'], PARAMS['min_percentage'],
PARAMS['dataset'], PARAMS['input_shape'])
else:
  (train_x, train_y), (validate_x, validate_y) =
common.get_dataset_iid(PARAMS['party_count'], PARAMS['seed'], PARAMS['dataset'])

for _ in range(5):
  if PARAMS['dataset'] == 'mnist':
    model = common.get_mnist_model(PARAMS['input_shape'])
  elif:
    model = common.get_cifar10_model(PARAMS['input_shape'])

  model.compile(optimizer=tf.keras.optimizers.Adam(PARAMS['learning_rate']),
    loss=PARAMS['loss'],
    metrics=['acc'])

  acc = 0
  rounds = 0
  while acc < PARAMS['target_acc'] and rounds < PARAMS['max_round_count']:
    for j in range(PARAMS['party_count']):
      train_images = train_x[j]
      train_labels = train_y[j]
      # MemoryLeak on GPU when using validation_split as model.fit-parameter
      if 0 < PARAMS['validation_split'] < 1:
        split_at = int(len(train_images[0]) * (1 - PARAMS['validation_split']))
        (train_images, val_images) = (train_images[split_at:], train_images[:split_at])
        (train_labels, val_labels) = (train_labels[split_at:], train_labels[:split_at])
```

```

      model.fit(train_images,
           train_labels,
           batch_size=PARAMS['batch_size'],
           epochs=PARAMS['epochs'],
           validation_data=(val_images, val_labels),
           shuffle=True)
      eval_history = model.evaluate(validate_x, validate_y)
      print(f'Evaluation loss: {eval_history[0]}')
      print(f'Evaluation acc: {eval_history[1]}')
      acc = eval_history[1]
      if acc >= PARAMS['target_acc']:
        break
    rounds = rounds + 1
  print(f'round count: {rounds}')
```

`decentralized_fo.py` – Decentralized FO for MNIST and CIFAR-10:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras.backend import clear_session
from tensorflow.keras.callbacks import Callback

import lib.common as common

PARAMS = {'learning_rate': 0.0001,
  'seed': 1234567890,
  'party_count': 5,
  'epochs': 1,
  'batch_size': 64,
  'validation_split': 0.2,
  'distribution': 'non-iid',
  'optimizer': 'adam',
  'loss': 'categorical_crossentropy',
  'target_acc': 0.70,
  'max_round_count': 500,
  'min_percentage': 5,
  'dataset': 'cifar10',
  'input_shape': [1,32,32,3]
}

common.setup_tensorflow()

if PARAMS['distribution'] == 'non-iid':
  (train_x, train_y), (validate_x, validate_y) =
common.get_dataset_non_iid(PARAMS['party_count'], PARAMS['min_percentage'],
PARAMS['dataset'], PARAMS['input_shape'])
else:
  (train_x, train_y), (validate_x, validate_y) =
common.get_dataset_iid(PARAMS['party_count'], PARAMS['seed'], PARAMS['dataset'])


for _ in range(5):
  if PARAMS['dataset'] == 'mnist':
    shared_model = common.get_mnist_model(PARAMS['input_shape'])
  elif:
    shared_model = common.get_cifar10_model(PARAMS['input_shape'])
  shared_model.compile(optimizer=tf.keras.optimizers.Adam(PARAMS['learning_rate']),
              loss=PARAMS['loss'],
              metrics=['acc'])
  shared_model.save("shared_model.h5")

  acc = 0
  rounds = 0
  while acc < PARAMS['target_acc'] and rounds < PARAMS['max_round_count']:
    # Initialize all_models
    all_models = []
    for j in range(PARAMS['party_count']):
```

```
      all_models.append(models.load_model("shared_model.h5"))

    for j in range(PARAMS['party_count']):
      model = all_models[j]
      train_images = train_x[j]
      train_labels = train_y[j]

      # MemoryLeak on GPU when using validation_split as model.fit-parameter
      if 0 < PARAMS['validation_split'] < 1:
        split_at = int(len(train_images[0]) * (1 - PARAMS['validation_split']))
        (train_images, val_images) = (train_images[split_at:], train_images[:split_at])
        (train_labels, val_labels) = (train_labels[split_at:], train_labels[:split_at])

      model.fit(train_images,
            train_labels,
            batch_size=PARAMS['batch_size'],
            epochs=PARAMS['epochs'],
            validation_data=(val_images, val_labels),
            shuffle=True)
      eval_history = model.evaluate(validate_x, validate_y)
      print(f'Evaluation loss (Set {j}): {eval_history[0]}')
      print(f'Evaluation acc (Set {j}): {eval_history[1]}')

    # update shared model
    for idx, layer in enumerate(shared_model.layers):
      weights = np.array([all_models[0].layers[idx].get_weights(),
                all_models[1].layers[idx].get_weights(),
all_models[2].layers[idx].get_weights(),      all_models[3].layers[idx].get_weights(),
all_models[4].layers[idx].get_weights()
                 ])
      means = weights.mean(axis=0)
      layer.set_weights(means)
    eval_history = shared_model.evaluate(validate_x, validate_y)
    print(f'Evaluation loss (common model): {eval_history[0]}')
    print(f'Evaluation acc (common model): {eval_history[1]}')
    acc = eval_history[1]
    shared_model.save("shared_model.h5")
    clear_session()
    rounds = rounds + 1
  print(f'round count {rounds}')
```

`decentralized_sl_imagenette.py` – Decentralized SL for Imagenette:

```
import multiprocessing
from multiprocessing import Pipe

import tensorflow as tf
import tensorflow_datasets as tfds
from numba import cuda
from tensorflow.keras import models
from tensorflow.keras.backend import clear_session
from tensorflow.keras.callbacks import Callback

import lib.common as common


PARAMS = {'learning_rate': 0.001,
      'epochs': 1,
      'batch_size': 64,
      'seed': 1234567890,
      'party_count': 5,
      'min_percentage': 0,
      'max_round_count': 200,
      'distribution': 'iid',
      'optimizer': 'adam',
      'loss': 'categorical_crossentropy'
    }


def load_dataset(batch_size, seed, start=0, end=100):
  train, val = tfds.load('imagenette/160px-v2',
                         split=[f'train[{start}%:{end}%]', 'validation'],
                         as_supervised=True,
                         shuffle_files=True,
                         read_config=tfds.ReadConfig(shuffle_seed=seed)
                         )

  train = train.map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE) \
                  .batch(batch_size) \
                  .prefetch(1)
  val = val.map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE) \
              .batch(batch_size) \
              .prefetch(1)
  return train, val


def load_validate(batch_size):
  val = tfds.load('imagenette/160px-v2', split=['validation'], as_supervised=True)
  return val[0].map(normalize_img,
            num_parallel_calls=tf.data.experimental.AUTOTUNE) \
                      .batch(batch_size) \
                      .prefetch(1)


```

```
def normalize_img(image, label):
  return tf.keras.applications.resnet.preprocess_input(tf.image.resize(image, (224, 224))),
       tf.one_hot(tf.cast(label, tf.int32), 10)


def get_model():
  inputs = tf.keras.Input(shape=(224, 224, 3))
  model = tf.keras.applications.ResNet50(weights=None, include_top=False)(inputs)
  model = tf.keras.layers.GlobalAveragePooling2D()(model)
  outputs = tf.keras.layers.Dense(10, activation='softmax')(model)
  model = tf.keras.Model(inputs, outputs)
  return model


def train(peer, round, conn):
  common.setup_tensorflow()
  train, validate = load_dataset(PARAMS['batch_size'],
                  PARAMS['seed'],
                  start=(100 // PARAMS['party_count']) * peer,
                  end=(100 // PARAMS['party_count']) * (peer + 1)
                 )
  model = models.load_model(f'shared_model.h5')
  model.fit(train, batch_size=PARAMS['batch_size'], epochs=PARAMS['epochs'])
  eval_history = model.evaluate(validate)
  print(f'Evaluation loss: {eval_history[0]}')
  print(f'Evaluation acc: {eval_history[1]}')
  conn.send(eval_history[1])
  conn.close()
  model.save('shared_model.h5')
  clear_session()
  cuda.close()


def initialize_model():
  common.setup_tensorflow()
  shared_model = get_model()
  shared_model.compile(optimizer=PARAMS['optimizer'],
             loss=PARAMS['loss'],
             metrics=['acc']
            )
  model.save('shared_model.h5')
  clear_session()
  cuda.close()


def is_finished(acc, acc_old, rounds):
  return acc < 0.75 and
       abs(acc - acc_old) > 0.000001 and
       rounds < PARAMS['max_round_count']


#
# # Train
```

```
for no_exp in range(1):
  p_init = multiprocessing.Process(target=initialize_model)
  p_init.start()
  p_init.join()
  acc = 0
  acc_old = -1
  rounds = 0
  while is_finished(acc, acc_old, rounds):
    for j in range(PARAMS['party_count']):
      print("#####\nSet:", j)
      # train "derived" models
      parent_conn, child_conn = Pipe()
      p_train = multiprocessing.Process(target=train, args=(j, rounds, child_conn))
      p_train.start()
      acc_old = acc
      acc = parent_conn.recv()
      p_train.join()
      if is_finished(acc, acc_old, rounds):
        break
    rounds = rounds + 1
  print(f'round count: {rounds}')
  print('cancelled: {abs(acc - acc_old) <= 0.000001}')
```

`decentralized_fo_imagenette.py` – Decentralized FO for Imagenette:

```
import multiprocessing
from multiprocessing import Pipe

import neptune
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
from numba import cuda
from tensorflow.keras import models
from tensorflow.keras.backend import clear_session
from tensorflow.keras.callbacks import Callback

import lib.common as common


PARAMS = {'learning_rate': 0.001,
      'epochs': 1,
      'batch_size': 32,
      'seed': 1234567890,
      'party_count': 5,
      'min_percentage': 1,
      'max_round_count': 200,
      'distribution': 'non-iid',
      'optimizer': 'adam',
      'loss': 'categorical_crossentropy'
     }


def get_dataset_disjunct(party, batch_size, validation_split, percentage):
  datagen = tf.keras.preprocessing.image.ImageDataGenerator(validation_split=validation_split,
                        preprocessing_function=tf.keras.applications.resnet.preprocess_input)
  if percentage == 0:
    path = f'../work_data/imagenette/train/disjunct/{party}'
  else:
    path = f'../work_data/imagenette/train/{percentage}_percent/{party}'
  return datagen.flow_from_directory(path, target_size=(224, 224),
         batch_size=batch_size, seed=PARAMS['seed'], shuffle=False)

def get_validation(batch_size):
  datagen = tf.keras.preprocessing.image.ImageDataGenerator(
                      preprocessing_function=tf.keras.applications.resnet.preprocess_input)
  return datagen.flow_from_directory(f'../work_data/imagenette/val',
                                  target_size=(224, 224),
                                  batch_size=batch_size,
                                  seed=PARAMS['seed'],
                                  shuffle=False)


def load_dataset(batch_size, seed, start=0, end=100):
  train, val = tfds.load('imagenette/160px-v2',
                         split=[f'train[{start}%:{end}%]', 'validation'],
```

```
                        as_supervised=True,
                        shuffle_files=True,
                       read_config=tfds.ReadConfig(shuffle_seed=seed)
                       )
  train = train.map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE) \
                                              .batch(batch_size) \
                                              .prefetch(1)

  val = val.map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE) \
                                              .batch(batch_size) \
                                              .prefetch(1)
  return train, val


def load_validate(batch_size):
  val = tfds.load('imagenette/160px-v2', split=['validation'], as_supervised=True)
  return val[0].map(normalize_img, num_parallel_calls=tf.data.experimental.AUTOTUNE) \
                   .batch(batch_size)


def normalize_img(image, label):
  return tf.keras.applications.resnet.preprocess_input(tf.image.resize(image, (224, 224))),
          tf.one_hot(tf.cast(label, tf.int32), 10)


def get_model():
  inputs = tf.keras.Input(shape=(224, 224, 3))
  model = tf.keras.applications.ResNet50(weights=None, include_top=False)(inputs)
  model = tf.keras.layers.GlobalAveragePooling2D()(model)
  outputs = tf.keras.layers.Dense(10, activation='softmax')(model)
  model = tf.keras.Model(inputs, outputs)
  return model


def train(peer):
  common.setup_tensorflow()
  if PARAMS['distribution'] == 'non-iid':
    train = get_dataset_disjunct(peer, PARAMS['batch_size'], 0.2, PARAMS['min_percentage'])
    validate = get_validation(PARAMS['batch_size'])
  else:
    train, validate = load_dataset(PARAMS['batch_size'], PARAMS['seed'],
                   start=(100 // PARAMS['party_count']) * peer,
                   end=(100 // PARAMS['party_count']) * (peer + 1)
                   )
  model = models.load_model('shared_model.h5')
  model.fit(train,
            steps_per_epoch=train.samples / PARAMS['batch_size'],
            batch_size=PARAMS['batch_size'],
            epochs=PARAMS['epochs']
            )
  eval_history = model.evaluate(validate, steps=validate.samples / PARAMS['batch_size'])
  print(f'Evaluation loss (Set {peer}): {eval_history[0]}')
  print(f'Evaluation acc (Set {peer}): {eval_history[1]}')
```

```
105   model.save(f'shared_model_{peer}.h5')
106   clear_session()
107   cuda.close()
108
109
110 def initialize_model():
111   common.setup_tensorflow()
112   shared_model = get_model()
113   shared_model.compile(optimizer=PARAMS['optimizer'],
114                  loss=PARAMS['loss'],
115                  metrics=['acc'])
116   shared_model.save('shared_model.h5')
117   clear_session()
118   cuda.close()
119
120
121 def update_model():
122   common.setup_tensorflow()
123   shared_model = models.load_model('shared_model.h5')
124   # Load all_models
125   all_models = []
126   for j in range(PARAMS['party_count']):
127     all_models.append(models.load_model(f'shared_model_{j}.h5'))
128   # update shared model
129   for idx, layer in enumerate(shared_model.layers):
130     weights = np.array([all_models[0].layers[idx].get_weights(),
131                 all_models[1].layers[idx].get_weights(),
132                 all_models[2].layers[idx].get_weights(),
133                 all_models[3].layers[idx].get_weights(),
134                 all_models[4].layers[idx].get_weights()
135                 ])
136     means = weights.mean(axis=0)
137     layer.set_weights(means)
138   shared_model.save('shared_model.h5')
139   clear_session()
140
141
142 def validate(conn):
143   common.setup_tensorflow()
144   shared_model = models.load_model('shared_model.h5')
145   validate = get_validation(PARAMS['batch_size'])
146   eval_history = shared_model.evaluate(validate, steps=validate.samples /
    PARAMS['batch_size'])
147   print(f'Evaluation acc (shared model): {eval_history[1]}')
148   conn.send(eval_history[1])
149   conn.close()
150   clear_session()
151
152
153 def is_not_finished(acc, acc_old, rounds):
154   return acc < 0.75 and (abs(acc - acc_old) > 0.000001) and rounds < PARAMS['max_round_count']
155
156
```

```
#
# # Train
for no_exp in range(2):
  p_init = multiprocessing.Process(target=initialize_model)
  p_init.start()
  p_init.join()
  acc = 0
  acc_old = -1
  rounds = 0
  while is_not_finished(acc, acc_old, rounds):
    print("\n\nRound:", rounds)

    # train "derived" models
    for j in range(PARAMS['party_count']):
      print("#####\nSet:", j)
      p_train = multiprocessing.Process(target=train, args=(j,))
      p_train.start()
      p_train.join()
    p_update = multiprocessing.Process(target=update_model)
    p_update.start()
    p_update.join()
    parent_conn, child_conn = Pipe()
    p_update = multiprocessing.Process(target=validate, args=(child_conn,))
    p_update.start()
    acc_old = acc
    acc = parent_conn.recv()
    p_update.join()
    rounds = rounds + 1
  exp.log_text('round count', str(rounds))
```

B.2 Python implementation of secure decentralized federated optimization (SecAvg)

`initialize_common_model.py` – Initialize common model for decentralized FO:

```
  import lib.common as common

  common_model = common.get_model()
  common_model.compile(optimizer='adam, loss='categorical_crossentropy', metrics=['acc'])
  common_model.save("init_model.h5")
```

`decentralized_fo.py` – Decentralized FO with `SecAvg`:

```
import numpy as np
import argparse
import os.path
import time
import logging

import lib.common as common
from tensorflow.keras import models
from tensorflow.keras.backend import clear_session

logging.getLogger('tensorflow').setLevel(logging.ERROR)


parser = argparse.ArgumentParser(description='decentralized federated learning.')
parser.add_argument('peers', type=int, help='The total count of peers.')
parser.add_argument('position', type=int, help='The position of the peer.')
parser.add_argument('-r','--rounds', dest='rounds', type=int, help='Number of learning rounds.
', default=5)

args = parser.parse_args()


def divide(weights, peers, rnd_gen):
    rn = [ rnd_gen.integers(1, 5, peers) for _ in range(len(weights))]
    for i in range(len(rn)):
        rn[i] = rn[i] / rn[i].sum()
    rn = np.array(rn).transpose()
    return rn * weights


def divide_layer(layer_weights, peers, rnd_gen=np.random.default_rng(time.time_ns())):
    w = [divide(np.ravel(weight), peers, rnd_gen) for weight in layer_weights]
    final_weights = [layer_weights for _ in range(peers)]
    for p in range(peers):
        for i in range(len(layer_weights)):
            final_weights[p][i] = w[i][p].reshape(layer_weights[i].shape)
    return final_weights


PARAMS = {'learning_rate': 0.001,
```

```
            'epochs': 1,
            'batch_size': 64,
            'validation_split': 0.2,
            'seed': 1234567890,
            'min_percentage': 0,
            'max_round_count': 3000,
            'distribution': 'non-iid',
            'optimizer': 'adam',
            'loss': 'categorical_crossentropy'
            }

if PARAMS['distribution'] == 'non-iid':
    (train_x, train_y), (validate_x, validate_y) = common.get_dataset_non_iid(args.peers, PARAMS['min_percentage'])
else:
    (train_x, train_y), (validate_x, validate_y) = common.get_dataset_iid(args.peers, PARAMS['seed'])

common.setup_tensorflow()

# Train
for round_count in range(args.rounds):
    if round_count == 0:
        model = models.load_model("init_model.h5")
    else:
        model = models.load_model(f"{round_count - 1}_avg_model_{args.position}.h5")

    history = model.fit(train_x[args.position], train_y[args.position], batch_size=PARAMS['batch_size'],
                        epochs=PARAMS['epochs'], validation_split=PARAMS['validation_split'])
    eval_history = model.evaluate(validate_x, validate_y)
    print(f"model (loss/acc): {eval_history[0]}/{eval_history[1]}")

    part_models = [models.clone_model(model) for p in range(args.peers)]

    for idx, layer in enumerate(model.layers):
        par_wt = np.array(divide_layer(np.array(layer.get_weights()), args.peers))
        for j in range(args.peers):
            part_models[j].layers[idx].set_weights(par_wt[j])

    for j in range(args.peers):
        part_models[j].save(f"{round_count}_part_model_{args.position}_{j}.h5")

    corresponding_files_present = False
    while not corresponding_files_present:
        print("Checking for present …files")
        corresponding_files_present = True
        for j in range(args.peers):
            if not os.path.isfile(f"{round_count}_part_model_{j}_{args.position}.h5"):
                corresponding_files_present = False
        if not corresponding_files_present:
            time.sleep(5)

```

```
    part_models = [models.load_model(f"{round_count}_part_model_{p}_{args.position}.h5") for p
 in range(args.peers)]

    part_avg_model = models.clone_model(model)
    for idx, layer in enumerate(part_avg_model.layers):
        weights = [part_models[p].layers[idx].get_weights() for p in range(args.peers)]
        layer.set_weights(np.array(weights).mean(axis=0))

    part_avg_model.save(f"{round_count}_part_avg_model_{args.position}.h5")

    corresponding_files_present = False
    while not corresponding_files_present:
        print("Checking for present avg …files")
        corresponding_files_present = True
        for j in range(args.peers):
            if not os.path.isfile(f"{round_count}_part_avg_model_{j}.h5"):
                corresponding_files_present = False
        if not corresponding_files_present:
            time.sleep(10)

    avg_models = [models.load_model(f"{round_count}_part_avg_model_{p}.h5") for p in range(
args.peers)]

    avg_model = models.clone_model(model)
    for idx, layer in enumerate(avg_model.layers):
        weights = [avg_models[p].layers[idx].get_weights() for p in range(args.peers)]
        layer.set_weights(np.array(weights).sum(axis=0))

    avg_model.compile(optimizer=PARAMS['optimizer'], loss=PARAMS['loss'], metrics=['acc'])
    eval_history = avg_model.evaluate(validate_x, validate_y)
    print(f"avg model (loss/acc): {eval_history[0]}/{eval_history[1]}")
    avg_model.save(f"{round_count}_avg_model_{args.position}.h5")

    clear_session()
```