Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Adversarial Training in Federated Learning using Constrained Optimization Methods

Jan König
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr.-Ing. Alexandra Dmitrienko
First Reviewer

Prof. Dr. Christian Kanzow
Second Reviewer

Torsten Krauß, M.Sc.
First Advisor

Submission
May 15, 2023

www.uni-wuerzburg.de

Abstract

Federated Learning (FL) is an approach to machine learning that facilitates the collaborative training of models and allows participants to keep their local and potentially sensitive data private. Due to its decentralized nature, FL is especially vulnerable to certain types of attacks. In a so-called backdoor attack, adversaries submit manipulated updates to the model aggregation process. For adversary-determined inputs, the newly aggregated model will then generate targeted false predictions. A common defense so far has been to measure the deviation between a global model and the model trained by a participant and reject models with a deviation above a certain threshold. Adversaries can counteract by adjusting their training objective in a way that penalizes large deviations. However, this merely encourages and not guarantees that the deviation threshold is not exceeded. As the main contribution, this thesis reformulates the adversarial training objective as a constrained optimization problem, a class of problems well-researched in mathematics that often can be solved by the Augmented Lagrangian Method. This eliminates the dilemma that adversaries face when having to decide between the effectiveness of their backdoor and remaining undetected, as is the case with current methods. A secondary, independent contribution is a scheme to detect FL participants training a hidden objective, such as a backdoor, for linear approximations of non-linear models (Neural Tangent Kernels). Finally, to evaluate the effectiveness of the proposed attack- and defense mechanisms, this thesis evaluates them against several existing state-of-the-art backdoor types (e.g., Semantic backdoor).

Zusammenfassung

Federated Learning (FL) ist ein Machine Learning Ansatz, der das kollaborative Trainieren von Modellen ermöglicht und es den Teilnehmern erlaubt, ihre lokalen und potenziell sensiblen Daten privat zu halten. Aufgrund seiner dezentralen Art ist FL besonders anfällig für bestimmte Attacken. Bei einer sogenannten Backdoor-Attacke (Hintertür) übermitteln Angreifer manipulierte Updates für den Aggregationsprozess von Modellen. Bei Eingaben, die von den Angreifern im Vorfeld bestimmt werden, liefert das neu aggregierte Modell nun gezielte, falsche Vorhersagen. Eine übliche Verteidigung hiergegen war es bisher, die Abweichung zwischen einem globalen Modell und dem trainierten Modell eines Teilnehmers zu messen und Modelle auszusortieren, welche Abweichungen über einem bestimmten Grenzwert aufweisen. Die Angreifer können dies umgehen, indem sie ihr Training Objective derartig adjustieren, dass große Abweichungen bestraft werden. Jedoch schafft dies lediglich einen Anreiz und ist keine Garantie dafür, dass der Grenzwert für Abweichungen nicht überschritten wird. Der Hauptbeitrag dieser Arbeit besteht darin, das Training Objective der Angreifer als restringiertes Optimierungsproblem zu formulieren. Dies ist eine Klasse von Problemen, welche in der Mathematik gut erforscht ist und oftmals mit der Augmented-Lagrangian-Methode gelöst werden kann. Dies beseitigt das Dilemma, in dem sich Angreifer befinden, wenn sie sich zwischen der Effektivität ihrer Hintertür und dem Umgehen der Verteidigung entscheiden müssen, wie es bei den derzeitigen Angriffsmethoden der Fall ist. Ein sekundärer, vom ersten unabhängiger Beitrag ist ein Schema zur Erkennung von Teilnehmern in FL, welche ein verstecktes Training Objective, wie z.B. eine Hintertür, für lineare Approximationen von nichtlinearen Modellen (Neural Tangent Kernels) in ihren Trainingsprozess einbauen. Um die Effektivität des vorgeschlagenen Angriffs und der Verteidigung auszuwerten, werden diese in Zusammenhang mit mehreren modernen Backdoor-Typen (z.B. Semantic Backdoor) getestet.

Contents

1. Introduction

Federated Learning (FL) is a collaborative and distributed machine learning (ML) approach initially proposed by researchers at Google Inc. in a paper from 2017 [1]. Applicable for many types of ML models, it was first introduced for gradient descent trained models like neural networks (NNs). It can be applied in scenarios where an abundance of computational resources for model training is needed that, however, is out of direct control of the model engineers, like potentially millions of mobile devices belonging to customers. Mobile devices are not only capable of providing the necessary computational resources for model training but are also a treasure trove of data generated during real-world usage. One of the main features distinguishing FL from previously proposed decentralized learning approaches is the consideration of explicitly not independent and identically distributed (*non-iid*) data among various participants allowing FL to graduate from lab-like scenarios to real-world use cases [2]. With a growing concern for the privacy of user data, many governments across the world have enacted laws that strictly regulate the collection, storage, and use of personal data by corporations. Examples of these laws include the *General Data Protection Regulation* (GDPR) in the EU [3], the *Personal Information Protection and Electronic Documents Act* (PIPEDA) in Canada [4], and the *Health Insurance Portability and Accountability Act* (HIPAA) in the US [5]. With the imperative of data privacy as the primary motivation in the development of FL architectures, FL lends itself particularly well to scenarios where protecting sensitive user data is paramount. Proposed use cases include but are not limited to autonomous driving [6], evaluating healthcare data [7], and mobile keyboard typing predictions [8]. The latter case has even seen deployment in production for Google's Gboard service [9].

The original paper [1] sparked a lot of interest in further research concerning not only further development but also aspects of security. Key questions that come to mind are: *How can user data be protected?* and *How secure is FL itself against adversarial users?* While the former question has already been answered in related work (cf. Section 3.2), it is the latter question that is still open-ended and the starting point for this thesis. As FL is vulnerable to so-called backdoor attacks, an adversary in control of one or even multiple user devices participating in the federation can inject manipulated model updates into the aggregation process. This leads the aggregated model to make false predictions for specific inputs determined by the adversary [10]. This is usually achieved by first manipulating the data used for training. As the term *backdoor* suggests, the adversary aims to remain undetected (*stealthiness*). A simple defense against backdoors consists of comparing the distances between the global model to model updates submitted by a client [10, 11]. An

adversary can attempt to bypass this defense by adding the model distance to the global model as a secondary objective to be minimized during the local model training procedure. The secondary objective is then weighted with a parameter $\alpha \in (0, 1)$. This, however, introduces a dilemma, as the adversary needs to choose between either obtaining a model with a perfectly efficient backdoor or limiting the model distance to remain stealthy. The adversary tries to find a balance by searching for the correct α value, which can become a time-consuming process. The first research question **(RQ1)** this thesis tackles is, therefore: *"Can the dilemma arising from the need for an α parameter be eliminated by formulating anomaly terms as hard constraints?"* This question can be answered in the affirmative:

As the main contribution, this thesis successfully eliminates said dilemma by reformulating the secondary objective as a hard constraint and employing a method from the field of Constrained Optimization in Mathematics. Mathematically speaking, the approach with the α parameter is similar to using the penalty method for constrained optimization [12]. When reformulating the secondary objective of limiting the model distance as a hard constraint, the adversary can make use of more sophisticated methods like the *Augmented Lagrangian Method* (ALM) [13, 14, 15, 16]. Ideally, with the ALM, the adversary is able to circumvent the dilemma by training a model containing an efficient backdoor while satisfying the constraint without the need for any additional hyperparameters. In particular, he no longer needs to search for an optimal α parameter.

An adversary able to adapt to distance metrics might also formulate the performance of the installed backdoor as a constraint in an attempt to ensure a certain level of backdoor accuracy when training the model. A second research question **(RQ2)**, therefore, asks: *"Is an adversary able to install a backdoor with pre-defined accuracy by formulating its performance as a mathematical constraint?"* In regards to this question, however, this thesis gives insufficient answers, as experiments hereto remained inconclusive.

In addition to the adversarial point of view, this thesis also regards FL from the defensive side. As the weights of an NN experience little change in the later stages of training [17], one can approximate the NN with a linear classifier once it has learned the most important features. The linear classifier then replaces the NN as the main model. The transition to a linear classifier expedites the model training process while further improving the performance of the original model. One type of approximation, the Neural Tangent Kernel (NTK) [18], has since been successfully implemented in an FL setting [19]. One interesting property of linear classifiers is that they are trained by solving a linear least squares problem [20] for which a *unique* minimum can be found. This is in contrast to NNs, where generally, during model training, no minimum at all is found, and the training process is halted once the model reaches a satisfactory performance. As the location of the minimum depends on the data used for training, any modification to the data would shift the location of the minimum. As an adversary aims to remain stealthy, he is expected to refrain from sharing his poisoned data used for model training. By loosening the restrictions on data privacy, a novel defense might now be able to detect this shift in the location of the minimum. A last research question **(RQ3)** now asks: *"Can a defense be found that detects adversaries attempting to install a backdoor into an NTK model?"* Like RQ1, this question can be answered affirmatively as well.

As the secondary contribution, this thesis exploits this property of linear classifiers by outlining a proof of concept for a defense that can detect if a potential adversary has trained his model with different data than reported. This defense only works under the assumption that user data is no longer kept strictly private. While this is a direct contradiction to the original purpose of FL, scenarios are imaginable where FL is primarily used for the ability to distribute the model training process. An organization wanting to spread the computational load for training a model across multiple of its own servers might

employ this defense to detect if one of its servers has been compromised. Future research could potentially improve the outlined defense, such that it can detect adversaries while maintaining the privacy of user data, e.g., by developing a zero-knowledge proof.

In summary, this thesis considers FL from an adversarial and defensive perspective in two separate parts. As a rule of thumb, the following chapters first take on the side of an adversary before explaining the defensive approach. In two parts, both the attack and defense proposed in this section were successfully implemented as part of a simulation of an FL system. To give more insights, a background section (Sec. 2) (re-)introduces the most essential concepts of adversarial FL before the following section positions this thesis in the landscape of related work from other researchers (Sec. 3). The approach section (4) explains the newly proposed attack and defense mechanisms in more detail. Sec. 5 then defines the FL setup and adversarial model. The optimization problems solved by the adaptive adversary and NTK training procedure are derived in a mathematical section (Sec. 6). Both implementations used for conducting experiments are explained in section (Sec. 7). Finally, an evaluation (Sec. 8) with state-of-the-art backdoor attacks, like the semantic backdoor [10] and standardized benchmark datasets like CIFAR-10 [21] gave overwhelmingly positive feedback for both the attack and defense, motivating further research.

2. Background

The following chapter contains a brief re-introduction of the most important concepts of machine learning that find application in this thesis. While a general understanding of ML, especially NNs, is assumed, they will be re-introduced (Sec. 2.1). The models for the adversarial part of this project are subject to constraints and trained with methods of constrained optimization for which most already existing research has been conducted in a pure math setting. Furthermore, Federated Learning will also be introduced (Sec. 2.3), as it defines the setting in which all of the models will be trained. Security-related terminology will be introduced as well (Sec. 2.4) to explain adversaries and their goals. Finally, Neural Tangent Kernels (NTKs), which are linear approximations of more complex models, will be introduced (Sec. 2.6), as they serve as the basis for the defensive side of this project.

2.1 The Goal of Machine Learning

In his book, Geron [22] describes ML as the development of a computer program that learns from data. This is achieved by curating a dataset comprised of training samples from which the program extracts knowledge. The part of the program that handles the learning and makes predictions is referred to as an ML model, as it encapsulates information about the real world as described by the data and creates a generalized image (i.e., model) of it. The learning can take place in either a supervised [23] or unsupervised [24] manner, depending on whether the training data is labeled. For example, images in the CIFAR-10 dataset [21] carry a descriptive label like airplane, bird, automobile, cat, etc. Since the models in this project predict labels of images, the focus is entirely on supervised learning.

In deep learning, a particular branch of machine learning, the models take on the shape of neural networks. A neural network is a mathematical function with a large number of parameters that are adjusted during the training stage to increase the prediction performance on a specific task. For image classification, the model takes an image as input and makes a prediction in the form of a label that best describes the input data. As models, upon initialization, produce random outputs on any input, they need to be trained to increase the prediction performance (also called "model" performance or model/prediction "accuracy"). During the training stage, the model first produces predictions on a large amount of data. Then, in supervised learning, the predictions are compared with the gold standard of the labeled dataset to compute a so-called prediction error ("loss"). Then an optimization algorithm, e.g., SGD [25], tries to optimize the model weights so that the loss is minimized. For evaluation, a separate dataset is used to ensure that the model has

indeed learned to recognize shapes and semantics as opposed to simply memorizing the images seen during training, meaning it has sufficiently generalized its knowledge. The general process of training an ML model is visualized by Fig. 2.1.

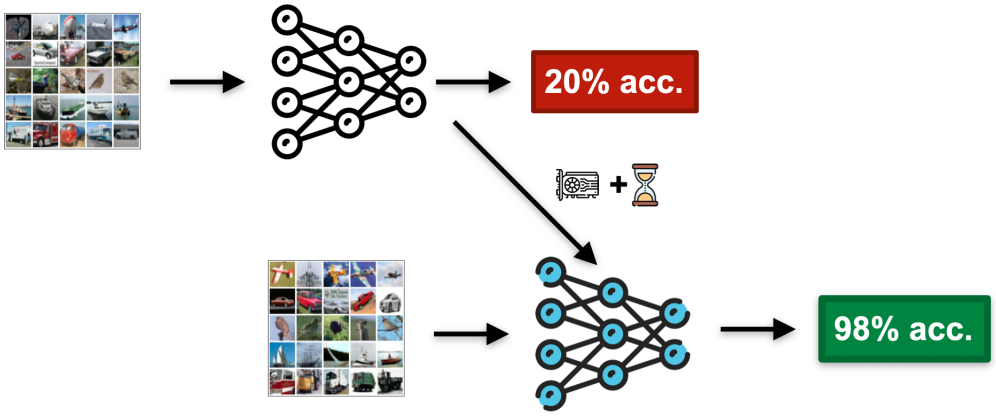


Figure 2.1: Desired outcome of training a neural network: After training a model that performs poorly on training data, the model should perform well on a separate dataset.

2.2 Training Deep Learning Models

All models in this project are deep neural networks which are models that can be written as the composition of multiple linear and non-linear functions to form a single, very complex function with up to millions of parameters (e.g., $\approx$ 11.2 million for ResNet18 [26]). The goal when training the model is to tweak these parameters such that the function predicts the correct output in as many cases as possible, which we refer to as a high model accuracy (MA, Eq. 2.1).

$$\text{MA} = \frac{\text{number of correctly predicted outputs}}{\text{total number of predictions made}} \tag{2.1}$$

In the realm of supervised deep learning, this can be achieved by calculating the model output, comparing it to the labels of the training set, and then calculating the error. The error now needs to be minimized, resulting in an unconstrained optimization problem which is usually solved using algorithms like SGD [25] or Adam [27]. The algorithm is then tasked with minimizing the loss of the model, which can be expressed as a mathematical function.

2.2.1 Models as Mathematical Functions

When training an NN, one needs to express the input data in numerical form. For instance, given an 8-bit RGB image of 32x32 pixels as in CIFAR-10 [21], one can express the image as a 3D vector of size 3x32x32 with integer values from 0 to 255. For simpler models, this 3D vector can be reshaped into a 1D array of length $3 \cdot 32 \cdot 32 = 3072$. The target labels are then also represented as integer numbers, e.g., car: 0, train: 1, bike: 2, ... In the case of a text prediction task, one can first define a corpus of admissible words and then assign every word in that corpus to a number. Now a sentence can also be represented as a 1D vector of integers and the prediction target once again as a single integer. Oftentimes the labels are converted to a so-called one-hot encoding [28]. Here, the label is represented as a vector that is 1 at the label index and 0 elsewhere: For a dataset with 3 different zero-based labels, the label bike = 2 would become $(0, 0, 1)$.

A model can now be described as a mathematical function $y\colon x \mapsto y(x, w)$ that takes an input vector x and a vector of parameters w (also called *weights*) and generates a prediction

$y(x, w)$. In most models, the function is a pairwise composition of functions $y_i^w(x)$ (also called *layers*) that depend on parameters and non-linear functions. The non-linear, so-called activation functions have no parameters and continuously map the output of the previous layer to a vector of the same dimension.

An example of a layer is the affine transformation, also called fully-connected layer:

$$y_i^w(x) = A_i \cdot x + b_i \tag{2.2}$$

for some arbitrary matrix $A_i \in \mathbb{R}^{\text{out} \times \text{in}}$ and bias vector $b_i \in \mathbb{R}^{\text{out}}$. In this case, the entries of A_i and b_i make up the parameters of y_i^w. The vector w can be seen as a short-hand for a vector comprised of all parameters of all layers ($w = (A_1, b_1, \cdots A_k, b_k)$). The layers must have matching input and output dimensions for the composition to work, as the activation functions do not change the dimensionality of the input. Instead, they element-wise transform the output of the previous layer. Common choices for activation functions are the sigmoid curve [29], ReLU [30], and the hyperbolic tangent [31]. The softmax function [32] is oftentimes used as the last activation layer, as it transforms the elements of the output vector into values that can be interpreted as a probability distribution of output labels. The highest probability then determines the model prediction.

A simple k-layer feedforward neural network is constructed by forming the composition of all layers.

$$y(x, w) = \sigma_k \circ y_k^w \circ \cdots \circ \sigma_2 \circ y_2^w \circ \sigma_1 \circ y_1^w(x) \tag{2.3}$$

The individual layers are the building blocks of the model and can be changed to form more complex models. The model is often identified by its weights by saying "the model w". A popular choice for image processing are convolutional layers [33] that take an image x in matrix shape as input and perform a discrete 2D convolution [34] on it by having a kernel stride over the input vector like a sliding window. In that case, the convolutional kernel, together with the bias vector, contains all of the layer's parameters.

Other common types of layers are batch normalization layers [35, 36] that transform a given input vector such that its components have zero mean and unit variance, meaning they are normalized with the mean and variance measured over multiple training examples. Additionally, max pooling layers [37] run a sliding window over the input, similar to a convolution, yet only keep the maximum component and discard the rest. The purpose of these layers is to keep the numerical information flowing through the model within small orders of magnitude (batch normalization) and to discard unnecessary information that does not impact model performance, hence expediting the training process (max pooling).

State-of-the-art model architectures (like ResNet [26]) are so complex that it becomes practically impossible to write its functional definition as a single equation while still retaining an understanding of how the model functions. This is why modern architectures are instead drawn as flowcharts (Figure 2.2). Nonetheless, the essential observation is that no matter how complicated or intimidating the functional definition may look, it is always - in theory at least - possible to define a model via a function of type Eq. 2.3. In the end, the model function might turn out to be highly non-linear and non-convex [38] with a number of trainable parameters that can easily exceed 10 million (ResNet18 [26]). However, the model function is still differentiable, which is the property on which the entire training process hinges.

2.2.2 Model training as Unconstrained Optimization Problem

In order to train the model, one first has to define a metric on which the model performance can be evaluated. For classification tasks, one mainly cares about model accuracy (MA).

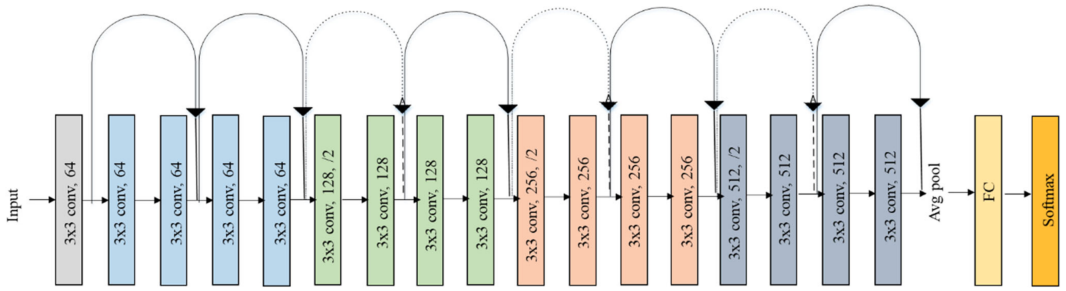


Figure 2.2: Visualization of ResNet18 architecture [26]. Figure taken from Razman et al. [39]

This metric, however, is not differentiable with respect to the model weights and balances. Therefore, one defines the loss function, which gives an estimate of the error made by the model when making predictions. Then, the main goal of model training is to minimize this loss function by changing the model parameters. A simple choice for a loss function is the Euclidean distance between the model output $y(x_i, w)$ and the true label $y_i = (y_i^1, \ldots, y_i^k) = (0, \ldots, 1, \ldots, 0)$ of training sample x_i as one-hot encoded vector. The overall loss is now the sum over all training examples $\mathbf{x} = (x_1, \ldots, x_n)$ with labels $\mathbf{y} = (y_1, \ldots, y_n)$

$$f(\mathbf{x}, \mathbf{y}, w) = \sum_{i=1}^{n} f(x_i, y_i, w) = \sum_{i=1}^{n} \|y_i - y(x_i, w)\|_2 \tag{2.4}$$

Another common loss function for classification tasks is the categorical cross-entropy loss [40]

$$f(\mathbf{x}, \mathbf{y}, w) = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_i^j \cdot \log\big(y(x_i, w)^j\big) = -\sum_{i=1}^{n} \log\Big(y(x_i, w)^{l_i}\Big) \tag{2.5}$$

where l_i is the integer label of training sample x_i. With the goal of minimizing the loss function by modifying the model parameters, the model training becomes the mathematical optimization problem

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \tag{2.6}$$

As great attention has been spent on keeping the model differentiable with respect to its parameters, gradient-based optimization methods are the first to come to mind. For the simple gradient descent algorithm, one first sets a learning rate η (commonly abbreviated as lr) and calculates the gradient of the loss function wrt. all model parameters and then updates the model parameters by taking a step in the negative direction of the gradient, as it determines the direction of the steepest descent.

$$w_{t+1} = w_t - \eta \cdot \nabla_w f(\mathbf{x}, \mathbf{y}, w) \tag{2.7}$$

The efficiency of the gradient descent method can be improved by splitting the set of training samples into smaller batches $\{(\mathbf{x}_1, \mathbf{y}_i) \ldots, (\mathbf{x}_n, \mathbf{y_n})\}$ and take gradient steps by iterating through the batches when calculating $\nabla_w f(\mathbf{x_i}, \mathbf{y}_i, w)$. This algorithm is called stochastic gradient descent (SGD) [25] and is the backbone of deep learning. While there are more sophisticated algorithms, like Newton's method [41] that rely on second-order derivatives, SGD-based algorithms like Adam [27] or SGD with Nesterov momentum [42] and weight decay [43] remain the default choice for model optimization.

One of the main challenges of deep learning is that not only the gradient computation but also the calculation of the objective function $f(\mathbf{x}, w)$ itself (forward pass) can be costly. However, SGD already decreases the computation burden by only calculating forward

passes for smaller batches, and the backpropagation algorithm [44] describes a very efficient way to calculate the gradients. Modern frameworks like PyTorch [45] implement all of the previously mentioned algorithms and provide an interface allowing programmers to construct and train deep learning models using building blocks, making it unnecessary to implement most basic algorithms from scratch.

2.3 Federated Learning

Federated Learning (FL) is a decentralized approach to machine learning [1] in which a group of clients collectively trains a model over multiple rounds. Every client has his own dataset which is not shared with other clients, and at the end of every round, all client models are aggregated to a central, global model which is then redistributed to every client before the next round of training starts. In the language of optimization, one tries to minimize the loss function by having multiple clients attempt to optimize the model in parallel based on their local dataset before aggregating their resulting model parameters. Figure 2.3 gives a basic schematic overview of an FL setup.

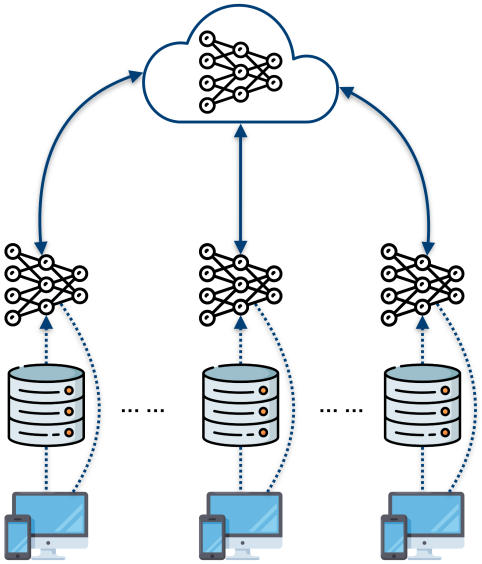


Figure 2.3: Example of an FL setup where clients with separate datasets train individual models that are aggregated to a single central model

2.3.1 Federated Optimization

For every round of training, a server shares the current state of the global model with a randomly selected fraction C from a pool of clients K participating in the federation. Every selected client is now tasked with continuing to train the model on its own for one round with its own dataset, which is shared with neither of the other clients nor the server. In a so-called *cross-device* scenario, a number of assumptions can be made about the dataset of every client:

- **Non-IID** The dataset belonging to every client is based on real-world device usage, and therefore the data of a single client is not representative of the rest of the group. The data is therefore not considered to be independent and identically distributed, or non-iid for short.
- **Unbalanced** As device usage may vary by client, some may have accumulated vastly more data than others.

- **Massively distributed** The number of clients is expected to exceed the average number of data samples per client.

A similar *cross-silo* scenario considers a smaller number of clients with large quantities of data.

While in theory, FL is an option for distributed training of any ML model type that is trained via a loss function, it is mainly applied to NNs. In this case, let $f(x_i, w)$ be the prediction loss of a model with parameters w for a sample x_i. Now let $\mathbf{x}_i$ be the set of n_k different training samples held by client $k \in K$ and $\mathbf{y}_i$ be their respective labels. Further let n be the total number of training samples held by all clients. Then the objective function becomes

$$f(\mathbf{x}_i, \mathbf{y}_i, w) = \sum_{k=1}^{K} \frac{n_k}{n} f(\mathbf{x}_k, \mathbf{y}_k, w) \quad \text{where} \quad f(\mathbf{x}_k, \mathbf{y}_k, w) = \frac{1}{n_k} \sum_{x_i \in \mathbf{x_k}} f(x_i, y_i, w) \tag{2.8}$$

Since the data is split explicitly non-iid between clients, it cannot be assumed that a model trained on $\mathbf{x_k}$ converges to a global model trained on the entire dataset $\mathbf{x}$.

2.3.2 Federated Averaging

As most deep learning models have relied on stochastic gradient descent (SGD) based methods for updating, FL is built around the SGD update mechanism. At first, the global model can be randomly initialized as usual. Then, after a C fraction of clients has received the parameters for the current model w_t, every client individually trains the model for one iteration, computes the gradient $g_k = \nabla f(\mathbf{x_k}, w_t)$ and sends it back to the server, which updates the global model using the aggregated gradients to

$$w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k \tag{2.9}$$

Here, η denotes the global learning rate (lr). Since this is equivalent to letting every client update its model by itself (resulting in w_{t+1}^k) and aggregating thereafter

$$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k \tag{2.10}$$

one can also make multiple update steps locally, allowing for batch learning via SGD. It is important, however, to always distribute the current state of the global model to every client at the beginning of every aggregation round so that every client can continue training from the same state.

2.4 Adversaries and Backdoors

In addition to the advantages offered by FL, there are, however, some downsides. Its major problem is that it relies on the trustworthiness of the clients. If one client has ulterior motives, like feeding the model false information, he can do so to his content, as the server simply aggregates client models without questioning their integrity. Attacks that induce intentional wrongful predictions (e.g., misclassifications in classification tasks or erroneous word predictions in natural language tasks) can be categorized like the following:

- **Targeted / Untargeted**
  For targeted attacks, an adversary selects a false label (target label) he wants the network to output for a given sample. This false prediction is caused by a trigger

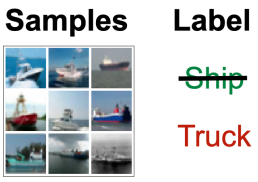


Figure 2.4: The figure shows a so-called label-swap backdoor, where all images of boats are supposed to be classified as "trucks". The original label of "boats" constitutes the backdoor trigger in this example.

in the sample he wants the model to misclassify. The adversary then aims to train a model with high prediction accuracy on the triggered samples. Furthermore, the adversary wants to remain undetected ("stealthiness"). Therefore, he simultaneously tries to retain the original MA (now also called "main accuracy") as an additional goal. This type of attack is also referred to as a backdoor, and the prediction accuracy on triggered samples is called backdoor accuracy (BA). In an untargeted attack, the adversary aims to reduce the MA. Therefore, any false prediction made by the model is considered a success.

- **Black-Box / White-Box**
  During a black-box attack, the adversary does not have access to information about the model architecture, model weights, hyperparameters, gradient updates, etc., as opposed to a white-box scenario where the adversary has access to all the available information about the model.
- **Evasion / Poisoning**
  The attack can happen at either inference-time (evasion), where the adversary is confronted with a pre-trained model he is unable to alter, or training-time (poisoning), where the adversary can exert influence over the training process of the model.

A standard method for an adversary to install a backdoor is to manipulate his data (data poisoning, Fig. 2.4), which is then used to train the model (model poisoning). A poisoned model is also referred to as "malicious" in contrast to a benign model. Moreover, an adversary is able to poison additional models by obtaining control over multiple clients (e.g., via malware). A schematic of an adversary infiltrating an FL system can be seen in Fig. 2.5. To combat maliciously trained models, there have been many proposals for defense mechanisms, which will be discussed in more detail in the related work section (Sec. 3.4).

2.5 Adversarial Dilemma

Many defense approaches that will be discussed in the related work section (Sec. 3.4) are based on the notion that malicious models can be caught by looking at certain distance metrics between global and local models w_0 and w, respectively. Popular choices for distance metrics d are the Euclidean distance (L2 norm) between two arbitrary models w_i and w_j:

$$d_1(w_i, w_j) = \|w_i - w_j\| \tag{2.11}$$

or, alternatively, the cosine distance summed over every model layer:

$$d_2(w_i, w_j) = \sum_k^n 1 - \frac{w_i^k \cdot w_j^k}{\left\|w_i^k\right\| \left\|w_j^k\right\|} \tag{2.12}$$

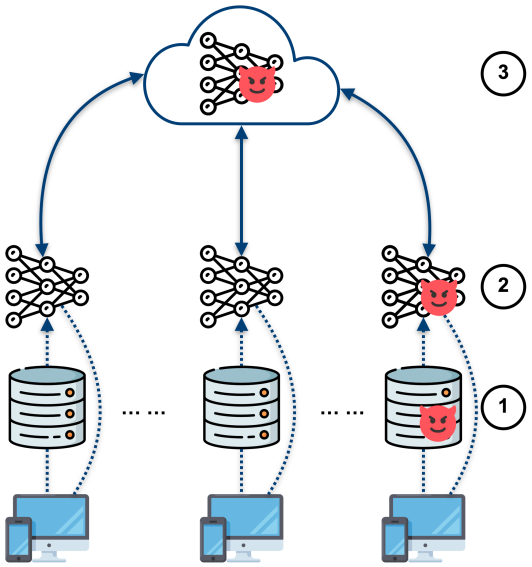


Figure 2.5: The figure shows how an adversary infiltrates an FL setup by taking control of a client to (1) poison its dataset s.t. its local model will be poisoned after the local training procedure, (2) with the ultimate goal of poisoning the global model after model aggregation (3).

With the goal of remaining stealthy, an adversary can attempt to train a malicious model while adapting to smaller distance values usually expected for benign models. Bagdasaryan et al. [10] achieve this goal by adding the distance to the global model as a secondary objective to the loss function, weighted with an additional parameter $\alpha \in (0, 1)$.

$$f_{\text{adapt}}(\mathbf{x}, \mathbf{y}, w) = \alpha \cdot f(\mathbf{x}, \mathbf{y}, w) + (1 - \alpha) \cdot d(w, w_0) \tag{2.13}$$

This approach, however, introduces a dilemma, as the adversary needs to decide between attaining an effective backdoor and limiting the distance to the global model. By tweaking the α parameter, the adversary can find a middle ground. Nonetheless, this search for the correct α value can turn out to be very time-consuming, and the adversary is far from guaranteed to find a value leading to satisfactory results. If the α value is chosen too small, the model will only learn to minimize the distance and potentially undo most of the training progress, decreasing the MA to the level of a naive classifier. This is in contrast to the goal of stealthiness. On the other hand, if the chosen α value is too large, the model will train as usual without adapting to the global model distance. As the main contribution, this thesis reformulates the objective (Eq. 2.13) into a constrained optimization problem to circumvent this dilemma. Sec. 4.1 gives a high-level intuition on how constrained optimization methods aid in this goal, and Sec. 6 discusses the actual optimization problem with more mathematical rigor.

2.6 Neural Tangent Kernels

Neural tangent kernels (NTKs) were first proposed in 2018 by Jacot et al. [18] following the observation that overparameterized models tend to change their parameters very slowly over time. Therefore, one can Taylor approximate [46] a model with starting parameters w_0 making a prediction on the sample x_i:

$$y(x_i, w_t) \approx y(x_i, w_0) + \nabla_w y(x_i, w_0)(w_t - w_0) + \dots \tag{2.14}$$

Since the approximation is linear if truncated after the first order, it can be interpreted as a linear transformation of coordinates x_i (in this specific case, the coordinates are our data). Given one-hot encoded labels $y_1, \ldots, y_n$ and the L2 norm as loss function, one now has a textbook example of a linear least squares problem [20].

$$\min_w \sum_{i=1}^{n} (y_i - y(x_i, w))^2 \tag{2.15}$$

Given the convexity property of the function, one can be sure that there is a unique minimum. There are many algorithms for finding the minimum for this type of problem (e.g., by calculating pseudo inverses [47]), but SGD-based algorithms provide satisfactory results as well. Since the model gradient needs to be stored for every sample x_i individually, computation of the NTK over a more extensive dataset requires a lot of memory. Therefore, further approximations can be made in the form of subsampling weights (dimensionality reduction), resulting in a so-called empirical NTK (eNTK) [48]. Other formulations of NTKs are possible, yet in FL settings, it was applied by Yu et al. [19], who showed that it could be advantageous to replace the original model with an NTK during the later stages of training.

The name neural tangent kernel originates from the fact that due to the Taylor approximation, the classifier is tangent to the original model and can be used as a kernel machine. It was initially envisioned as an approximation for a neural network of infinite size, yet it is its convexity property that is most useful for this project, as it has the potential to provide the basis for a defense against adversaries.

3. Related Work

Since its original proposal in 2017, a handful of publications have explicitly dealt with FL. However, as FL is part of the larger field of machine learning (especially deep learning), FL also benefits from the vast amount of research in those fields.

Work related to the adversarial part of this thesis can be divided into sections dealing with FL itself (Sec. 3.1), its aggregation algorithms, and motivations to choose more advanced methods for aggregation (Sec. 3.2), a section about backdoors and attacks that can occur in an FL setting (Sec. 3.3), and a section about defenses against adversaries (Sec. 3.4). Another section is devoted to the previous use of constrained optimization methods in deep learning (Sec. 3.5). For the defensive part of this thesis, a final section will deal with Neural Tangent Kernels and their use to FL (Sec. 3.6).

It should be noted that some publications include multiple ideas, e.g., for both new attacks and defenses (including different classes of defenses). Hence, some works are cited multiple times in different subsections.

3.1 Federated Learning

Algorithms for distributed learning have been around as early as the 1980s [49]. In the following years, many more have been proposed [50, 51, 52]. The novelty of FL, initially published by McMahan et al. in 2017 [1], is that it works with many clients (potentially millions) and highly non-iid data.

Federated Learning setups can be classified into two categories: *cross-device* and *cross-silo*. The former setting is the default scenario described by McMahan et al. [1] and is the default scenario for most FL-devoted research. The cross-device scenario assumes that the client data is distributed among an abundant client base composed of somewhat unreliable devices. In a real-world scenario, this would relate to a business-to-customer relationship where a large corporation hosts the server, and the devices and their data belong to individual end-users. Google's Gboard service [9] implements a FL in a cross-device scenario. The latter scenario considers business-to-business partnerships where the pool of clients comprises larger entities (e.g., healthcare and financial services providers), each of whom with large silos of data. Unlike in cross-device scenarios, clients can provide copious computational resources with high reliability. Despite being a primarily theoretical approach that has seen little real-world implementation yet, cross-silo scenarios have been studied in several publications [53, 54, 55, 56, 57]. Notably, Fereidooni et al. [58] developed a solution for mobile security providers to train risk detection models collaboratively.

3.2 Federated Aggregation

McMahan et al. [1] originally introduced *Federated Averaging (FedAvg)* as an update algorithm for the global model, and it is the default FL algorithm. The models are trained with SGD (stochastic gradient descent), and the global model is obtained by merely averaging over every client's SGD update. However, the performance of FedAvg can be subpar (as shown in [59]) when dealing with highly non-iid data. In such a scenario, a globally optimal model is not necessarily optimal for every client, leading local updates to divert the model from the previously optimal state. This reasoning stems from Karimireddy et al. [60], who also proposed SCAFFOLD as an alternative aggregation algorithm to overcome the limits of FedAvg. With SCAFFOLD, every client continues training their local model and uses the difference between its local and the global model for parameter optimization. It, therefore, also replaces SGD but remains very similar.

Another motivation for replacing FedAvg is the desire to stop adversaries. There have been a handful of aggregation algorithms designed to reduce the impact of adversaries or stop them entirely, like *Krum* [61] and Trimmed Mean and Median Aggregation [62]. Defending against adversaries is explored more thoroughly in Sec. 3.4.

Gradient updates can leak information about the training data [63], which is at odds with the promise of data privacy made by FL. Therefore, Bonawitz et al. [64] developed a *secure aggregation* protocol that uses data masking to protect client data privacy. This means the server can no longer tell what exact model updates individual clients are trying to push and only inspect the global model after aggregation. This impedes many defenses against poisoning attacks, as most of them rely on inspecting client models. Hence, most defenses assume that no *secure aggregation* is used.

3.3 Backdoor Attacks on Federated Learning

Backdoor attacks and defenses in deep learning (not restricted to FL) have been an active subject of research for much longer and are summarized in survey articles like [65, 66]. Unsurprisingly, researchers experimented with backdoors in FL only a short time after its proposal. In order to install a backdoor, the adversary must decide first on what type to implement. Once he has poisoned his data accordingly, the adversary needs to find a method to poison a global model while remaining stealthy. One promising approach was outlined by Bagdasaryan et al. [10], who were the first to install a backdoor in an FL setting successfully.

3.3.1 Backdoor Types

Bagdasaryan et al. [10] proposed the idea of so-called semantic backdoors. These are intentional misclassifications of unmanipulated data triggered by human-visible properties (e.g., "pictures of cars with a striped wall in the background shall be classified as birds"). Bagdasaryan et al. [10] and [67, 68] also experimented with pixel trigger backdoors that are activated with an adversary-crafted pixel pattern that is added to the input image. An example of both backdoor types can be seen in Fig. 3.1. Wang et al. [69] managed to install so-called *edge-case backdoors* using examples that are seemingly easy to classify but make the model struggle in deciding between two classes. Chen et al. [70] proposed a *Blend* backdoor by superimposing images with random noise as a trigger. *Clean Label* backdoor attacks [71, 72] modify training data with a trigger while retaining the original label. The backdoor then works by preparing an image of a different label with the same trigger, leading to a false classification by the model, as it associates the trigger with the original label during training. A relatively simple backdoor type that appears in many publications as a benchmark is a label-swap backdoor where all samples x_i of a given label, e.g., $y_i = 1$ are relabeled with $\tilde{y}_i = 2$.

(a) Benign Image **(b)** Pixel Trigger **(c)** Semantic Backdoor

Figure 3.1: The figure shows two backdoor-triggered images of cars from the CIFAR-10 dataset (low resolution due to data source) next to an unmodified image (a). Whereas in (b), the 2x2 patch of red pixels serves as a trigger, the blue and yellow background constitutes a trigger in (c).

3.3.2 Adversarial Training

Bagdasaryan et al. [10] initially achieved their goal of installing a backdoor into an FL model by substituting the global model through an adversary-chosen model. Furthermore, to circumvent simple defenses that sort out models deviating too far from benign models by regarding a specific metric, e.g., the distance to the global model (c.f. Sec. 3.4.1), the authors train adversarial models locally with a modified loss function (see Eq. 2.13 in Sec. 2.5). This modified loss function, however, comes with a new hyperparameter $\alpha \in (0, 1)$. By tweaking α, the adversary can focus on training the actual model or adapting the model to a certain distance. The latter approach, similar to a simple penalty method of constrained optimization, has since become the standard for backdoor training. The additional term added to the original loss function penalizes deviating models in the form of an increasing loss function. Hence, this penalty method merely incentivizes but does not strictly enforce smaller model distances. The incentive can be increased by tweaking the α parameter ($\alpha \to 1$), but it comes at the cost of the training objective moving out of sight. For this reason, Rieger et al. [73] speak of an *adversarial dilemma*, as the adversary has to decide between training an effective model with little control over the distance to the global model on the one hand or training a model close to the global model which on the other hand might perform poorly in terms of both main- and backdoor accuracy. An α balancing both extremes must be searched manually, which can be time-consuming, as it requires multiple attempts. As of yet, no circumvention of this dilemma is known.

More sophisticated attack attempts include, but are not limited to: Xie et al., who split the backdoor trigger into parts and distribute images poisoned with a partial trigger among different clients [74], and Nguyen et al. [75], who managed to backdoor FL models that detect malicious network traffic caused by IoT devices.

3.4 Defenses Against Backdoors in Federated Learning

The defenses against backdoors can be categorized into two classes. The first class comprises filtering approaches (Sec. 3.4.1). These aim to identify adversarial models so that they can be left out of the aggregation process. The second class comprises mitigation approaches (Sec. 3.4.2). Here, the goal is to render malicious models ineffective during the aggregation process, thereby keeping any backdoor from reaching the global model.

3.4.1 Filtering Approaches

The simplest approach by Bagdasaryan et al. [10] limits the distance that model updates are allowed from the global model. Their approach, however, was circumvented by their own constrain and scale attack, discussed in Sec. 3.3. The Auror defense [76] first tries to determine indicative model features and then applies K-Means clustering on updates for those features to detect outliers. DeepSight [73] clusters models by their pairwise distances and additional features stemming from a deeper model inspection. FoolsGold [77] takes the update histories of clients and the pairwise distances of client updates into account to assign weights to every client update when aggregating while only considering the model output layer. Zhao et al. [78] analyze models via a client-feedback loop. Some defenses, e.g., Flame [79], combine multiple defenses by first clustering client updates with the HDBSCAN algorithm [80] for outlier detection before aggregating the updates after clipping their size and adding noise. CrowdGuad [81] filters malicious models by analyzing models using private data in client-located secure enclaves running in Trusted Execution Environments.

3.4.2 Mitigation Approaches

Yin et al. [62] use their *Trimmed Mean* and median aggregation methods to limit a malicious model's influence. Both methods are similar to FedAvg. Trimmed Mean only aggregates updates closest to the mean of all updates. What updates to discard differs and is decided for every model parameter individually. Their median aggregation differs from FedAvg in that parameter updates are aggregated to their median instead of their mean. (Multi-)Krum [61] only selects a single (a few) model(s) for aggregation every round, namely the one(s) which had the shortest average distance to all other client models. Both approaches are unfit for non-iid scenarios, as for highly skewed data distributions, outliers for model updates are not necessarily malicious and still deserve to be considered. Zhao et al. [82] managed to repair backdoored models by taking advantage of the fact that by connecting two independently well-trained models via a parametric curve in parameter space, one receives a well-performing model at every point along the curve (*mode connectivity* [83]). As mode connectivity is no longer valid for backdoored models, taking a point along the curve allows for repairing such models. This approach, however, decreases the main accuracy of the entire model.

3.5 Constrained Federated Learning

In recent years, a handful of works have trained neural networks with loss functions subject to inequality constraints. Most of these publications use the ALM, as a step up from the classic Lagrangian method, which can only consider equality constraints. Its first implementation for distributed learning systems stems from Chatzipanagiotis et al. [84]. Sanagalli et al. [85] introduce a constraint to help a binary classifier train on a vastly imbalanced dataset. The only implementation of the ALM in a Federated Learning setting stems from Tang et al. [57], albeit not for the model training process itself, as the optimization problem considered by the authors dealt with creating incentives for clients in a cross-silo setting to contribute more computational resources. In a general deep learning setting, Rony et al. [86] use the ALM to replace the simple penalty method in adversarial example generation. As there are no known attacks where an adversary leverages the ALM to install backdoors, this will be the starting point for the adversarial part of this thesis.

3.6 Neural Tangent Kernels

A framework that has drawn much attention since its inception in 2018 is called *Neural Tangent Kernels (NTKs)* [18]. NTKs are first-order Taylor approximations of arbitrary neural networks (i.e., *tangent* to the original model, Sec. 2.6) that behave like kernels known from *kernel methods* in machine learning. While devised initially to approximate a single-layer fully connected neural network, they have since been implemented for more sophisticated architectures, like CNNs or even ResNet [19]. Fort et al. [17] found that while a complex neural network outperforms an NTK in the early stages of training, its "fate" is settled after the first few epochs, and the NTK of the model can then be used for faster convergence. Utilizing NTKs in the late stages of model training is akin to model fine-tuning [87]. Hence, Yu et al. [19] conclude that NTKs are incapable of learning new features yet prove helpful in applying the knowledge of already learned features. Additionally, Yu et al. [19] have successfully replicated these findings within a federated learning setting. The authors trained a simplified NTK, called empirical NTK (eNTK) [48], using the SCAFFOLD algorithm by Karimireddy et al. [60]. NTKs also prove useful for an attack that prevents the generalization of more complex models in a black-box environment [88]. While there have been first attempts at leveraging NTKs for backdoor attacks [89] on fully-sized neural networks, there are no known defenses based on the properties of NTKs, which is the focus of the defensive part of this thesis. Furthermore, this thesis considers a scenario where the global model is replaced by an NTK in the later stages of model training, unlike [89], which merely considers NTKs for their attack.

4. Approach

This thesis investigates both sides of an adversarial setup in a Federated Learning (FL) setting. In the first part (Sec. 4.1), the setup is regarded from an adversarial point of view, and the main goal is to install a backdoor into an FL system. So far, defenses based on anomaly detection functions have only been met with haphazardly engineered solutions that add the anomaly term to the loss function, incentivizing the backdoored model not to stray too far from the global model. Undertakings that regard these anomaly terms as hard constraints which must not be violated to remain undetected have yet to be made. This situation provides an obvious starting point for the thesis project. For the second part (Sec. 4.2), this thesis switches to the defensive side. As NTKs are an exciting development that has already been explicitly applied for model training in FL [19] and to construct attacks [88], albeit not backdoors, it would be interesting to see if NTKs (in particular their convex properties) can be used to detect backdoored models.

4.1 Adversarial Training with Constrained Optimization

The **main part** of the thesis will therefore consist of designing an attack that does indeed regard anomaly terms as hard constraints in an attempt at installing a backdoor. The method used by Bagdasaryan et al. [10] penalizes high deviations from the global model by adding additional terms to the loss function. This can lead to a scenario where constraints are vastly violated in the form of large model distances. Conversely, constrained optimization methods only yield minima that adhere to constraints. If successful, it would eliminate the dilemma discussed in related work (Sec. 3.3). As finding the correct value for some α parameter is no longer necessary, this would be a significant contribution to the field. The method of Lagrange multipliers [90, 91] is usually the first consideration for a constrained optimization problem. Its drawback, however, is that it only allows for adaptation to equality constraints. In our scenario, this would equate to a model trained to keep an exactly defined distance from the global model. The local optimization problem would then become:

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \quad \text{such that} \quad d(w, w_0) = \delta \tag{4.1}$$

Here, $d(w, w_0)$ defines the distance of the local model w to the global model w_0, and δ is the desired target. Obvious choices for distance functions are the Euclidean and the cosine distance as defined in the background section (Sec. 2.5, Eqs. 6.1 and 6.2). As there is a realistic probability that no model with satisfactory performance (i.e., prediction accuracy) can be found at that particular distance, the consideration of inequality constraints comes

to mind. Here, instead of constraining the model to a fixed distance, one can allow for smaller distances up to a maximum threshold, as many defenses expect malicious models to have larger distances than their benign counterparts. In this scenario, the optimization problem becomes:

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \quad \text{such that} \quad d(w, w_0) \leq \delta \tag{4.2}$$

with δ defining the maximum allowed distance. The drawback is that optimization methods with the ability to consider inequality constraints can become much more complicated than the well-known method of Lagrange multipliers. Nonetheless, there are optimization methods related to the Lagrangian method with said capability, one of them being the so-called augmented Lagrangian method (ALM) [16, 13, 14]. In the example with a single constraint, one can set up a Lagrange function in a way similar to the classic Lagrange method, which replaces the original loss function as training objective:

$$L(\mathbf{x}, \mathbf{y}, w, \lambda) = f(\mathbf{x}, \mathbf{y}, w) + \frac{1}{2\rho}\{\max^2(0, \lambda + \rho\,[d(w, w_0) - \delta]) - \lambda^2\} \tag{4.3}$$

The parameter λ is updated via SGD along with all the other model weights, and ρ is a parameter for which the ALM itself governs the update rules. Nonetheless, it cannot be assumed that adversarial training results in larger distances to the global model compared to benign models. Since defenses might also flag models with distances far smaller than most, an adversary can introduce further constraints to ensure a minimum distance. This would confine the distances of backdoored models to an interval. An adversary knowledgeable about the distances of all other benign models in the federation can center this interval around the mean of the benign model distances, making it hard for based defenses based on anomaly detection to flag the backdoored models. For a target-distance interval of $[\varepsilon, \delta]$ the problem from Eq. 4.2 then becomes

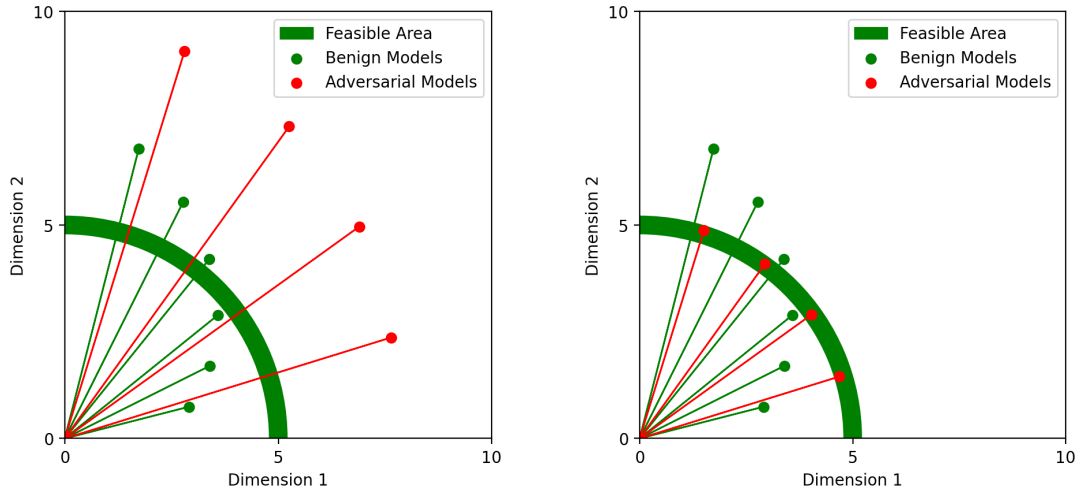
$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \quad \text{such that} \quad \varepsilon \leq d(w, w_0) \leq \delta \tag{4.4}$$

In another imaginable scenario, the adversary first goes about the ordinary training procedure without any malicious interference, only to measure the distance of the now benign model. After discarding the benign model, he restarts the training procedure with the poisoned dataset. Generally, an adversary could adapt to any distance metric, allowing an adversary to circumvent defenses based on different metrics. If an adversary has more profound insights into a defense and what values specific metrics need to have, he can formulate constraints to match those values. As defenses might measure model distances with multiple metrics at once (e.g., both Euclidean and cosine distance), the adversary can adapt to multiple metrics simultaneously by introducing additional constraints. A more rigorous explanation of how the ALM can be leveraged to solve a problem like Eq. 4.4 and problems with multiple constraints can be found in Sec. 6.1 and Sec. 6.2.

Figure 4.1 visualizes an example outcome targeted by the adversary. The figure shows a scenario where the adversary first measures the distances between the global model and all benign models before starting the adversarial training process and adapting its own model to a small band around the average of all benign model distances.

In addition to distance metrics, the adversary can also set other constraints. When keeping the original client dataset $(\mathbf{x}, \mathbf{y})$ separate from its set of poison samples and labels $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$, he can also set the classification loss on the poison dataset as a constraint:

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \quad \text{such that} \quad f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) \leq \delta \tag{4.5}$$



(a) Before adaption **(b)** After adaption

Figure 4.1: This mock-up figure sketches a 2D projection of local model weights in relation to the global model located at the origin. The adversarial models adapt to a small band around the mean distance of the benign models (feasible area). (a) shows the outgoing scenario before adaption, and (b) shows the ideal outcome for the adversary after adaption.

This way, only models with a classification loss of less than δ are accepted as a result of the optimization method. With the rate of poisoned samples in the client dataset leading to a trade-off between backdoor efficiency and main task performance (a high poisoning rate leads to an effective backdoor but a decrease in main task accuracy and vice versa), the adversary would now be able to produce models with high backdoor as well as main task accuracy while keeping only a small set of poisoned samples. Ideally, the set of poisoned images $\tilde{\mathbf{x}}$ would be less than 10% the size of the original client dataset $\mathbf{x}$.

Once implemented, the ALM can be seen as an extension of the SGD algorithm, and subsequent experiments can test its efficiency for various types of backdoors. One caveat is that the definition of success by a real-world adversary might differ significantly from the mathematical point of view. The attack is considered successful if the backdoor can be installed up to a high degree of accuracy (meaning the backdoor works for, e.g., 60% of trigger images) while remaining stealthy. This, however, does not necessarily translate to the model being a solution to the optimization problem in the mathematical sense, as it is far from guaranteed that the model constitutes a minimum of the Lagrangian.

4.2 NTK-based Defense

While its convexity properties make Neural Tangent Kernels (NTKs) desirable for ordinary model training, the fact that there is a unique minimum (Sec 2.6) can potentially help in constructing a defense as well. The **secondary part** of this project, therefore, consists of designing and implementing a defense against backdoors, with NTKs playing a crucial part. As discussed in the related work section (Sec. 3.6), switching to training NTKs instead of full neural networks (NNs) can be advantageous in the later stages of an FL setup. This is because a NN learns most important features early on during training, and its weights barely change during later stages. At this point, the global model is already close to convergence, and by training an NTK, one merely fine-tunes the model. In such a scenario, the server aggregates the client updates to a global NTK instead of a proper NN. Yu et al. [19] have already implemented an FL setup that leverages NTKs in later stages of the model training process and shown that this setup is plausible.

As NTKs are linear approximations of NNs, their training process can become a convex optimization problem if the associated loss function is also convex, like the L2 or categorical cross entropy (Eq. 2.5) loss functions. As with every other convex optimization problem, a unique minimum exists, in this case, in the form of a unique set of weights for the NTK. This makes training an NTK faster and computationally less expensive than a full NN. In a scenario where FL is mainly used for its decentralized training of models and data privacy is not a priority, the server can be granted reading permissions to the client datasets. Then, in an NTK scenario, by analyzing the properties of the client-trained NTKs, the server can potentially tell if the clients indeed used the same dataset the server has access to, or another, potentially poisoned dataset. An adversary might still attempt to train an NTK with a poisoned dataset but is unlikely to provide the server access to this dataset, as his malicious intent could be discovered with little difficulty. Instead, the adversary would show the server the original, unpoisoned client dataset. If the server can now determine that the client NTK was trained with a different dataset than it has access to, the client NTK can be flagged as potentially malicious.

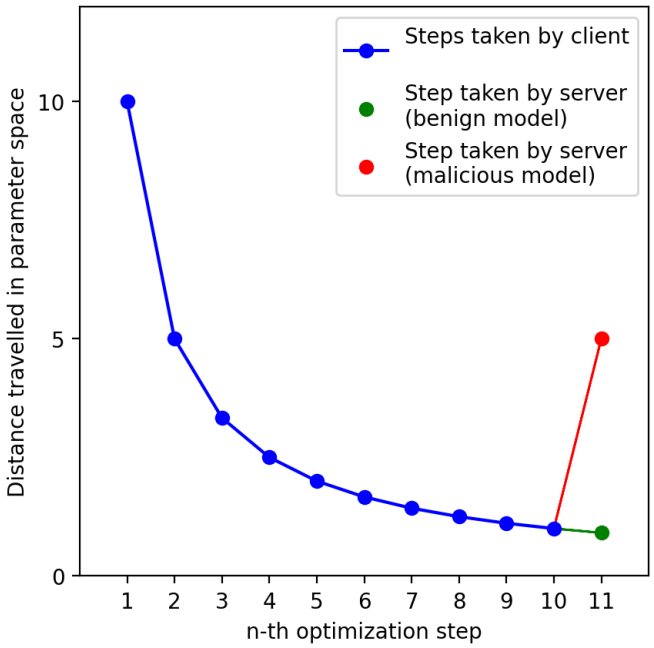


Figure 4.2: The figure shows the distances traveled in parameter space by a client training an NTK for 10 steps. When the server takes step number 11, there is a sharp increase if an adversary trained the NTK but a further decrease for benign models.

Since training the NTK poses a convex optimization problem, one would expect the clients to submit NTKs to the server whose weights are within close range of the unique minimum. This is a contrast to proper NNs where, depending on their size, a true minimum might never be found with limited computational resources. Sec. 8.2.4 shows a scenario where, for a proper NN, a minimum is indeed found, but only after ≈ 100 epochs of local training, which (a) leads to an overfitted model, and (b) requires longer periods of local training than is usual for an FL scenario. With the weights of the NTK in close vicinity of the minimum, taking further optimization steps would lead to barely measurable changes in the weights. A server with access to client datasets can confirm the client NTK training progress by calculating a single additional update step by itself. One key observation is that, like for proper NNs, the NTK optimization problem

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \tag{4.6}$$

depends on the dataset $(\mathbf{x}, \mathbf{y})$. In contrast, if an adversary has trained the NTK with a poisoned dataset $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ he will instead have solved

$$\min_{w} f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) \tag{4.7}$$

which is a different optimization problem than the server is assuming. If the server now takes its additional optimization step, there is a high probability that its size will no longer be diminishingly small. Such a scenario can be seen in Figure 4.2.

If a single additional step is taken for every client NTK update to the server, their distances can be compared. If outliers occur among the recorded step sizes, their associated NTKs can be flagged as potentially malicious. While large step sizes are no proof of a backdoor, they pose strong evidence that the client has trained his NTK with a different objective, e.g., by using a dataset unknown to the server. Since for conventional NNs, usually, no minimum is found during local training, and the training process is halted once the model reaches a satisfactory performance, this type of defense is expected to work only for NTKs.

5. System Setting and Threat Model

This chapter gives a high-level overview of the Federated Learning setting that constitutes the framework for all following experiments. It establishes the relationship between the server and the clients and defines how they are collaboratively training a machine learning model (Sec. 5.1). As this thesis heavily concentrates on adversarial training, a threat model (Sec. 5.2) will describe what vectors adversaries can use to fulfill their goal of disrupting the training process. In short, the system setup consists of a typical Federated Learning setup with a default number of 10 clients, collaboratively training a neural network for an image classification task. A short overview can be seen in Figure 5.1. The system setup and thread model are then valid for both parts (adversarial and defensive) of this thesis.

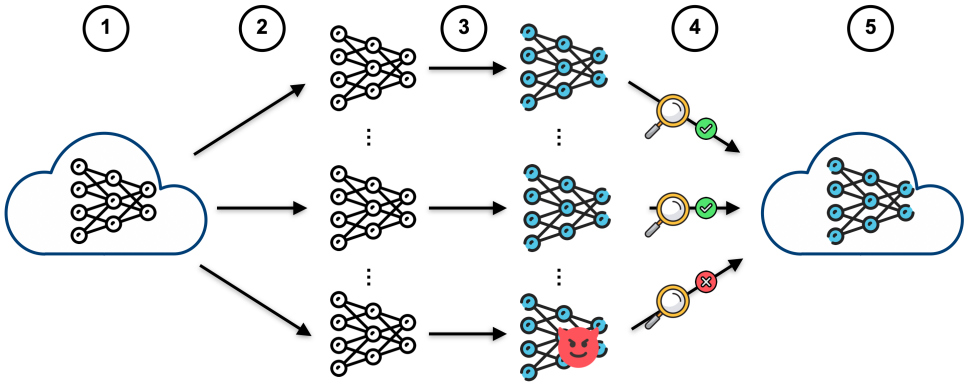


Figure 5.1: Visualization of the system setup. The steps include: (1) the server initializes the global model, (2) the global model is distributed to the clients, (3) every client trains their model on local data, but some clients poison the model, (4) potentially malicious models are filtered out (5) the remaining models are aggregated. For an effective defense the aggregated model remains unpoisoned.

5.1 Federated Learning Setup

A Federated Learning setup, as described in the background section (Sec. 2.3), will be at the base of this project. The goal is to collaboratively train a machine learning model that, once it has been adequately trained, is capable of assigning a label from a predetermined list (e.g., for the CIFAR-10 dataset [21] these would be: airplane, automobile, bird, ...)

to images which it has never seen before during the training process. It is, therefore, a standard image classification task in the realm of supervised learning. The main accuracy (MA) is later evaluated with a separate testing dataset.

The setup consists of a server S that orchestrates the training process of K clients on separate devices. The server determines the model architecture (like ResNet [26], or AlexNet [92]) and, therefore, the mathematical function that is to be optimized, the aggregation algorithm (like FedAvg [1] or SCAFFOLD [60]), and all global and local training hyperparameters. Every client in the federation has its own dataset that is used for the local training process and is not shared with any other clients. Every client then undertakes the local training procedure on their platform.

5.1.1 Hyperparameters

The server sets the local training and aggregation hyperparameters. While the clients can change the local training parameters, they are not expected to do so, as a defense might detect them as potentially malicious.

Global Hyperparameters

The hyperparameters affecting the federated learning system itself, as well as the aggregation process, are:

- K, the number of clients admitted to the federation. For this setting, the default value for this parameter is 10.
- C, the fraction of clients tasked with training the model and submitting their update to the server for a given round of aggregation. The default value is 1, meaning all clients participate in local training for every round
- R, the number of total rounds for local training and aggregation.
- w_0, the weights of the initial global model. The server can either start with an already pre-trained model before initialing the federated learning process. This model could potentially have been trained on completely different data that none of the federation clients can access. Otherwise, the model is initialized with random weights.

Local Training Hyperparameters

These are hyperparameters that affect the local training of the client models. This list is, in theory, identical to the list of hyperparameters one has when training a deep learning model with a single client.

- η, the learning rate for the SGD update algorithm
- *momentum* and *weight decay*
- b, the size of the training batches for the SGD algorithm
- e, the number of local training epochs before submitting the model to the server.

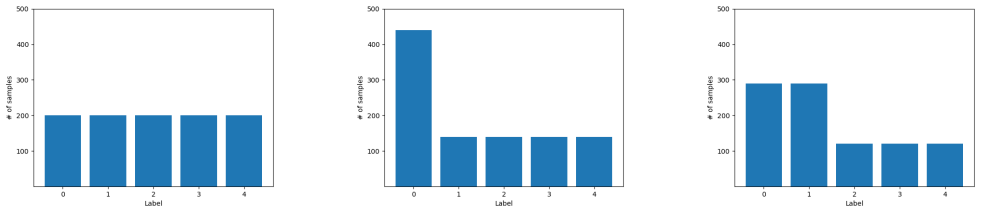
5.1.2 Client Datasets

The datasets held by the clients contain the images used for training the model, along with their respective labels. The images must be compatible with the model prediction function, meaning the number of color channels and the dimensions (resolution in pixels) are fixed. Furthermore, the labels must be within a set predetermined by the server (car, airplane, cat, dog, ...). In general, samples are heavily non-iid with respect to their labels.

This means that it is possible for some clients to have most of their dataset consist of pictures of airplanes while having only a handful of pictures depicting dogs or even none at all. For sampling of client data, three types of distributions are considered:

- **iid**
  The data is iid sampled.
- **one-class non-iid**
  A main-label is assiged to a client. A certain percentage of the client data is then of the main label, while the rest is iid sampled.
- **one-class non-iid**
  Similar to one-class non-iid, except that a certain percentage of samples is split evenly among two main labels.

The distributions are also visualized in Fig. 5.2. Though the samples are label-wise non-iid among the clients, every client has the same number of samples. For all experiments evaluated in Sec. 8, every client had 40 batches with 64 samples each (2560 in total). However, a scenario where all training samples are non-iid, even across clients, is also possible.



(a) iid data **(b)** one class non-iid data **(c)** two class non-iid data

Figure 5.2: Example of iid., one-class, and two-class non-iid data: Shown is the data held by an individual client sorted by label. (1000 samples, 5 different labels) In (a), the client data is iid. In (b), 70% of the data is iid while the remaining 30% of samples are of a main label, whereas in (c), the remaining 30% of samples are split 50/50 among two main labels.

The adversarial part of this thesis considers both cross-device and cross-silo scenarios where the server does not have access to any client data as well. As the NTK-based defense proposed in Sec. 4.2 only works under the assumption that the server has access to client data, this restriction is lifted only for the purpose of implementing and evaluating this defense. This also limits the setting to a cross-silo scenario, where a single entity owns all devices used for model training and Federated Learning is merely utilized for its distributed computation, and data privacy is not strictly necessary.

5.2 Threat Model

As this thesis is supposed to investigate Federated Learning in an adversarial setting, one has to introduce a threat model. The purpose of this threat model is to establish what influence an adversary has over the training process, what his goals are, and how he plans to achieve them. This threat model is valid for both parts of this thesis. The adversary has gained control of up to half of the client devices (50% - 1) and now has access to all local training parameters, client labels, and datasets. The goal of the adversary is to install a backdoor during local training with the intention that it persists after the global model aggregation by the server. On the other hand, the server is aware of the threat of adversaries and deploys defense mechanisms to detect potentially malicious client updates.

The assumption is that one adversary gains root-level access to the devices of $m < \frac{n}{2}$ clients, meaning that he has complete control over up to half of all the clients in the federation. Setting this limit is reasonable, as most defenses assume that benign models lie close to the mean of many metrics. Once the adversary has access to a client device, he can change every local training hyperparameter and the client dataset. While the server usually defines the local training hyperparameters, there is no way for the server to know if the client actually adhered to these parameters or if, indeed, changes were made. It is even possible for the adversary to change the update mechanism from SGD to something else entirely. As the adversary merely has to submit the model weights for aggregation at the end of a local training round, he can train the model in any way he wants to, irrespective of the Federated Learning setting, as long as he does not change the architecture of the model itself. In particular, he can also train the model on the original unpoisoned client dataset, and collect metrics about the model to gain insights about the behavior of a benign model, only to discard it afterward and start over from the original model with his adversarial procedure. As for the client datasets, the adversary can make arbitrary changes to his data as well. The training labels can be changed to no longer reflect the truth, but also the images themselves can be modified. The adversary can modify individual pixels, superimpose the images with noise, or swap out images (with new labels) altogether.

In this setting, the goal of the adversary is to install a backdoor into the global model. As described in the background section (Sec. 2.4), this is a targeted misclassification of a selected subset of images. This means that for some images from a poison set $\{x_0, \ldots, x_n\}$ the global model should no longer classify these images with their correct labels $\{y_0, \ldots, y_n\}$, but instead with false, adversary chosen labels $\{\tilde{y}_0, \ldots, \tilde{y}_n\}$. Furthermore, the backdoor should remain undetected by the server, and the accuracy on non-poison images should remain high. The adversary's goals can be summarized in the following bullet points:

- **High Backdoor Accuracy**
  By conducting local training with a dataset that contains images from the poison set, the adversary wants the model to predict the a priori chosen false label with high accuracy. This high accuracy on poisoned samples should carry over to the global dataset after aggregation and persist for multiple rounds.

- **High Main Task Accuracy**
  The process of installing the backdoor should not interfere with the goal of training a model with a high prediction accuracy on unpoisoned samples (stealthiness).

- **Evading Detection**
  As the backdoor should persist over multiple rounds, the adversary does not want to be caught by a defense during the aggregation process.

In order to meet these goals, the adversary usually continues training on a mostly unaltered client dataset and only poisons a portion (e.g., 10%) of the dataset, as with a poisoning rate of 100%, the model would lose most of its accuracy on benign samples and the main task accuracy might even fall to the level of a naive classifier. As the backdoor would otherwise even be detectable, the adversary is incentivized to only use poison samples that are unlikely to occur in other clients' datasets or the evaluation dataset. For example, for a model that is supposed to recognize traffic signs, the adversary will most likely not mislabel all stop signs, as such a backdoor would be easy to find, but only mislabel stop signs with, e.g., a sticker attached to them. By training a model with a poisoned dataset, the adversary aims to produce a poisoned model with a built-in backdoor (*model-poisoning* through *data-poisoning*).

As many defenses evaluate the distances (measured by some metric like L2 norm or cosine distance) of all client models to the global model, the adversary also tries to adapt these

metrics to values similar to those of benign models. Therefore, the adversary changes the local training procedure by introducing constraints that allow him to only produce models with a limited distance to the global model, which he then submits for aggregation. In order to meet these constraints, the adversary can resort to constrained optimization methods like the penalty method or the (augmented) Lagrangian method. When setting up the constraints, the adversary must define the distances to which he wishes to adapt his models. As stated earlier, the adversary can train the model with a benign dataset, measure the metrics to the global model and subsequently adapt to the exact distance that a benign model produced by the same client would have. For some experiments, the adversary will also be granted access to the distance metrics of the benign client models. This way, the adversary can adapt his model to the mean of the distances of all benign clients.

5.3 Defenses

As the server does not want adversaries to tamper with the model, it can employ defense strategies to prevent adversaries from attacking the global model. A defense works by investigating client updates submitted for aggregation and flagging potentially malicious updates. These supposed malicious updates can then be filtered before the aggregation process and have, therefore, no impact on the next global model. As the model training generally takes place in a non-iid scenario, the main difficulty resides in telling genuinely malicious models apart from models trained on unusual yet otherwise benign datasets. Similar to the adversary's goal, the defense should not interfere with the model training process, and its goals are summarized in the following:

- **Retain Main Task Accuracy**
  Deploying the defense should not lead to a decreasing main task accuracy, similar to the adversary's goal. Ideally, in cases where no adversaries were found, the aggregated model should be identical to the aggregated model one would have if there had been no defense in place.
- **Mitigating Backdoors**
  The defense should keep backdoors previously trained by adversaries into local models from reaching the global model. This can be achieved by either assigning potentially malicious updates smaller weights for aggregation or completely discarding them (See related work discussion: detection and filtering vs. mitigation defenses, Sec. 3.4). This project focuses primarily on filtering defenses.
- **Unsupervised**
  The defense should require no further human interaction after determining an initial set of defense-specific hyperparameters. These parameters can be set before the initial round and should remain unchanged for further aggregation rounds.
- **Limited Computation**
  As the distribution of computational load for model training is one of the main advantages of Federated Learning, the amount of computation required for the defense should also be limited. A defense that requires almost as much computation as the model training itself would contradict the purpose of FL.

All filtering defenses discussed in the related work section (Sec. 3.4) investigate client updates after they were submitted to the server and before aggregation. While it is possible to design defenses for other types of attacks, the defenses considered for this setting should catch model updates containing backdoors as described in the threat model. With the adversary trying to adapt to the global model measured by distance metrics (L2 norm and cosine distance), the defenses used for this setting should also primarily take distance metrics into account.

6. The Optimization Problem

This chapter will set up the constrained optimization problem that is to be solved by an adaptive adversary as described in Sec. 5.2 more thoroughly, as well as give a short derivation of the Augmented Lagrangian Method (ALM) that plays a central role in solving said optimization problem. Furthermore, for the defensive side of this project, this chapter will introduce a version of the Neural Tangent Kernel, the empirical Neural Tangent Kernels (eNTK), which is more suitable for use in real-world scenarios with limited computational resources, along with the SCAFFOLD [60] aggregation algorithm for Federated Learning (FL). The optimization problem and eNTK described will later be implemented as a computer program making this chapter the theoretical precursor to the immediately following implementation chapter.

6.1 The Objective Function

The background section already covers the global optimization problem governing FL and the local optimization problem solved by individual clients. As a reminder, the optimization problem for individual clients is usually unconstrained where clients C_i with labeled datasets $(\mathbf{x}, \mathbf{y})$ simply search for an optimal set of weights w^* minimizing the classification loss $f(\mathbf{x}, \mathbf{y}, w)$ of the neural network which serves as the objective function. The optimization problem is simply $\min_w f(\mathbf{x}, \mathbf{y}, w)$, which can turn out to be impossible already depending on the complexity of the network.

A network with many layers and millions of weights can suffer from vanishing and/or exploding gradients [93] where the partial derivatives of f with respect to individual weights can become either immeasurably tiny or assume orders of magnitude, making the network behave chaotic to small updates of single weights. To combat this behavior, one usually introduces multiple batch normalization layers [35, 36] that transform layer outputs to mean-zero and unit-variance vectors when constructing a neural network. Moreover, input samples $\mathbf{x}$ are usually transformed into unit sizes as well. The ultimate goal is to keep both model inputs and weights within a few orders of magnitude, usually in the interval of about $[-1, 1]$, although these should not be understood as constraints. In short, when designing a network architecture, one tries to keep the objective function as well-behaved as possible despite its complexity. In the following, f denotes the loss function of an arbitrary neural network.

As described in the approach (Sec. 4.1), an adversary with poisoned data $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ wants to constrain the distances of his model weights w to the global model w_0. As a reminder, the distance metrics d_1, d_2 are the Euclidean distance between two arbitrary models w_i and w_j:

$$d_1(w_i, w_j) = \|w_i - w_j\| \tag{6.1}$$

And the cosine distance summed over every model layer:

$$d_2(w_i, w_j) = \sum_{k}^{n} 1 - \frac{w_i^k \cdot w_j^k}{\left\|w_i^k\right\| \left\|w_j^k\right\|} \tag{6.2}$$

Ideally, the adversary wants to constrain these two metrics to intervals $[\varepsilon_1, \delta_1]$ and $[\varepsilon_2, \delta_2]$ for d_1 and d_2 respectively. The optimization problem for the adversary then becomes:

$$\min_{w} f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) \quad \text{such that} \quad \varepsilon_1 \leq d_1(w, w_0) \leq \delta_1 \ \text{ and } \ \varepsilon_2 \leq d_2(w, w_0) \leq \delta_2 \tag{6.3}$$

In the field of constrained optimization, however, one considers constraints of the standardized form $g(x) \leq 0$. Therefore, the constraints in Eq. 6.3 must be rewritten. This can be done by splitting the constraints into four and rearranging

$$g_1(w) \coloneqq \varepsilon_1 - d_1(w, w_0) \ \text{ and } \ g_2(w) \coloneqq d_1(w, w_0) - \delta_1 \tag{6.4}$$

as well as

$$g_3(w) \coloneqq \varepsilon_2 - d_2(w, w_0) \ \text{ and } \ g_4(w) \coloneqq d_2(w, w_0) - \delta_2 \tag{6.5}$$

The optimization problem now simplifies to

$$\min_{w} f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) \quad \text{such that} \quad g_i(w) \leq 0, \ \ i \in \{1, 2, 3, 4\} \tag{6.6}$$

If the adversary also wants to constrain the classification loss on the poison data and only keep the loss on the benign data unconstrained, he can introduce a fifth constraint

$$g_5(w) \coloneqq f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) - \gamma \leq 0 \tag{6.7}$$

and now set up the problem

$$\min_{w} f(\mathbf{x}, \mathbf{y}, w) \quad \text{such that} \quad f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) - \gamma \leq 0 \ \text{ and } \ g_i(w) \leq 0, \ \ i \in \{1, 2, 3, 4\} \tag{6.8}$$

Here γ defines the maximum permissible loss on the poison dataset. This problem formulation essentially specifies the training on the backdoor dataset along with the four remaining constraints as the number one priority.

6.2 From Classic to Augmented Lagrangian

The first choice for solving constrained optimization problems is usually the method of Lagrange multipliers. The drawback is that this classic method can only take on equality constraints of the form $h_i(w) = 0$, $i = 1, \ldots, k$. Therefore, one can switch to the ALM, which also facilitates optimization with respect to inequality constraints, as introduced in the previous section. This method was first introduced by Hestenes [14], and Powell [94] and was later extended by Rockarfellar [15] to work with inequality constraints. The following short derivation stems from an unpublished survey by Kanzow [13] and is now summarized to fit the adversarial FL setup.

The ALM is a combination of the classic Lagrangian method and the penalty method. For equality constraints $h(w) = (h_1(w), \ldots, h_k(w)) = 0$ the Lagrangian is defined as

$$L_\rho(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w, \lambda) = f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) + \frac{\rho}{2}\|h(w)\|^2 + \lambda^T h(w) \tag{6.9}$$

$$= f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) + \sum_{i=1}^{k} \Big(\frac{\rho}{2}(h_i(w))^2 + \lambda_i \cdot h_i(w)\Big) \tag{6.10}$$

The vector $\lambda = (\lambda_1, \ldots, \lambda_k)$ contains the Lagrange multipliers to be optimized alongside the model weights w. The squared norm of the constraint functions $\|h(w)\|^2$ constitutes the penalty term, and the ρ defines a penalty parameter which is updated in steps as well.

Any inequality constraints $g(w) = (g_1(w), \ldots, g_k(w)) \leq 0$ that should be adhered to can now be rewritten as equality constraints by introducing additional parameters that need to be optimized, similar to λ. If an inequality constraint $g_i(w) \leq 0$ is satisfied, it assumes the value of

$$g_i(w) = -s_i^2 \iff g_i(w) + s_i^2 = 0 \tag{6.11}$$

for some $s_i \in \mathbb{R}$. These s_i are now the newly introduced parameters, but not for long, as their minimization can be carried out beforehand. The Lagrangian, which has now become

$$L_\rho(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w, \lambda, s) = f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) + \sum_{i=1}^{k} \Big(\frac{\rho}{2}(g_i(w) + s_i^2)^2 + \lambda_i \cdot (g_i(w) + s_i^2)\Big) \tag{6.12}$$

with $s = (s_1, \ldots, s_k)$ for a problem with solely inequality constraints can now be minimized with respect to s by simply solving

$$\min_{s_i^2} \ \frac{\rho}{2}(g_i(w) + s_i^2)^2 + \lambda_i \cdot (g_i(w) + s_i^2) \tag{6.13}$$

for individual s_i^2, for which

$$(s_i^*)^2 = \max\left(-\frac{\lambda_i}{\rho} - g_i(w), 0\right) \tag{6.14}$$

is a solution. After inserting the its solution into the Lagrangian, $L_\rho(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w, \lambda)$ can now be simplified to[1]

$$L_\rho(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w, \lambda) = f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w, \lambda) + \frac{1}{2\rho}\sum_{i=1}^{k} \big(\max^2(0, \lambda_i + \rho g_i(w)) - \lambda_i^2\big) \tag{6.15}$$

One can now attempt to minimize the Lagrangian by running minimization algorithms like SGD. Normally, for the ALM, λ is updated via the Hestenes-Powell method:

$$\lambda_i \leftarrow \max(0, \lambda_i + \rho g_i(w)) \tag{6.16}$$

However, while conducting experiments, the following update rule has led to better results:

$$\lambda_i \leftarrow \lambda_i + \rho g_i(w) \ \text{ if } \ g_i(w) \geq 0 \quad \text{and} \quad \lambda_i \leftarrow 0 \ \text{ else} \tag{6.17}$$

After coming up with an update rule for the penalty parameter ρ, the ALM can be summarized as Algorithm 1. All parameters except the initial weights w^0 initialized in (S1) are hyperparameters. Unlike the α from Sec. 2.5, they, however, do not lead to an adversarial dilemma and, as discussed in the evaluation (Sec. 8), a constant set of values leading to satisfactory results for most experiments can be found.

[1] This step includes lengthy algebraic manipulations.

Algorithm 1: ***(Augmented Lagrangian Method for Inequality Constraints)***

(S1) Initialize (w^0, λ^0), $\tau \in (0,1)$, $\sigma > 1$ and set $k = 0$

(S2) If (w^k, λ^k) is a critical point of L_ρ: Stop

(S3) Perform unconstrained minimization (e.g., via SGD): $\min_w L_\rho(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w^k, \lambda^k)$

(S4) Update λ according to Eq. 6.17

(S5) If $\left\|\min\{-g(w^{k+1}), \lambda^{k+1}\}\right\| \leq \tau \left\|\min\{-g(w^k), \lambda^k\}\right\|$, then $\rho_{k+1} = \rho_k$, else $\rho_{k+1} = \gamma \rho_k$

(S6) Update $k \leftarrow k + 1$, and return to (S2)

The so-called KKT conditions [95, 96] define sufficient conditions [97] for a solution (w^*, λ^*) to problems Eqs. 6.6 and 6.8 and are given by:

- **Stationary Condition**

$$\nabla f(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, w) + \sum_{i=1}^{k} \lambda_i \nabla g_i(w) = 0 \tag{6.18}$$

- **Primal Feasibility**

$$g_i(w) \leq 0, \;\; i = 1, \ldots, k \tag{6.19}$$

- **Dual Feasibility**

$$\lambda_i \geq 0, \;\; i = 1, \ldots, k \tag{6.20}$$

- **Complementary Slackness**

$$\lambda_i \cdot g_i(w) = 0, \;\; i = 1, \ldots, k \tag{6.21}$$

This means that any point (w^*, λ^*) satisfying these conditions is also a solution to the optimization problem. In particular, the complementary slackness condition justifies setting λ_i to zero once $g_i(w) < 0$, as Eq. 6.21 stipulates that λ_i must be zero in that case.

6.3 The Empirical Neural Tangent Kernel (eNTK)

The Neural Tangent Kernel, a linear approximation of a neural network, as described in the background section (Sec. 2.6), can lead to prohibitively expensive memory requirements, as the gradient of the model prediction function $\nabla_w y(x_i, w)$ not only needs to be calculated for every sample x_i but also stored in memory. This is in contrast to a full-sized neural network where the $\nabla_w y(x_i, w)$ also need to be calculated at first, yet they are merely needed for the gradient calculation of the prediction loss $\nabla_w f(\mathbf{x}, \mathbf{y}, w)$ over an entire batch of samples. This allows for more efficient memory usage, as one can calculate the partial derivatives $\partial_{w_j} f(\mathbf{x}, \mathbf{y}, w)$ sequentially, only keeping the final result stored in memory and discard the individual contributions of $\partial_{w_j} y(x_i, w)$ (see "autograd", Sec. 7.1). Therefore, full-sized NTKs for large neural networks become infeasible over a larger dataset. Yu et al. [19] have therefore made further approximations and implemented an *empirical* Neural Tangent Kernel (eNTK) in an FL setting that overcomes these obstacles.

6.3.1 Setting up the eNTK

As a reminder, the NTK approximation of the global model prediction function with weights w_0 of a given sample x_i is the first-order Taylor approximation:

$$y(x_i, w) \approx y(x_i, w_0) + \nabla_w y(x_i, w_0)^T \cdot (w - w_0) \tag{6.22}$$

If the input sample has q dimensions and is labeled with a one-hot encoded vector of dimension c (number of classes), then $y\colon \mathbb{R}^q \to \mathbb{R}^c$. If furthermore the vector of weights has dimension P, then $\nabla_w y(x_i, w_0)^T \in \mathbb{R}^{c \times P}$. The problem now lies in finding an optimal set of weights $\tilde{w} = w - w_0 \in \mathbb{R}^P$ for the linear classifier.

The first approximation now consists of randomly re-initializing the last layer of the model (i.e., the last elements of w_0). As the expected prediction is now zero for every class ($\mathbb{E}[y(x_i, w_0)] = 0$), even for a formerly pre-trained model, all but the first output dimensions can be dropped, and only $y_1(x_i, w_0)$ is considered. As now $\nabla_w y_1(x_i, w_0)^T \in \mathbb{R}^{1 \times P}$, the dimensionality of the weights vector needs to be extended ($\tilde{w} \in \mathbb{R}^{P \times c}$) in order for there to be still c output dimensions. Because of the zero expected value, $y(x_i, w_0) \approx 0$, and the approximation simplifies to:

$$y(x_i, w) \approx \nabla_w y_1(x_i, w_0)^T \cdot \tilde{w} \tag{6.23}$$

Finding the optimal $\tilde{w}$ is now a simple linear regression task. Another approximation consists of subsampling the parameter weights from P to p dimensions. In addition to expediting computation, this has the added benefit of reducing the necessary bandwidth for communication between the server and the clients.

The authors summarize their derivation in three steps:

1. The last layer of w_0 is re-initialized with random weights
2. Compute the input transformation $\phi\colon \mathbb{R}^q \to \mathbb{R}^P$, $x_i \mapsto \nabla_w y_1(x_i, w_0)$ for every sample x_i
3. Downsample $\nabla_w y_1(x_i, w_0)$ for every x_i ($\mathbb{R}^P \to \mathbb{R}^p$)

Once an optimal $\tilde{w}^*$ has been found, a prediction on a sample x can now simply be made by

$$\phi(x)^T \tilde{w}^* = \nabla_w y_1(x, w_0)^T \tilde{w}^* \tag{6.24}$$

One should note that in order to make a prediction on a new sample (forward pass), one first needs to calculate the gradient of the model at the initial stage of NTK training (backward pass). With the approximations made by Yu et al., however, this should not be a concern in terms of computational complexity.

6.3.2 eNTK Optimization

When working with NTKs, the L2 norm is usually chosen as the loss function. Hence, for a dataset of samples $\mathbf{x} = (x_1, \ldots, x_n)$ with one-hot encoded labels $\mathbf{y} = (y_1, \ldots, y_n)$

$$f(\mathbf{x}, \mathbf{y}, w) = \frac{1}{2} \sum_{i=0}^{n} \left\| \phi(x_i)^T w - y_i \right\|^2 \tag{6.25}$$

After computing the gradient

$$\nabla_w f(\mathbf{x}, \mathbf{y}, w) = \sum_{i=0}^{n} \phi(x_i) \cdot \left(\phi(x_i)^T w - y_i \right) \tag{6.26}$$

one can use the SGD-based FedAvg algorithm for federated optimization of w. Nonetheless, as the authors note that FedAvg shows subpar performance for convex optimization problems like the above, especially in setups with non-iid data, they train the eNTK model with an implementation of SCAFFOLD [60] as opposed to FedAvg. SCAFFOLD is mostly based on FedAvg with the introduction of *update correction* as the sole difference.

Consider a client that has just received the newly aggregated global model w_0. In FedAvg, the client would now perform gradient updates with a learning rate η for every batch over M epochs. During the NTK training stage, the client now continues training with merely a single batch of data. The primary reasons are memory usage and that NTKs are merely used for model fine-tuning at the end of the training procedure. Initialized with $c_0 = 0$, the update correction is now defined as

$$c_{t+1} = c_t + \frac{1}{M\eta} (w_{-1} - w_0) \tag{6.27}$$

Here w_{-1} are the weights the client has previously submitted to the server. Therefore, the update correction tracks differences between the local and global model updates. The client update w_M that is later sent to the server is now iteratively computed over M steps with

$$w_{t+1} = w_t - \eta (\nabla_w f(\mathbf{x}, \mathbf{y}, w_t) - c_t) \tag{6.28}$$

starting from w_0.

7. Implementation

In order to evaluate the theoretical deliberations from earlier chapters, one will have to conduct experiments. The standard approach for this scenario is to write a computer program simulating the behavior of multiple clients training machine learning models. Since Federated Learning has become a popular research subject, a lot of code this work can build upon has become readily available.

After giving a brief introduction on how to train neural networks (NNs) with the PyTorch framework in general, this chapter explains how to extend the Federated Learning setup created and used by Torsten Krauß for his research in [81] to allow for adaptive adversaries using the Augmented Lagrangian Method (ALM) as described in Sec. 4.1 and 6.2. Moreover, Yu et al. [19] have open-sourced the code for their implementation of an eNTK in a Federated Learning setting which provides a great starting point for the defensive approach of this thesis.

7.1 PyTorch

As described in Sec. 2.2.2, NNs can assume complex architectures, making it difficult to keep track of gradients. Especially the calculation of the loss function gradient with respect to weights from earlier layers can become tedious when done manually. Furthermore, solving the optimization problem of model training can become prohibitively expensive. Fortunately, in recent years frameworks like PyTorch [45] and TensorFlow [98] have become available to the public. These frameworks provide an interface for the Python programming language allowing the design of NNs using pre-implemented building blocks and simplifying the training process by offering implementations of common (unconstrained) optimization methods. Moreover, these frameworks enable hardware accelerated computing with specialized hardware, e.g., via the NVIDIA CUDA [99] interface. With most of the available code relevant to this project using PyTorch as the main framework, this project will, too, leverage the PyTorch framework.

At the core of PyTorch lies *Autograd*, an automatic differentiation engine keeping track of all arithmetic operations involving `torch.Tensor` objects in the shape of a computational graph. In the context of PyTorch, a tensor represents any numerical object of arbitrary dimension from single numbers $x \in \mathbb{R}$ over vectors $v \in \mathbb{R}^n$ and matrices $A \in \mathbb{R}^{n \times m}$ to higher dimensional objects. A dataset comprised of n different 32×32 pixel RGB images is also represented as tensor of shape $n \times 3 \times 32 \times 32$. For gradient computation, one

can simply call the `.backward()` method on a tensor object to calculate all its partial derivatives. This is illustrated by the following code snippet:

After defining $z(x, y) = x^2 + 2y$ with $x = 5$ and $y = 3$

```
x = torch.tensor([5.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
z = x*x + 2*y #equal to 31
```

the gradient $\nabla z(x, y)$ can be calculated with

```
z.backward()
```

and the individual partial derivatives can be accessed with

```
print(x.grad) #prints out 10
print(y.grad) #prints out 2
```

Later, the `.backward()` method is called for the loss function of an NN to calculate its gradient. A simple NN can now be defined as a class:

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(3, 10, 5) # 3 RGB input channels
        self.pool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(1960, 10) # 10 output classes

    def forward(self, x):
        x = self.pool(F.relu(self.conv(x)))
        x = torch.flatten(x, 1) # reshape 2D images to 1D tensors
        x = F.softmax(self.fc(x))
        return x

net = Net()
```

The `def __init__(self):` function called upon instantiating a new instance of the network with `net = Net()` initializes the network layers with random parameters. In this case, `nn.Conv2d(3, 10, 5)` defines a 2D convolutional layer with 3 input channels and 10 square-shaped kernels of size 5. Given an 3x32x32 shaped tensor as input, the layer outputs a 10x28x28 shaped tensor. The pooling layer defined by `nn.MaxPool2d(2, 2)` does not have any trainable weights but discards all but the maximum value over a 2x2 grid, effectively cutting the 10x28x28 values down to 10x14x14=1960. The third layer `nn.Linear(1960, 10)` defines an affine transformation from 1960 to 10 dimensions. The function `def forward(self, x):` defines how the previously defined layers are combined to form an actual function taking a batch of images in tensor shape as input to output classifications with the addition of activation functions like ReLU and Softmax.

If one now wants to train the network via SGD, one can first define an optimizer object and pass the network parameters and the desired learning rate as parameter:

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

After creating batches of images and labels in tensor shape and opting for a loss function (e.g. categorical cross entropy), one can take optimization steps by:

```
loss_function = nn.CrossEntropyLoss()
for batch, label in data:
    optimizer.zero_grad() # resets previously calculated partial derivatives to 0
    predictions = net(batch)
    loss = loss_function(predictions, labels)
    optimizer.step() # optimizes model parameters
```

The example given above is a very simple example of an NN made up of merely two trainable layers. More complex architectures like ResNet18 [26] span hundreds of lines of code and are, therefore, not discussed in more detail at this point.

7.2 FL Framework

To simulate a Federated Learning setup as described in the background section, one can create a Python application handling the local model training sequentially on a single device before aggregating the individual models. Ideally, the application would automatically run multiple experiments with varying parameters, perform analyses and render visualizations of results. Fortunately, such a comprehensive application has already been developed by Torsten Krauß for his research in [81]. Therefore, the main goal of this project can be achieved by extending this application with a module that implements the ALM for adversarial clients.

This section will give a brief overview of the already existing application before explaining the module contributed by this project. As the codebase in its entirety contains approximately 30,000 lines of code, only excerpts will be presented. Furthermore, the code in the following is presented in an abridged form to improve readability and ought to be understood as pseudo-code, mostly following Python syntax.

7.2.1 Federated Learning in Python

The application is launched by a runner script `Runner.py` which gathers all parameters and hands them off to an `class Executor()` object, which then runs an experiment with a fixed set of parameters. The `Runner.py` script starts by parsing parameters from an external `parameter.yaml` file.

```
#Runner.py
params_from_file = InputParamLoader.load_params_from_file('./parameter.yaml')
parameter_parser = ParameterParser(params_from_file)
```

These are parameters that remain constant throughout all experiments, like the number of benign and malicious clients and the number of local training epochs per client. After that, the runner script defines parameters to be changed during experiments.

- `poison_data_rate_list = [...]`
  The poison data rate defines the percentage of an adversarial client's dataset that has been poisoned.
- `random_seed_list = [...]`
  To make experimental results reproducible, the random number generators are seeded with a fixed number.

- `initial_global_model_list = [...]`
  Experiments can be started with either pre-trained global models or randomly initialized naive models.
- `dataset_list = [...]`
  Models can be trained with different benchmark datasets like CIFAR-10 [21], MNIST [100] or GTSRB [101].
- `benign_malicious_lr_list = [...]`
  defines the learning rate for both benign and malicious clients separately
- `backdoor_list = [...]`
  defines a list of backdoor types (semantic backdoor, label-swap, pixel trigger, ...)
- `model_list = [...]`
  contains a list of different model architectures.
- `distribution_list = [...]`
  Client data can either be iid, one-class non-iid, or two-class non-iid See Fig. 5.2 for more explanation.

Nested for-loops then iterate through every parameter list to create a setup for every parameter combination

```
for poison_data_rate in poison_data_rate_list:
    # ... ... ...
        for distribution in distribution_list:

            fl_setup = parameter_parser.add_params(
                poison_data_rate, distribution, # ... ... ...
            )

            executor = Executor(fl_setup)
            executor.run()
```

which is then passed to the experiment executor which now initializes the experimental setup

```
# in class Executor():
def run(self) -> None:
    self.__initial_setup() # create folders, loggers and reserve hardware (GPU)
    self.__initialize_random() # with random seed
    self.__initialize_models_for_training() # create models for every client
    self.__load_dataset() # load dataset as a whole
    ...
```

before running the following steps as part of the experiment:

- `self.__create_federation_client_datasets()`
  The data is split among the federation clients according to the parameters set for data distribution. Furthermore, transformations to images like normalization and random flip are applied (Fig. 7.1).
- `self.__analyze_global_model()`
  The performance of the global model (i.e., prediction accuracy) is evaluated on a test dataset beforehand to allow for comparisons after training.

- `self.__local_training()`
  The local models are trained sequentially for every benign and malicious client. In a real-world scenario, this step would be executed in parallel. **This is also the step where the module developed as part of this project is called.**
- `self.__analyze_and_aggregate_local_models()`
  After training the local models, their performance is evaluated for the first time, and distances to the global model are calculated before aggregating them in different groups (only benign/malicious/all models). Thereafter, the aggregated models are evaluated again.
- `self.__run_defenses()`
  After model aggregation, defense mechanisms, as described in the related work section (Sec. 3.4), can attempt to detect malicious models.
- `self.__visualize_results()`
  As a last step, visualizations of all metrics and results are rendered. Some of them are featured in the next chapter.

(a) original **(b)** transformed

Figure 7.1: Transformations applied to CIFAR-10 images: (a) shows the original images without any transformations applied and (b) shows the same images but normalized to match the mean and variance values averaged over the entire dataset. Some images are flipped horizontally as well.

7.2.2 Augmented Lagrangian Loss

The main implementation of this project is a Python module that implements the ALM for constrained optimization. A first idea is to design the module as a `torch.optim` class similar to `torch.optim.SGD`, as both SGD and Augmented Lagrangian are methods of constrained optimization. The main difficulty with `torch.optim` modules, however, is that they take low-level concepts of PyTorch into account, and most of their code deals with runtime optimization as opposed to the actual optimization algorithm. Looking at the actual Augmented Lagrangian algorithm in the previous chapter (Algorithm 1, Sec. 6.2), one can see that most of the computational complexity stems from the unconstrained optimization step for the Lagrange function (S2). The main difference between the ALM and SGD is that instead of minimizing a loss function directly, one tries to minimize a Lagrange function with a few additional parameters (i.e., Lagrange multipliers) that need to be updated during the optimization process. It would therefore be advantageous to implement the ALM on top of SGD. As can be seen in Sec. 7.1, loss functions in PyTorch can be stateful, meaning they are class instances able to keep track of parameters. Therefore, the Augmented Lagrangian module is implemented as a class that is called like a loss function

during the model training process. Figure 7.2 shows a simplified flowchart demonstrating how the module extends the already existing application.

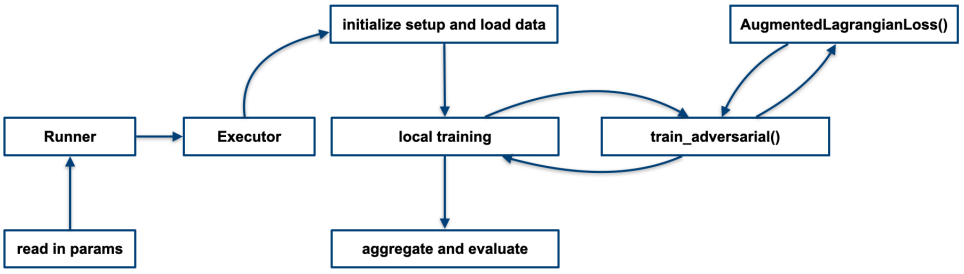


Figure 7.2: Simplified flowchart of the Federated Learning application with the new addition of the Augmented Lagrangian module

The module considers two scenarios:

1. In the default scenario, the training data is poisoned beforehand according to the poison data rate hyperparameter.
2. The optional scenario considers the second research question (RQ2). Here the unconstrained optimization is carried out with an unpoisoned dataset, whereas the poisoned data is used for the evaluation of the optional fifth constraint.

The `class AugmentedLagrangianLoss()` is initialized with the following set of parameters:

- `criterion`: The original loss function (e.g., Categorical Cross Entropy)
- `net`: A pointer to the network of the adversarial client.
- `net0_sate_dict`: The initial state of the local model (i.e., global model)
- `lambda_0`: Initial value for the Lagrange multipliers
- `d_L2, d_cosine`: Targeted values for Euclidean and cosine distances from the local to the global model
- Optional `d_poison`: Targeted loss of the model on the poison samples.

During initialization, the class further sets the hyperparameters for the ALM (cf. Algorithm 1) and the actual limits of the constraints as internal states:

```
# Augmented Lagrangian hyperparameters (Algorithm 1)
    self.rho = 0.1
    self.tau = 0.9
    self.gamma = 1.0 # setting gamma = 1, makes rho constant
# constraint limits
    var = 0.05
    self.eps1 = d_L2 - var
    self.delta1 = d_L2 + var
    self.eps2 = d_cosine - var
    self.delta2 = d_cosine + var
# optional
    self.delta3 = d_poison
```

When initialized, the module provides the following helper functions:

- `def __l2_distance(self)`
  (for internal use) Calculates the Euclidean distance of the local model to the global model
- `def __cos_distance(self)`
  (for internal use) Calculates the cosine distance of the local model to the global model
- `def is_inbound(self)`
  Checks if all constraints are satisfied and returns a boolean value
- `def reinit(self)`
  Re-initializes the module, in particular the Lagrange multipliers
- `def get_metrics(self)`
  Returns a dictionary of all metrics (e.g. last calculated L2/cosine distance), making internal information available for logging

In general, all values calculated by the module are saved as class variables, s.t. the helper functions do not need any additional arguments. Furthermore, all metrics are made accessible externally via the `def get_metrics(self)` function. This allows for evaluation of the module performance and debugging via the logging interface of the already existing application, as well as automated figure generation.

The most important function of the module, however, is `def forward(...)` which calculates the Lagrange function. Akin to other PyTorch loss functions, it takes the model output and designated labels as arguments. Optionally, it also takes additional model outputs and labels for data consisting of exclusively poisoned samples.

```
1  def forward(self, model_output, labels, poison_output=None, poison_labels=None):
2
3      self.main_loss = self.criterion(model_output, labels)
4      self.g1 = (self.eps1 - self.__l2_distance())
5      self.g2 = (self.__l2_distance() - self.delta1)
6      self.g3 = (self.eps2 - self.__cos_distance())
7      self.g4 = (self.__cos_distance() - self.delta2)
8
9      rho = self.rho #only needed for code readability
10     self.adv_loss = 1/(2*rho) * (max(0, self.l1 + rho * self.g1)**2 - self.l1**2) \
11             + 1/(2*rho) * (max(0, self.l2 + rho * self.g2)**2 - self.l2**2) \
12             + 1/(2*rho) * (max(0, self.l3 + rho * self.g3)**2 - self.l3**2) \
13             + 1/(2*rho) * (max(0, self.l4 + rho * self.g4)**2 - self.l4**2) \
14
15     # optional part
16     if poison_output != None:
17         self.ba_loss = self.criterion(poison_output, poison_labels)
18         self.g5 = (self.ba_loss - self.delta3)
19         self.adv_loss += 1/(2*rho) * (max(0, self.l5 + rho * self.g5)**2 - self.l5**2)
20
21     self.total_loss = self.main_loss + self.adv_loss
22     return self.total_loss
```

As a first step, the function computes the classification loss of the model via the original loss function (e.g., cross-entropy loss). This is also the first term of the Lagrangian and, in this case, dubbed *main loss*. Then the constraints $g_1(w, w_0), \ldots, g_4(w, w_0)$ are evaluated before computing the second term of the Lagrangian (here called *adversarial loss*). If necessary,

the procedure is repeated for the last constraint g_5, which is computed by calculating the loss on a poison-only dataset. Finally, the function adds both main and adversarial loss to return the complete Lagrangian but stores both losses separately as class variables to allow for easier logging.

For updating the Lagrange multipliers and penalty parameter ρ / `rho`, the function `def update(self)` can be called. It follows Algorithm 1 from Sec. 6.2 for ρ but offers a choice for the update method of Lagrange multipliers. In accordance with Algorithm 1 the λ_i are updated with

```
self.l1 = max(0, self.l1 + self.rho * self.g1)
# repeat for (l2, g2), ..., (l5, g5)
```

but an alternative choice that turns out to provide better stability for the ALM is

```
self.l1 = self.l1 + self.rho * self.g1 if self.g1 >= 0 else 0
# repeat for (l2, g2), ..., (l5, g5)
```

In the secondary update method, the λ_i are directly set to 0 if the respective constraint is satisfied ($g_i < 0$, not "$\leq$" to satisfy the complementary slackness condition), effectively deactivating the ALM and returning to conventional model training until constraints are violated again.

After initialization, the module can be called as ordinary loss function in the adversarial training loop of the already existing application. This loop is contained by a `def train_adversarial(...)` function called by the `self.__local_training()` function from the executor for every adversarial client individually. In compliance with the threat model (Sec. 5.2), this function can be altered to incorporate the ALM. After initializing the `AugmentedLagrangianLoss` module and optionally loading separate poisoned data, the adversarial training loop is now:

```
# in "def train_adversarial(...)" after initializing ALM_loss
    for epoch in range(self.__malicious_epochs):

        RE_INIT_LAGRANGE_EPOCH = 7 # re-initializes the ALM at epoch 8 (zero-based)
        if epoch == RE_INIT_LAGRANGE_EPOCH:
            ALM_loss.re_init()

        for data, targets in self.__dataset_backdoor: # OPTIONAL self.get_dataset_train()
            main_output = local_model(data)

            # optional
            poison_output = local_model(poison_data)
            # --------

            ALM_loss.forward(main_output, targets, poison_output, poison_targets)
            log(ALM_loss.get_metrics())

            optimizer.zero_grad()
            if epoch < RE_INIT_LAGRANGE_EPOCH:
                ALM_loss.main_loss.backward()
            else:
                ALM_loss.total_loss.backward()

```

```
24              optimizer.step()
25              ALM_loss.update()
26          eval_and_log_performance(local_model, testing_data)
```

As can be seen in lines 4-6, the ALM is re-initialized at a later training epoch, and lines 19-22 show that before re-initialization, the ALM is not used for model training at all, as one takes optimization steps with respect to the main loss. Experiments (Sec. 8) have shown that a delayed start of the ALM is beneficial. The reason for first initializing the Augmented Lagrangian module without actually using it as opposed to waiting with the initialization until later epochs is that its metrics are calculated and logged regardless and give insights into how model distances evolve over multiple epochs of adversarial training. Otherwise, now using the `ALM_loss.forward()` function, as opposed to the original loss function, poses the only major difference in the default scenario, as changes to the already existing `def train_adversarial(...)` function were kept minimal.

7.3 NTK

The defensive approach of this project is based on the eNTK implementation by Yu et al. [19]. In the paper, the authors describe how their eNTK can be used in the last steps of training for models of various architectures (e.g., ResNet18 [26]). The code the authors have open-sourced[1], however, only shows the implementation for a rather simple CNN architecture. Nonetheless, the code can be adapted to work with more complex architectures.

The first half of their code handles a generic Federated Learning setup where models are trained via FedAvg and will not be explained in more detail in this chapter. It should just be noted that modifications were made that allow the setup to handle more complex model architectures like ResNet and distinguish between ordinary clients and malicious clients with poisoned datasets. This section will therefore skip directly to the second stage of model training, where the global model NN is exchanged for an eNTK.

The transition from NN to eNTK begins by re-initializing the output layer of the NN. As explained in Sec. 6.3.1 for the eNTK, only the first output dimension is of further interest. Both steps can be accomplished with a single line of code:

```
1 global_model.fc = nn.Linear(512, 1) # assigns a new, randomly initialized layer
```

The actual eNTK is computed for every client individual as it depends on the client training data and is handled by the `def compute_eNTK(...)` function:

```
1  def compute_eNTK(model, data, subsample_size, seed):
2      torch.manual_seed(seed) # same parameters are now sampled for every eNTK
3
4      # subsample random weights from every layer except for output layer
5      n_weights = model.n_of_weights - model.n_of_weights_output_layer
6      random_weights = torch.randperm(n_weights)[:subsample_size]
7
8      grads = []
9      for i in range(len(data)):
10         model.zero_grad()
11         first_out = model.forward(data[i])[0]
```

[1] https://github.com/yaodongyu/TCT

```
12          first_out.backward()
13
14          grad = torch.Tensor
15          for param in list(model.parameters()):
16              grad = torch.cat([grad, param.grad.flatten()])
17
18          grad = grad[random_weights]
19          grads.append(grad)
20
21      return grads
```

Sec. 6.3.1 explains how the dimensionality of the eNTK is reduced to decrease memory requirements. This is achieved by first calculating the model gradients as usual but only keeping a random subset of the entire gradient, which is comprised of partial derivatives with respect to every model weight. As the random subset needs to remain a constant for every client, the random number generator is re-seeded every time at the beginning of the function before retrieving the random subset (lines 2-6). The rest of the function runs a forward pass on individual samples from the dataset, calculates the gradients, and flattens them to a 1D tensor.

When training the NTK, one first initializes the weights for every client `w_client` as well as the global weights `w_global` with zeros before computing the client updates via the SCAFFOLD [60] algorithm. As SCAFFOLD takes into account differences in previous client updates and the global model updates ("update corrections", Sec. 6.3.2), the `def scaffold_update(...)` function takes both the previous client update and the global model as arguments before calculating any update steps.

```
1  def scaffold_update(grads, y, w_client, h_client, w_0, n_steps, lr):
2
3      # calculate update correction
4      h_updated = h_client + (1 / (n_steps * lr)) * (w_0 - w_client)
5      # now overwrite local model with global model
6      w_client = w_0
7
8      c = 1 / len(y)
9      for i in range(n_steps):
10         # the @ symbol designates matrix multiplication
11         update = lr * (c * grads.t() @ (grads @ w_client - y) - h_updated)
12         w_client = w_client - update
13
14         # logging step size of first update
15         if i == 0:
16             log(update)
17
18     return w_client, h_updated
```

After calculating the updates for all clients and aggregating their weights `w_client` to a new global set of weights `w_global`, predictions on a test dataset can be obtained by first using the `def compute_eNTK(...)` function to compute gradients `grads_test` using the original weights of the NN and then running a simple matrix multiplication `grads_test @ w_global`.

As the dimensionality of objects like `grads` and `w_client` can be obscure, the following example tries to provide clarification: Consider a dataset with pictures of c different classes

and a batch size of b. Furthermore, the original NN has N weights, which are subsampled to a count of n for the eNTK. Then the `grads` object containing the n partial derivatives for each of the b samples has a dimensionality of $b \times n$. The client weights `w_client` now must have a dimensionality of $n \times c$ because the matrix multiplication `grads @ w_client` yields predictions for each of the b samples and c classes (dimensionality $b \times c$). The one-hot encoded labels `y` and update corrections `h_updated` then have dimensionalities of $b \times c$ and $n \times c$, respectively.

8. Evaluation

After implementing (Sec. 7) the newly proposed ideas (Sec. 4), one can conduct experiments to gain insights and evaluate how effective an adversary utilizing the Augmented Lagrangian method (ALM) can adapt to certain distance metrics. Therefore this chapter discusses the results of these experiments by first giving an overview of what parameters were used during experimentation before explaining the actual results. The evolution of local models during training and their gradients are briefly discussed as well. Furthermore, this chapter also shows the results of NTK-based defense, which constitutes the secondary objective of this thesis.

8.1 Hardware and Software

All of the experiments were run on a designated machine learning workstation provided by the Secure Software Systems Group at the University of Würzburg and administrated by Christoph Sendner. The around-the-clock availability of the workstation meant that longer experiments were able to finish overnight. The workstation is equipped with an AMD EPYC 7413 24-Core processor, 128 GB of RAM, and most important for accelerated training of deep learning models: One NVIDIA A16 with four virtual GPUs.

The implementation was tested with version 3.9.12 of the Python interpreter and PyTorch version 1.13.0. The usual Python affiliated libraries Matplotlib, NumPy, Jupyter Notebooks, etc., were used to analyze the experimental output and to generate figures.

8.2 Augmented Lagrangian

8.2.1 Scenarios

The implementation of the ALM was tested as part of the Federated Learning setup described in Sec. 5 with 10 clients, of which 4 were malicious. The benign clients are first to train their local models, so their Euclidean and cosine distances to the global model can be calculated. The adversaries then try to adapt to the mean of the benign model distances. Additional experiments where adversaries train benign models first to adapt to their own benign metrics when training malicious models are run as well. To verify the stability of the ALM across different scenarios, experiments were run for varying sets of hyperparameters, datasets, and backdoor types. Experiments were run on three different datasets (CIFAR-10 [21], GTSRB [101], and MNIST [100]), all of which are popular benchmark datasets for image classification. The types of backdoors that were considered are as follows:

Parameter	Values
Number of Clients	Benign: 6 Malicious: 4
Local Training Epochs	Benign: 10 Malicious: 10, 15, 100
Datasets	CIFAR-10 GTSRB MNIST
Backdoor Types	Pixel Trigger Label Swap Semantic (CIFAR-10 only)
Initial Global Model	Randomly initialized 10 rounds pre-trained with iid data PyTorch pre-trained on ImageNet
Data Distribution	iid one-class non-iid (0.0, . . . , 0.9) two-class non-iid (0.0, . . . , 0.9)
Random Seed	42, 13, 0, 1
Poison Data Rate	0.1, 0.2, . . . , 1.0
Initial Learning Rate	0.1, 0.01, 0.001
Momentum/Weight Decay	0.9/0.005
Batch Size	64

Table 8.1: This table contains all parameters that were used during experimentation with the ALM.

- **Pixel Trigger Backdoor**
  Depicted in Fig. 3.1. In this particular case, the trigger consists of the maximum color value of the image after normalization multiplied by 5. The trigger (size: `image_width` / 16) is located in the lower-left corner of the image.
- **Label-Swap Backdoor**
  All images of class 1 are reassigned to label 2 (CIFAR-10: Automobile → Bird, GTSRB: Speed Limit 30 → Speed Limit 50, MNIST: Digit 1 → Digit 2)
- **Semantic Backdoor**
  The backdoor is implemented as in Fig. 3.1 and only available for the CIFAR-10 dataset. One could theoretically implement it for other datasets, yet the only official implementation by Bagdasaryan et al. [10] considers CIFAR-10 only.

All clients were assigned 40 batches of data with 64 samples each (2560 in total). The distribution of labels for the training data of a particular client was either iid or one-class/two-class non-iid with a parameter ranging from 0.1 to 0.9 in 0.1 step increments that determines the fraction of iid data (see Fig. 5.2). The rest of the data was distributed to the client with the respective main label (see Fig. 5.2). Furthermore, for the initial global model, one can choose between randomly initializing a new model or a model that has already been trained over multiple rounds (in this case, 10). In the latter case, one simulates an adversary gaining control of clients not until a later stage. Furthermore, one can opt for a model that has already been trained on the ImageNet dataset [102] (weights provided by the PyTorch library). In this case, Federated Learning is used for model fine-tuning. These and other parameters like the rate of poisoned data, initial learning rate, and random seeds are summarized in Table 8.1.

As the number of possible parameter combinations grows exponentially with the number

Parameter	Values
Local Training Epochs	Benign: 10, Malicious: 10
Datasets	CIFAR-10
Backdoor Types	Pixel Trigger
Initial Global Model	10 aggregation rounds pre-trained
Data Distribution	two-class non-iid (0.3)
Random Seed	42
Poison Data Rate	0.1
Initial Learning Rate	0.01

Table 8.2: This table shows the parameters for the default scenario. Parameters that are constant throughout all experiments, like the number of clients, are omitted from this table.

of hyperparameters, a default scenario is chosen, and only a single parameter is adjusted at a time compared to the default scenario. For the default scenario, the pixel trigger backdoor was used in combination with the CIFAR-10 backdoor and a two-class non-iid data distribution with an iid rate of 30%. The other parameters are shown in Table 8.2.

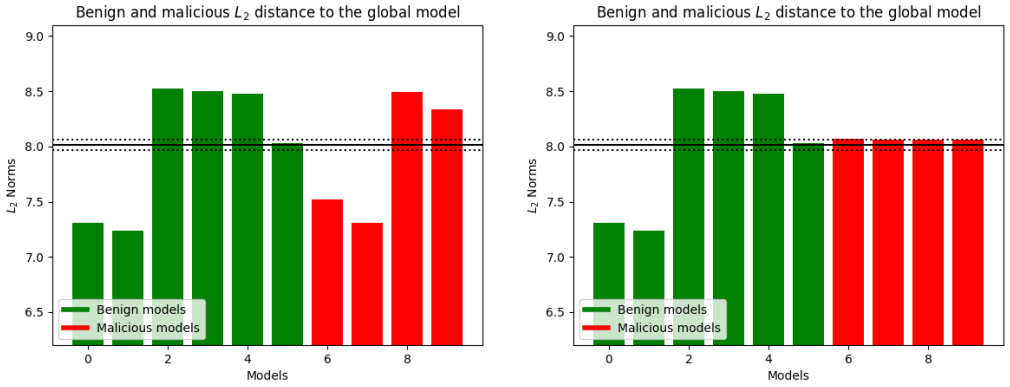
Theoretically, the ALM comes with three hyperparameters of its own (ρ_0, τ and γ, Sec. 6.2, Algorithm 1). Since this project aims to eliminate the adversarial dilemma by eliminating the α parameter, the introduction of new hyperparameters would be counterproductive. Therefore, γ was set to 1, making ρ a constant which was then set to $\rho = 0.1$ for all experiments, effectively eliminating all additional hyperparameters.

8.2.2 Adaptive Adversary Results

The ALM leads to generally favorable results for an adaptive adversary. Initial experiments where the ALM is initialized at the beginning of local training resulted in broken models that performed as badly as a naive classifier or even output NaN (not a number) values. Therefore, the initialization of the ALM was delayed to epoch number 8 early on during the experimentation stage of this project. Therefore, an adversary trains his malicious models without any type of adaption during the first 7 epochs of local training and only then starts to adapt to the target distance metrics. Furthermore, it was found to be beneficial to increase the learning rate to a constant value of 0.1 (no momentum or weight decay) upon initialization of the ALM. This change was kept for all future experiments.

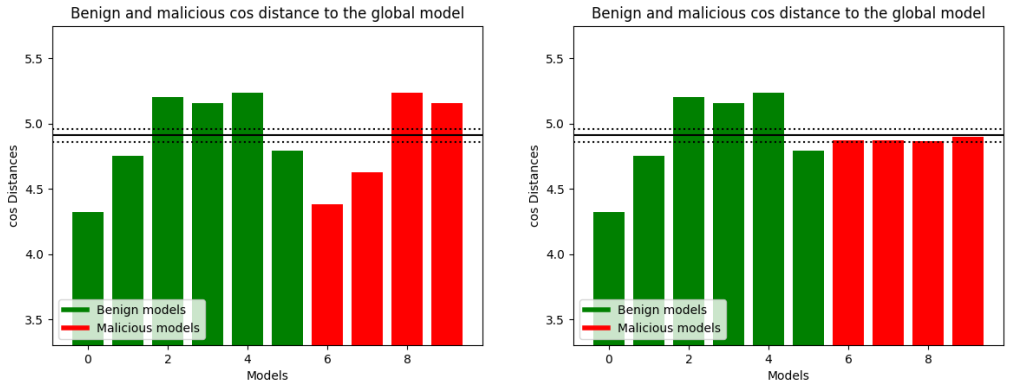
Figure 8.1 shows the Euclidean and cosine distances between the global and local models. As one can see in the figure, the malicious clients can adapt to the constraints in merely three epochs of Lagrangian training. The major eye-catching difference between Euclidean and cosine distances is that while for the Euclidean distance, the malicious models converge to the upper bound of the constraint (dotted line), they tend to converge to the lower bound of the constraint for cosine distances. This behavior can be observed not only for the default scenario but for all other experiments as well. This concentration of malicious models drastically reduces the standard deviation of model distances. For the cosine distance, it decreases from 0.327 to 0.013 and from 0.550 to 0.005 for the Euclidean distance.

The adversarial models in aggregation have a perfect backdoor accuracy of 100%, yet compared to an aggregation of only benign models, the main accuracy drops from 58.60% to 53.03%. Given the poison data rate (pdr) of 0.1, this behavior is expected as a high backdoor accuracy incurs a decrease in main accuracy by definition. The results are combined in Table 8.3. The drop in main accuracy compared to the initial global model when aggregating benign models can be explained by the data distribution. As the model



(a) L_2 norm, before

(b) L_2 norm, after



(c) cos distance, before

(d) cosine distance, after

Figure 8.1: L_2 and cosine distances of local models to the global model in the default scenario (non-iid data). The black line marks the mean of the benign models, and the dotted lines mark the permitted interval for the adaption.

Aggregation	Main acc. / %	Backdoor acc. / %
Initial	58.60	1.57
Only Benign Models	58.29	0.28
Only Adv. Models	53.03	100.00
All Models	57.45	98.96

Table 8.3: Shown in this table are classification accuracies before and after aggregating either only benign or malicious models or all models for the default scenario (non-iid data).

Aggregation	Main acc. / %	Backdoor acc. / %
Initial	58.60	1.57
Only Benign Models	58.89	1.41
Only Adv. Models	**59.18**	100.00
All Models	59.15	99.23

Table 8.4: Shown in this table are classification accuracies before and after aggregating either only benign or malicious models or all models for the default scenario with iid data.

was pre-trained on iid data, this slight drop in accuracy is expected behavior as well. After aggregating all models, the backdoor persists with near perfect accuracy. When changing the data distribution to iid, the aggregated adversarial models even have a higher main accuracy of 59.18% compared to only 58.89% when only aggregating benign models (Table 8.4). This result looks like an outlier at first but can be attributed to the use of a higher learning rate by the ALM compared to the benign, SGD-trained models. Later experiments compare SGD-trained benign models with ALM-trained adversarial models while using the same learning rate for both models (Sec 8.2.4).

While the adversarial clients in Figure 8.1 seem to reach the upper or lower bound of the constraints at all times, a close inspection of the raw numbers reveals that the constraints were not met completely. The first of the adversarial clients (left most red bar in the figure) has a Euclidean distance of 8.072 after 10 epochs of local training. The permitted range, i.e., the average of benign Euclidean distances ± 0.05, however, is $[7.964, 8.064]$. This means that the permitted range above the mean was overshot by a factor of $(8.072 - 8.014)/0.05 = 1.16$. This measure can be formalized by defining an overshoot factor

$$\eta = \frac{\max(\{|\text{distances of adv. models - benign model mean distance}|\})}{\text{half interval size}} \tag{8.1}$$

for the Euclidean and cosine distances. Put in words, this coefficient describes the maximum difference between the model distance of any adversarial model and the mean distance of all benign models in relation to the permitted interval size. A factor of $\eta \leq 1$ means that the constraint is satisfied. For the iid scenario, this factor increases to $\eta_{L2} = 2.05$. This inaccuracy can be remedied by increasing the number of local training epochs to 15. In both scenarios, these factors decrease to 0.92 and 0.99 for the non-iid and iid scenarios, respectively, meaning the constraints are satisfied. The reason for not making 15 local training epochs the default is that the adaption becomes too perfect, making it easy for a defense to sort out a group of clients with exactly matching Euclidean distances, even though the latter result is the more desirable outcome from a mathematical perspective. Furthermore, prolonging the local training sessions from 10 to 15 epochs does not improve the prediction accuracies in any meaningful way. On the other hand, Fig. 8.1 shows that despite missing the constraints by a small factor, the ALM produces very acceptable results after merely 3 epochs (reminder: the method was only used during epochs 8-10).

To show the efficiency of the ALM when given more time, the default number of local training epochs was changed to 15 epochs. Table 8.5 shows a selection of results for various scenarios. On rare occasions, the constraints were still not met at the end of 15 epochs, though a later analysis shows that they were indeed fulfilled at least at one point during training. In summary, it can be said that the results are favorable in nearly all scenarios. Not a single experiment is known to yield worse results compared to an equivalent experiment with non-adaptive adversaries (no Augmented Lagrangian) in terms of classification accuracy, which could not be explained by random chance. Though, for the sake of transparency, it should be mentioned that a "mirror" experiment with non-adaptive adversaries was not conducted for all scenarios due to time limitations.

Table 8.5 also shows that the ALM seems to struggle with random initial models in regards to L2 adaption. While classification accuracies barely better than for a naive classifier are expected, the overshoot factor assumes values of up to $\eta_{L2} = 1.46$. Nonetheless, Figure 8.1 shows that values in this order of magnitude are not a bad outcome. With the knowledge that the ALM tends to overshoot on the upper side for Euclidean distances, one can target a slightly smaller distance than the benign model mean (e.g., 7.964 as opposed to 8.064) and be left with a model satisfying the initial constraints. One outlier, however, that cannot be explained is the two-class non-iid (0.2) setup with a semantic backdoor with an L2 overshoot factor of $\eta_{L2} = 3.41$. As a later analysis shows, the constraints were

Change from Default	init. MA	ben. MA	adv. MA	BA all	η_{L2}	$\eta_{\cos}$
Semantic Backdoor						
iid data	58.60	58.59	60.05	60.00	0.98	0.98
pdr 0.5	58.60	58.29	38.99	100.00	0.89	1.00
PyTorch Pre-Trained	10.55	38.07	36.60	80.00	1.00	0.83
Random Initial Model	11.35	10.79	10.81	60.00	1.46	0.72
two-class non-iid (0.2)	58.60	52.84	46.00	80.00	3.41	0.95
Pixel Trigger Backdoor						
MNIST Dataset	97.24	96.25	96.31	87.11	1.01	0.99
GTSRB Dataset / iid data	83.86	84.03	82.43	68.11	0.91	1.00
Random Initial Model	11.35	10.79	12.23	61.51	1.13	0.65
one-class non-iid (0.2)	58.60	60.08	51.26	99.48	0.69	0.53
pdr 0.7	58.60	58.29	24.82	100.00	0.59	1.00
Label-Swap Backdoor						
iid Data	58.60	58.89	54.33	28.20	1.00	0.99
Random Initial Model	11.35	10.79	10.00	100.00	1.20	0.75

Table 8.5: The table shows main accuracies for the initial model, the benign models aggregated and the adversarial models aggregated after training with the ALM, as well as the backdoor accuracy after aggregating all models. Furthermore, the table lists the two overshoot factors η_{L2} and $\eta_{\cos}$.

satisfied during earlier stages of training, meaning early stopping could have prevented this outcome.

Lastly, experiments with adversaries adapting to their own benign models were run. The results thereof do not lend themselves particularly well to visualization, as every client attempts to adapt to different distances. Nonetheless, the results were of similar quality compared to all of the previous experiments ($\eta_{L2} < 1.20$). Corollary, adversaries can adapt to any other desired distance value as long as its size is of comparable order of magnitude. Attempting to adapt to distance values like 1000 produces either naive or outright broken models with NaN output. Results with different random number generator seeds yielded similar results as well. In some experiments, the initial learning rate was changed according to the values in Table 8.1. For the case of $\text{lr} = 0.1$, the same value as for the ALM, most local models became naive, justifying a different choice for the initial learning rate. Experiments with a lower initial learning rate of $\text{lr} = 0.001$, in turn, lead to positive results.

For 10 epochs of local training, the benign clients needed ≈ 29 seconds on average, whereas adversarial clients training for 10 epochs on average needed ≈ 77 seconds. This 2.7-fold increase can be attributed to the adversary needing to compute distance metrics for every optimization step. The time needed for a higher number of epochs scales linearly. While an increase of this size is considerable, one needs to keep in mind that FL was specifically designed to work with clients with vastly differing computational capabilities. A client needing almost three times as long for computing updates is not necessarily indicative of malicious behavior.

8.2.3 Model Evolution

The inexplicable result of the outlier with $\eta_{L2} = 3.41$ motivates further inquiry into the evolution of the local model during training. Figure 8.2 shows the Euclidean distance of the client model that caused the outlier result during the Lagrangian training stage. The figure shows that the client model was clearly within the bounds of the constraint

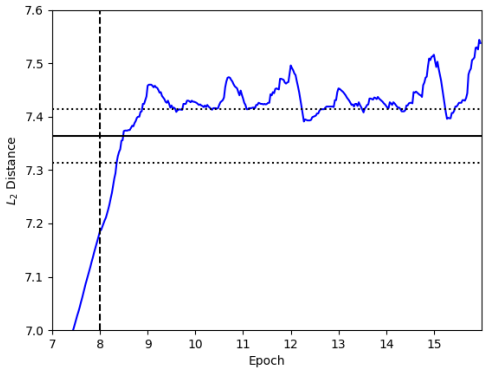


Figure 8.2: L_2 distance of the adversary model causing the overshoot factor of $\eta_{L2} = 3.41$. The horizontal lines mark the permitted interval for adaption, and the vertical line marks the start of Augmented Lagrangian training.

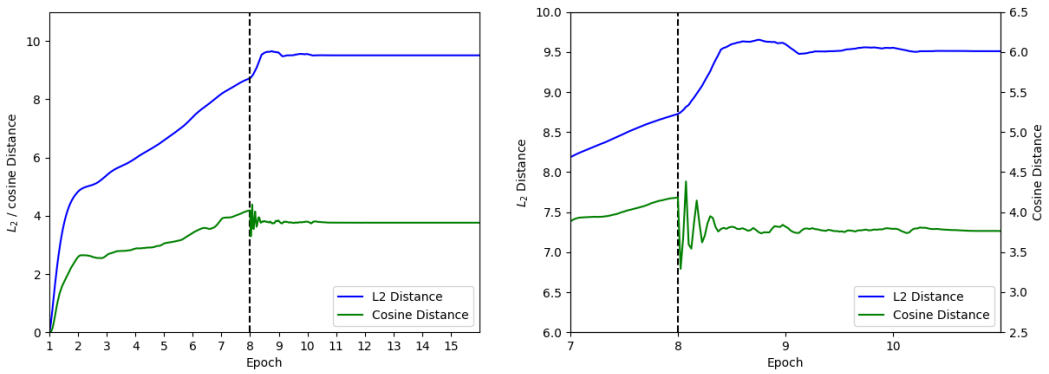
at multiple points during the training process. Not until the end of the last epoch did the Euclidean distance deviate thus far from the benign model mean. By stopping the training process early while the constraints are satisfied (e.g., during epoch number 12), the adversary can ensure the desired outcome. Given the stability of the ALM in this scenario (discussed in Sec. 8.2.4), the adversary is very likely to be left with a model of acceptable performance when stopping early. The actual cause for the sharp increase in distance at the end remains unknown but is of no further concern, as it can be prevented by early stopping.

Figure 8.3 shows the evolution of both Euclidean and cosine distances of an adversarial client training a model with a semantic backdoor and iid data. The depicted scenario is exemplary for most other parameter combinations that were tested. Subfigure (a) shows how during the initial, non-adaptive adversarial training stage, both metrics sharply increase during the first epochs and then continue to increase gradually. Subfigure (b) highlights the transition to the ALM at the beginning of epoch number 8. During the following epoch, the model rapidly adjusts to meet the targeted Euclidean distance, and the cosine distance takes on the form of a damped oscillator before eventually converging to the target. From there on, both metrics see little change in epochs 9 through 11. Changes after epoch number 11 are barely measurable, meaning the ALM has converged to the constraints in less than 5 epochs of training.

8.2.4 Gradient Size

To further investigate the stability of the ALM over a longer period, an experiment with pixel-trigger backdoor and iid data was conducted with 100 epochs of local training. Figure 8.4 shows the L2 Norm of the model gradient and cross-entropy loss on training data (pdr=0.1) at the end of every epoch for one adversarial client. The results are compared with the same client (now benign) training a model on an unpoisoned dataset. As for the ALM, the learning rate for the benign client was also increased to 0.1 at the beginning of epoch number 8. In the figure, the results of the benign client are barely visible, as both experiments effectively lead to the same result. In both cases, the loss of training data, as well as the size of model gradients, rapidly decreases when switching to the ALM learning rate and, over time, tends to zero in later epochs. For the adversarial models, the overshoot factors were $\eta_{L2} = 1.00$ and $\eta_{\cos} = 0.43$, meaning the distance constraints were satisfied.

Despite the classification loss on training data and gradient sizes approaching zero, the



(a) All epochs **(b)** Transition to aug. Lag. method

Figure 8.3: L_2 and cosine distances of an adversary training a local model with semantic backdoor and iid data. The vertical line marks the start of Augmented Lagrangian training.

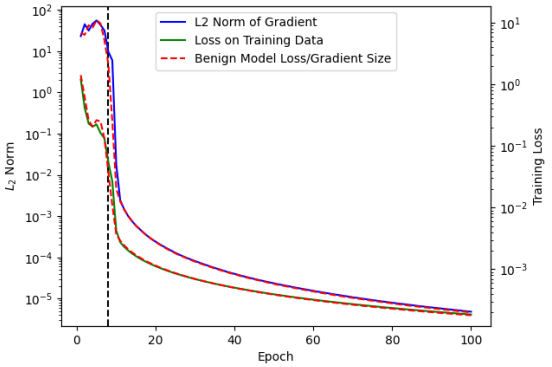


Figure 8.4: L2 norm of a model gradient and loss on the training data over 100 epochs of training. Both scales are logarithmic. The figure depicts an adversarial client training a pixel-trigger backdoor with iid data. The vertical line marks the start of Augmented Lagrangian training. An additional plot with dashed lines shows these metrics for the same client, now benign, training on an unpoisoned dataset with the same hyperparameters, which includes the increase in learning rate from 0.01 to 0.1 at the beginning of epoch number 8.

model is vastly overfitted, with the main accuracy on the testing dataset still remaining at 53.94%. Even though 100 epochs of local training might not aid in training a model, this experiment is nonetheless insightful from a mathematical perspective: It underscores the stability of the ALM as loss and gradient sizes are monotonously decreasing during model training. The roughly equivalent gradient sizes compared to a benign client using plain SGD show that constrained adversarial training with the ALM is just as effective as conventional model training with SGD. Furthermore, as the loss function is designed to have a global minimum at 0, the results are an indication for the ALM converging to a global minimum. Though, the last statement is unproven, as there very well may exist a local minimum close to (but not at) 0. With gradient sizes of $\approx 10^{-6}$ and constraints satisfied to two significant digits ($\eta_{L2} = 1.00$), one can say that the KKT conditions (Sec. 6.2) for a solution to the optimization problem are almost fulfilled. As experiments from Tab. 8.5 only ran for 15 epochs of local training, their model gradient sizes were only $\approx 10^{-3}$ at the end. For models with overshoot factors of $\eta_{L2} < 1$ and $\eta_{\cos} < 1$ the complementary slackness condition was satisfied by default due to the update rule Eq. 6.17.

8.2.5 Defenses

To evaluate the stealthiness of adversarial models trained with the ALM, additional experiments with defenses that attempt to sort out these models before aggregation were conducted. The adversarial models were trained with a semantic backdoor in an otherwise default scenario and tested on (multi) Krum[61] and HDBSCAN [80] clustering, which is the basis of FLAME [79].

Krum, by default, only picks a single model. The defense does so by first calculating the pairwise cosine distances of local models to each other. The model with the least overall distance to other models, save for outliers, is then chosen for aggregation. In the above scenario, Krum chose a benign model. Even multi Krum, when set up to pick three models, exclusively picks benign models. The first adversarial model is picked not until setting multi Krum up to pick four models. One explanation for adaptive adversarial models struggling to overcome Krum is that they were trained to keep certain distances to the global model, *not other local models*. As shown earlier, the ALM can be used to adapt to arbitrary distance values. Therefore, it is theoretically possible to train models while keeping specified Euclidean and cosine distances to other benign models. Such experiments were, however, out of the scope of this project.

The clustering defense also calculates pairwise cosine distances like Krum but uses the HDBSCAN algorithm to generate two clusters. All four of the adversarial models were chosen for aggregation, along with two benign models. The conclusion is that adapting malicious model distances (with respect to the global model) to the mean of benign model distances does not automatically minimize the distance to other models, yet it also does not produce outliers that can be caught by a clustering algorithm.

8.2.6 Backdoor as Constraint

For the last group of adversarial experiments, the poisoned data was kept separate from the original client data. A fifth constraint was introduced, as explained in Sec. 4.1, that limited the loss on the poisoned data. The results were not convincing as it reintroduced a trade-off between main and backdoor accuracy. Considered were two scenarios with iid data and pixel-trigger as well as a semantic backdoor. At first, the poisoned data was limited to a single batch of 64 samples, and the loss on the poisoned dataset was constrained to ≤ 0.01. In both scenarios, all clients failed to implement a backdoor (0% semantic / 1.47% pixel-trigger backdoor, aggregation of only adversarial models). Over the course of the following experiments, the amount of poison data was increased to 5 batches, the constraint limit was set to 0, and the first four constraints (Euclidean and cosine distance adaption) were dropped. By the end, the backdoor accuracy among adversarial models increased to 40.00% which fell back to 0% when aggregating all models for the semantic backdoor. The pixel-trigger backdoor showed little persistence as well, as can be seen in Tab. 8.6. Furthermore, the main accuracy decreases considerably with the pixel-trigger backdoor, even leading to a naive classifier. The increase to 5 batches of poison data also means that the pdr is now on par $(5/(40+5) \approx 0.11)$ with the original pdr of 0.1, whereas the intent of this setup was to maintain a smaller pdr. In summary, the experiments with separate poisoned data produced worse results in every metric.

8.3 NTK Defense

To evaluate the NTK-based defense proposed in Sec. 4.2, the open-sourced implementation by Yu et al. [19] was used with slight modifications as explained in Sec. 7.3. After loading a pre-trained model, the NN was trained with one additional round of local training with a limited number of 4 epochs plus aggregation. Up until that point, no client data had been

Aggregation	MA before	BA before	MA after	BA after
Semantic Backdoor				
Only Adv. Models	60.05	80.00	46.89	40.00
All Models	59.12	60.00	56.74	0.00
Pixel-Trigger Backdoor				
Only Adv. Models	59.83	99.97	10.03	100.00
All Models	59.30	99.92	23.75	78.13

Table 8.6: This table compares the main and backdoor accuracies (MA / BA) of aggregated models before and after separating poisoned data and introducing the fifth constraint.

poisoned. Simultaneously with the transition to eNTK training, the data of 4 out of the 10 clients was poisoned. During eNTK training, the clients were limited to a single batch of data (batch size = 64). Furthermore, their learning rate was set to 0.5, and the eNTK weights were updated with 5000 steps before submitting the weights for aggregation. All of the clients had 70% two-class non-iid data. The initial global model was either pre-trained on ImageNet and provided by PyTorch or pre-trained on CIFAR-10 with iid data for 9 rounds of aggregation. The two backdoor types considered were either the semantic or pixel-trigger backdoor from Sec. 8.2.1. Any combination of initial global model and backdoor leads to four different scenarios to be considered.

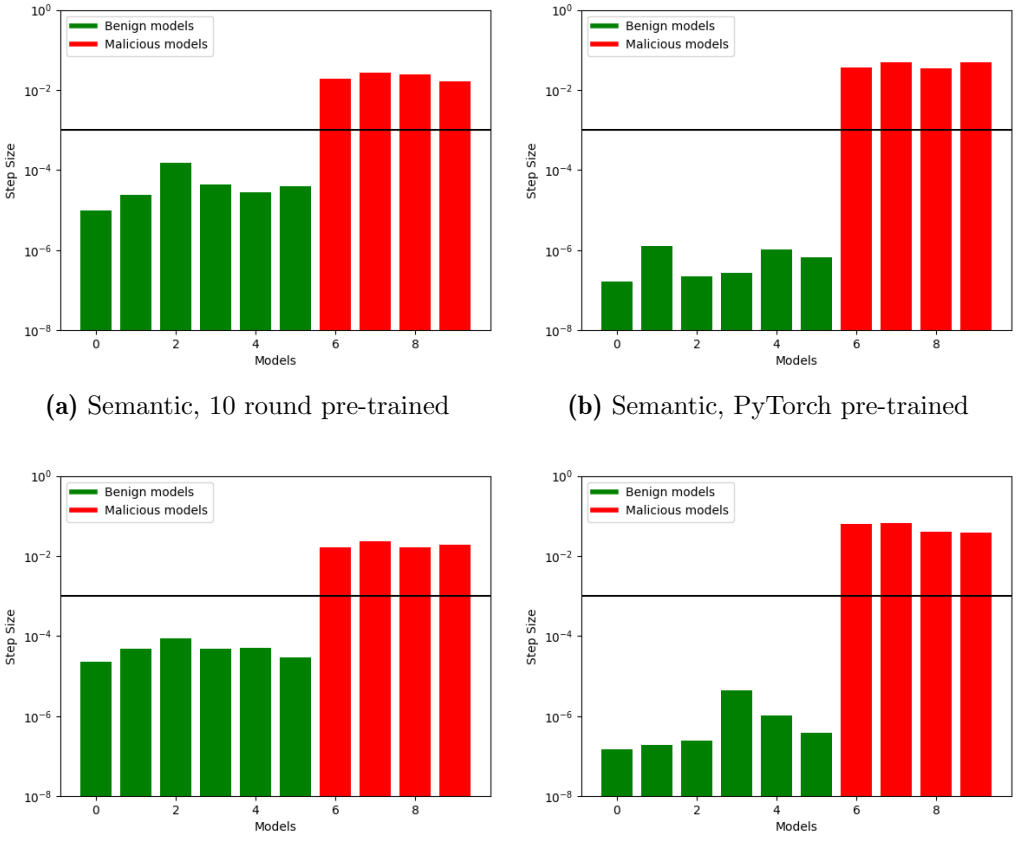
After updating their eNTK weights for 5000 steps, the clients not only sent their updated weights to the server for aggregation but also their datasets. The important detail is that the adversarial clients only send their unpoisoned data to the server (discussed in Sec.4.2). With the data and updated weights, the server then calculates a single additional update step for every client and compares the step sizes. With a global threshold of 1×10^{-3}, the server flags every model update causing larger step sizes as potentially malicious and does not include these models in the aggregation process. The results for each of the four scenarios are shown in Fig. 8.5. The server is able to detect malicious models without any false positives or false negatives. In other words: The defense works flawlessly.

Table 8.7 shows the model accuracies before and after aggregation of the benign clients. As the classification accuracy of each model improves after aggregation, it can confidently be said that the defense does not interfere with the training process. Nonetheless, the results of Yu et al. [19] could not be fully replicated, as much higher increases were expected. The authors report an increase from 35.37% to 77.93% for a ResNet18 trained on CIFAR-10 data. However, these findings could not even be replicated in initial experiments without any adversaries or defense in place. During experimentation, it could be observed that any tweaking of hyperparameters that resulted in a higher main accuracy also increases the difference in server-calculated update steps between benign and malicious models. Therefore, any set of hyperparameters that improves the eNTK classification accuracy also makes the defense less susceptible to the choice of the threshold. Table 8.7 further shows the average backdoor accuracy for every scenario. As the malicious clients detected by the defense managed to install very effective backdoors (above 90% in most scenarios), it can also be said that switching to eNTK training alone does not prevent adversaries from installing backdoors.

For the entire local training process, the average client needs about 8.5 seconds. This measure includes the time needed to calculate the model gradients needed for the eNTK and the 5000 local update steps. A single update step only takes about 0.001 seconds. But since the server must also calculate the model gradients for every client before making his additional update step, he faces a lot of overhead. On average, the defense needs about 38 seconds to evaluate all 10 clients.

Scenario	MA Before	MA After	avg. BA malicous models
Semantic Backdoor			
10 Round Pre-Trained	58.90	62.50	95.00
PyTorch Pre-Trained	36.30	40.63	90.00
Pixel-Trigger Backdoor			
10 Round Pre-Trained	60.00	65.63	94.92
PyTorch Pre-Trained	33.90	50.00	71.48

Table 8.7: This table compares the main and backdoor accuracies (MA / BA) of all four scenarios before and after aggregation. The table furthermore shows the average BA of the malicious models that were detected by the defense.



(a) Semantic, 10 round pre-trained

(b) Semantic, PyTorch pre-trained

(c) Pixel-Trigger, 10 round pre-trained

(d) Pixel-Trigger, PyTorch pre-trained

Figure 8.5: The figure shows the step size of the additional update step taken by the server for every client in each of the four scenarios. The step sizes are plotted on a log scale. The horizonal line at 1×10^{-3} marks the threshold above which client updates are marked as malicious.

9. Conclusion and Future Work

This concluding chapter summarizes the results of this project and argues why they pose significant contributions to the field of adversarial FL (Sec. 9.1). Furthermore, this chapter gives ideas for future work (Sec. 9.2).

9.1 Conclusion

The concept of FL was proposed initially to address growing concerns about user privacy. It allows for the utilization of unused computational resources of end-user devices as well as their private data for the purpose of training an ML model while keeping said data inaccessible to any entity but the end-user device itself. The focus on user privacy, however, comes with a loss of control over the model training procedure. This allows adversaries to install so-called backdoors into the model in the hope of remaining undetected (stealthiness).

To remain stealthy, the adversary tries to train a model that not only contains the desired backdoor but also has a distance to the initial model which does not exceed a certain threshold. Until now, the adversary achieved this goal by adding the distance to the global model to the loss function as a secondary objective to be minimized during training. This additional objective was then weighted with a parameter $\alpha \in (0, 1)$. However, the adversary still faced a dilemma: If the chosen parameter is too small, the secondary objective would not be learned properly, and a simple defense could easily detect the backdoor. On the other hand, by choosing a parameter that is too high, only the secondary objective would be learned, rendering the backdoor ineffective. This thesis successfully circumvented this dilemma by rephrasing the training objective into a constrained optimization problem that could be solved with the Augmented Lagrangian Method (ALM). By reformulating the secondary objective into a set of mathematical constraints with the ALM, the adversary is able to exclusively train backdoored models satisfying these constraints. Though the ALM comes with additional hyperparameters in theory, a single set of hyperparameters was found to yield satisfactory results for all experiments, effectively eliminating the need for the α parameter. This means that an adversary from now on no longer needs to decide between installing an effective backdoor or limiting the model distance. The adversary also no longer needs to spend time in search of the right α value to balance this trade-off. It was further shown that with the ALM, an adversary could train a model adhering to distance constraints in as little as three epochs of local training. Once the constraints are met, the ALM remains stable and makes the size of the model gradient approach zero

while showing little to no deviation from the constraints, which is indicative of finding a local minimum.

Furthermore, with his backdoored model, the adversary is able to bypass a clustering-based defense that measures the cosine distance to other client models. This is a non-trivial outcome as for this project, the ALM only considered the cosine distance to the global model, and the adversary was subsequently unable to bypass a second defense (Krum[61]) which evaluates the same metric. Therefore, if an adversary wants to reliably bypass a defense, he needs to instruct the ALM to adapt to the specific metrics evaluated by this defense.

From a defensive perspective, this thesis also provided interesting results. As an NN learns the most important features in the early stages of training, its weights barely change in later training epochs. Therefore, once the model has been trained to a certain degree, one can make an approximation of the NN and continue training with a so-called Neural Tangent Kernel (NTK). This now linear classifier can be trained much faster to attain better prediction accuracy. The use of NTKs in an FL setup has already been shown to provide better results compared to traditional model training [19]. This thesis further showed that when switching to NTK training, a server can detect every single malicious model by requiring the clients to reveal their training data and calculating a single additional update step. This, however, stands in sharp contrast to the original purpose of FL, which aims to keep user data private. Nonetheless, due to its other benefits, FL has since found applications in scenarios where the privacy of user data is not a number one priority. Therefore, the findings can be used to design an effective defense against adversaries with little additional effort. This defense is based on the assumption that an adversary does not want to reveal his poisoned data and, therefore, only sends benign data to the server. His attack is discovered by the fact that his model was optimized for the poisoned and not the benign data. Calculating a single additional update step reveals this disparity to the server. This is a trivial result from a mathematical perspective, yet as of now, this principle has not been used in a defense against adversaries. With the initial purpose of detecting backdoors, this defense can generally detect every model that has been trained with different data than reported and can therefore be deployed more broadly going forward.

9.2 Future Work

In the evaluation section (Sec. 8.2.1), it was shown that an adversary could use the ALM to adapt his malicious model to desired target distances. While the use of the ALM turned out to be effective in virtually all tested scenarios, not every imaginable setup was tested. Most notable is the absence of model architectures other than ResNet18[26] among the experiments. It would therefore be interesting to see if results on other architectures with a higher number of weights like AlexNet[92] ($\approx$ 62.4 million vs. 11.2 million for ResNet18) are similar. Furthermore, the malicious models obtained with the ALM were merely tested against two defenses. By using the ALM to specifically adapt to metrics evaluated by a defense, one can attempt to bypass other state-of-the-art defenses in the future.

The defense based on NTKs was presented as a proof of concept. During the experimental stage of this project, it was only evaluated on a selected few scenarios. Nonetheless, the few but promising results are an invitation to conduct further experiments and more thoroughly evaluate its efficiency in other scenarios. Its major drawback is that it is based on the assumption that the server has access to client data. In the future, one can make attempts to improve the defense to the point where it can be reconciled with the imperative of keeping client data private, e.g., by developing a zero-knowledge proof.

While the results presented in this thesis were mostly positive, not every research question could be answered affirmatively. The second research question asked if an adversary can formulate prediction loss on a small set of poisoned data as a constraint to reliably install a backdoor with a specified minimum accuracy using the ALM. However, experiments in regards to this question remained inconclusive, as they re-introduced the very dilemma that was eliminated for distance adaptions: The adversary has to choose between either fulfilling the constraint or obtaining a high prediction accuracy. As these experiments were conducted with the same hyperparameters as all prior experiments, a different set of hyperparameters might lead to a better outcome. A search for better hyperparameters is therefore left as potential future work.

List of Figures

Icons in figures are courtesy of Flaticon.com [103].

List of Tables

Bibliography

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-Efficient learning of deep networks from decentralized data," in *The 20th International Conference on Artificial Intelligence and Statistics* (A. Singh and J. Zhu, eds.), vol. 54 of *Proceedings of Machine Learning Research*, pp. 1273–1282, PMLR, 20–22 Apr 2017.

[2] H. Zhu, J. Xu, S. Liu, and Y. Jin, "Federated Learning on non-iid data: A survey," *Neurocomput.*, vol. 465, p. 371–390, nov 2021.

[3] "General Data Protection Regulation," 2018. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`.

[4] "Personal Information Protection and Electronic Documents Act," 2000. `https://laws-lois.justice.gc.ca/eng/acts/p-8.6/`.

[5] "Health Insurance Portability and Accountability Act," 1996. `https://www.govinfo.gov/content/pkg/PLAW-104publ191/pdf/PLAW-104publ191.pdf`.

[6] H. Zhang, J. Bosch, and H. H. Olsson, "End-to-End Federated Learning for autonomous driving vehicles," in *International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021*, pp. 1–8, IEEE, 2021.

[7] N. Rieke, J. Hancox, W. Li, F. Milletarì, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein, S. Ourselin, M. Sheller, R. M. Summers, A. Trask, D. Xu, M. Baust, and M. J. Cardoso, "The future of digital health with Federated Learning," *npj Digital Medicine*, vol. 3, p. 119, Sep 2020.

[8] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated Learning for mobile keyboard prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[9] B. M. et al, "Federated Learning with formal differential privacy guarantees." `https://ai.googleblog.com/2022/02/federated-learning-with-formal.html`. Accessed: May 2023.

[10] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor Federated Learning," in *The Twenty Third International Conference on Artificial Intelligence and Statistics* (S. Chiappa and R. Calandra, eds.), vol. 108 of *Proceedings of Machine Learning Research*, pp. 2938–2948, PMLR, 26–28 Aug 2020.

[11] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, "Analyzing Federated Learning through an adversarial lens," in *The 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 634–643, PMLR, 09–15 Jun 2019.

[12] K. Bryan and Y. Shibberu, "Penalty functions and constrained optimization," *Dept. of Mathematics, Rose-Hulman Institute of Technology. http://www. rosehulman. edu/bryan/lottamath/penalty. pdf*, 2005.

[13] C. Kanzow, "A survery of safeguarded Augmented Lagrangian methods in optimization," *Unpublished*, 2020.

[14] M. R. Hestenes, "Multiplier and gradient methods," *Journal of optimization theory and applications*, vol. 4, no. 5, pp. 303–320, 1969.

[15] R. T. Rockafellar, "The multiplier method of Hestenes and Powell applied to convex programming," *Journal of Optimization Theory and Applications*, vol. 12, pp. 555–562, Dec 1973.

[16] "Wikipedia - Augmented Lagrangian method." `https://en.wikipedia.org/wiki/Augmented_Lagrangian_method`. Accessed: May 2023.

[17] S. Fort, G. K. Dziugaite, M. Paul, S. Kharaghani, D. M. Roy, and S. Ganguli, "Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the Neural Tangent Kernel," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5850–5861, 2020.

[18] A. Jacot, F. Gabriel, and C. Hongler, "Neural Tangent Kernel: Convergence and generalization in neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[19] Y. Yu, A. Wei, S. P. Karimireddy, Y. Ma, and M. I. Jordan, "TCT: Convexifying Federated Learning using bootstrapped Neural Tangent Kernels," *arXiv preprint arXiv:2207.06343*, 2022.

[20] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[21] "The CIFAR-10 dataset." `https://www.cs.toronto.edu/~kriz/cifar.html`. Accessed: May 2023.

[22] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[23] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, *et al.*, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.

[24] Z. Ghahramani, "Unsupervised learning," *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures*, pp. 72–112, 2004.

[25] "Wikipedia - Stochastic gradient descent." `https://en.wikipedia.org/wiki/Stochastic_gradient_descent`. Accessed: May 2023.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[28] "Wikipedia - One hot." `https://en.wikipedia.org/wiki/One-hot`. Accessed: May 2023.

[29] Wikipedia, "Wikipedia - Sigmoid function." `https://en.wikipedia.org/wiki/Sigmoid_function`. Accessed: May 2023.

[30] Wikipedia, "Wikipedia - Rectifier (neural networks)." `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`. Accessed: May 2023.

[31] "Wikipedia - Hyperbolic functions." https://en.wikipedia.org/wiki/Hyperbolic_functions. Accessed: May 2023.

[32] Wikipedia, "Wikipedia - Softmax function." https://en.wikipedia.org/wiki/Softmax_function. Accessed: May 2023.

[33] J. Brownlee, "How do convolutional layers work in deep learning neural networks?." https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/. Accessed: May 2023.

[34] "Wikipedia - Multidimensional discrete convolution." https://en.wikipedia.org/wiki/Multidimensional_discrete_convolution. Accessed: May 2023.

[35] K. Doshi, "Batch norm explained visually — How it works, and why neural networks need it." https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739. Accessed: May 2023.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[37] J. Brownlee, "A gentle introduction to pooling layers for convolutional neural networks." https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/. Accessed: May 2023.

[38] "Wikipedia - Convex function." https://en.wikipedia.org/wiki/Convex_function. Accessed: May 2023.

[39] F. Ramzan, M. U. G. Khan, A. Rehmat, S. Iqbal, T. Saba, A. Rehman, and Z. Mehmood, "A deep learning approach for automated diagnosis and multi-class classification of Alzheimer's disease stages using resting-state fMRI and residual neural networks," *Journal of medical systems*, vol. 44, pp. 1–16, 2020.

[40] K. E. Koech, "Cross-Entropy loss function." https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e. Accessed: May 2023.

[41] "Wikipedia - Newton's method in optimization." https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization. Accessed: May 2023.

[42] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, PMLR, 2013.

[43] A. Kumar, "Weight decay in machine learning: Concepts." https://vitalflux.com/weight-decay-in-machine-learning-concepts/. Accessed: May 2023.

[44] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[45] PyTorch, "PyTorch." https://pytorch.org/. Accessed: May 2023.

[46] "Wikipedia - Taylor series." https://en.wikipedia.org/wiki/Taylor_series. Accessed: May 2023.

[47] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[48] M. A. Mohamadi and D. J. Sutherland, "A fast, well-founded approximation to the empirical Neural Tangent Kernel," *arXiv preprint arXiv:2206.12543*, 2022.

[49] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Transactions on Automatic Control*, vol. 31, no. 9, pp. 803–812, 1986.

[50] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, (Los Angeles, California), pp. 456–464, Association for Computational Linguistics, June 2010.

[51] D. Povey, X. Zhang, and S. Khudanpur, "Parallel training of DNNs with natural gradient and parameter averaging," 2014.

[52] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," in *Advances in Neural Information Processing Systems* (C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, eds.), vol. 28, Curran Associates, Inc., 2015.

[53] L. Chu, L. Wang, Y. Dong, J. Pei, Z. Zhou, and Y. Zhang, "Fedfair: Training fair models in cross-silo Federated Learning," *arXiv preprint arXiv:2109.05662*, 2021.

[54] C. Huang, J. Huang, and X. Liu, "Cross-silo Federated Learning: Challenges and opportunities," *arXiv preprint arXiv:2206.12949*, 2022.

[55] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "Batchcrypt: Efficient homomorphic encryption for cross-silo Federated Learning," in *The 2020 USENIX Annual Technical Conference (USENIX ATC 2020)*, 2020.

[56] Y. Huang, L. Chu, Z. Zhou, L. Wang, J. Liu, J. Pei, and Y. Zhang, "Personalized cross-silo Federated Learning on non-iid data," in *The AAAI Conference on Artificial Intelligence*, vol. 35, pp. 7865–7873, 2021.

[57] M. Tang and V. W. Wong, "An incentive mechanism for cross-silo Federated Learning: A public goods perspective," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pp. 1–10, IEEE, 2021.

[58] H. Fereidooni, A. Dmitrienko, P. Rieger, M. Miettinen, A.-R. Sadeghi, and F. Madlener, "Fedcri: Federated mobile cyber-risk intelligence," in *Network and Distributed Systems Security (NDSS) Symposium*, 2022.

[59] K. Hsieh, A. Phanishayee, O. Mutlu, and P. Gibbons, "The non-iid data quagmire of decentralized machine learning," in *International Conference on Machine Learning*, pp. 4387–4398, PMLR, 2020.

[60] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh, "Scaffold: Stochastic controlled averaging for Federated Learning," in *International Conference on Machine Learning*, pp. 5132–5143, PMLR, 2020.

[61] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[62] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-Robust distributed learning: Towards optimal statistical rates," in *The 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 5650–5659, PMLR, 10–15 Jul 2018.

[63] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

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

[65] Y. Li, Y. Jiang, Z. Li, and S.-T. Xia, "Backdoor learning:A survey," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–18, 2022.

[66] Y. Gao, B. G. Doan, Z. Zhang, S. Ma, J. Zhang, A. Fu, S. Nepal, and H. Kim, "Backdoor attacks and countermeasures on deep learning: A comprehensive review," *arXiv preprint arXiv:2007.10760*, 2020.

[67] T. Gu, B. Dolan-Gavitt, and S. BadNets, "Identifying vulnerabilities in the machine learning model supply chain," in *The Neural Information Processing Symposium Workshop Mach. Learning Security (MLSec)*, pp. 1–5, 2017.

[68] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks,"

[69] H. Wang, K. Sreenivasan, S. Rajput, H. Vishwakarma, S. Agarwal, J.-y. Sohn, K. Lee, and D. Papailiopoulos, "Attack of the tails: Yes, you really can backdoor Federated Learning," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 16070–16084, Curran Associates, Inc., 2020.

[70] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[71] A. Saha, A. Subramanya, and H. Pirsiavash, "Hidden trigger backdoor attacks," in *The AAAI conference on artificial intelligence*, vol. 34, pp. 11957–11965, 2020.

[72] A. Turner, D. Tsipras, and A. Madry, "Label-consistent backdoor attacks," *arXiv preprint arXiv:1912.02771*, 2019.

[73] P. Rieger, T. D. Nguyen, M. Miettinen, and A.-R. Sadeghi, "DeepSight: Mitigating backdoor attacks in Federated Learning through deep model inspection," 2022.

[74] C. Xie, K. Huang, P.-Y. Chen, and B. Li, "DBA: Distributed backdoor attacks against Federated Learning," in *International Conference on Learning Representations*, 2020.

[75] T. D. Nguyen, P. Rieger, M. Miettinen, and A.-R. Sadeghi, "Poisoning attacks on Federated Learning-based IoT intrusion detection system," 2020.

[76] S. Shen, S. Tople, and P. Saxena, "Auror: Defending against poisoning attacks in collaborative deep learning systems," in *The 32nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), p. 508–519, Association for Computing Machinery, 2016.

[77] C. Fung, C. J. Yoon, and I. Beschastnikh, "Mitigating sybils in Federated Learning poisoning," *arXiv preprint arXiv:1808.04866*, 2018.

[78] L. Zhao, S. Hu, Q. Wang, J. Jiang, C. Shen, X. Luo, and P. Hu, "Shielding collaborative learning: Mitigating poisoning attacks through client-side detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2029–2041, 2020.

[79] T. D. Nguyen, P. Rieger, H. Chen, H. Yalame, H. Möllering, H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, S. Zeitouni, F. Koushanfar, A.-R. Sadeghi, and T. Schneider, "FLAME: Taming backdoors in Federated Learning," in *31st USENIX Security Symposium (USENIX Security 22)*, (Boston, MA), pp. 1415–1432, USENIX Association, Aug. 2022.

[80] R. J. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II 17*, pp. 160–172, Springer, 2013.

[81] P. Rieger, T. Krauß, M. Miettinen, A. Dmitrienko, and A.-R. Sadeghi, "Close the gate: Detecting backdoored models in Federated Learning based on client-side deep layer output analysis," *arXiv preprint arXiv:2210.07714*, 2022.

[82] P. Zhao, P.-Y. Chen, P. Das, K. N. Ramamurthy, and X. Lin, "Bridging mode connectivity in loss landscapes and adversarial robustness," *arXiv preprint arXiv:2005.00060*, 2020.

[83] T. Garipov, P. Izmailov, D. Podoprikhin, D. P. Vetrov, and A. G. Wilson, "Loss surfaces, mode connectivity, and fast ensembling of DNNs," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[84] N. Chatzipanagiotis, D. Dentcheva, and M. M. Zavlanos, "An Augmented Lagrangian method for distributed optimization," *Mathematical Programming*, vol. 152, pp. 405–434, 2015.

[85] S. Sangalli, E. Erdil, A. Hötker, O. Donati, and E. Konukoglu, "Constrained optimization to train neural networks on critical and under-represented classes," *Advances in Neural Information Processing Systems*, vol. 34, pp. 25400–25411, 2021.

[86] J. Rony, E. Granger, M. Pedersoli, and I. Ben Ayed, "Augmented Lagrangian adversarial attacks," in *The IEEE/CVF International Conference on Computer Vision*, pp. 7738–7747, 2021.

[87] F. Mu, Y. Liang, and Y. Li, "Gradients as features for deep representation learning," *arXiv preprint arXiv:2004.05529*, 2020.

[88] C.-H. Yuan and S.-H. Wu, "Neural tangent generalization attacks," in *International Conference on Machine Learning*, pp. 12230–12240, PMLR, 2021.

[89] J. Hayase and S. Oh, "Few-shot backdoor attacks via Neural Tangent Kernels," *arXiv preprint arXiv:2210.05929*, 2022.

[90] "Wikipedia - Lagrange multiplier." `https://en.wikipedia.org/wiki/Lagrange_multiplier`. Accessed: May 2023.

[91] M. Saeed, "A gentle introduction to method of Lagrange multipliers." `https://machinelearningmastery.com/a-gentle-introduction-to-method-of-lagrange-multipliers/`. Accessed: May 2023.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[93] "Wikipedia - Vanishing gradient problem." `https://en.wikipedia.org/wiki/Vanishing_gradient_problem`. Accessed: May 2023.

[94] M. J. Powell, "A method for nonlinear constraints in minimization problems," *Optimization*, pp. 283–298, 1969.

[95] H. Kuhn and A. Tucker, "Nonlinear programming," in *Second Berkeley Symposium on Mathematical Statistics and Probability*, pp. 481–492, 1951.

[96] W. Karush, *Minima of functions of several variables with inequalities as side conditions*. PhD thesis, The University of Chicago, 1939.

[97] H.-C. H. Ryan Tibshirani, Jayanth Krishna Mogali, "Lecture 12: KKT conditions." `https://www.stat.cmu.edu/~ryantibs/convexopt-F16/scribes/kkt-scribed.pdf`. Accessed: May 2023.

[98] "TensorFlow." `https://www.tensorflow.org/`. Accessed: May 2023.

[99] "About CUDA - NVIDIA developer." `https://developer.nvidia.com/about-cuda`. Accessed: May 2023.

[100] "The MNIST database of handwritten digits." `http://yann.lecun.com/exdb/mnist/`. Accessed: May 2023.

[101] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *IEEE International Joint Conference on Neural Networks*, pp. 1453–1460, 2011.

[102] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[103] "Flaticon.com." `https://www.flaticon.com/`. Accessed: May 2023.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, May 15, 2023

.....

(Jan König)