Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Constrained Learning for Improved Attack Resilience in Federated Learning

Tobias Fella
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
First Reviewer

Torsten Krauß
First Advisor

Prof. Dr. Samuel Kounev
Second Reviewer

Submission
4. September 2023

www.uni-wuerzburg.de

Abstract

Federated Learning is an approach to machine learning where several participants train models locally, followed by aggregating these models into a combined model. This approach has several advantages over classic, centralized machine learning, e.g., that the training data does not have to leave the client's devices. Defenses against poisoning attacks on Federated Learning are an area of active research. Most existing approaches detect malicious models by focusing only on the adversarial side. This thesis goes a different way, by proposing changes to benign training on the client side, which lead to improved detection capabilities of existing defense mechanisms. We do this by constraining certain model characteristics, so-called metrics, in benign learning. This causes the benign models to be more similar to each other within a certain metric that is leveraged by the defense mechanism. The central parts of our research are an evaluation of how to best constrain malicious learning, evaluating, among others, techniques from Multi-Task Learning. We then evaluate the system's capability of detecting malicious models and other effects it has on the behavior of the Federated Learning environment.

Zusammenfassung

Federated Learning ist eine Variante des maschinellen Lernens, bei der mehrere Geräte lokal Modelle trainieren, die dann zu einem neuen Modell zusammengefasst werden. Dieser Ansatz hat mehrere Vorteile gegenüber zentralisiertem maschinellem Lernen, unter anderem, dass die Trainingsdaten das lokale Gerät nicht verlassen müssen. Verteidigungssysteme gegen Poisoning Attacken im Federated Learning sind ein Gebiet, in dem aktiv geforscht wird. Die meisten bisherigen Systeme fokusieren sich darauf, bei unverändertem Training bösartige Modelle zu finden, indem alle Modelle auf bestimmte Eigenschaften untersucht werden. Diese Arbeit ändert das, indem sie ein neuartiges System vorschlägt, bei dem das Training von allen, bösartigen und gutartigen, Systemen geändert wird, um die Erkennungsfähigkeit von Verteidigungen zu verbessern. Dies wird erreicht, indem das Training in bestimmten Metriken eingeschränkt wird, mit dem Ziel, dass die gutartigen Modelle ähnlicher sind und damit die bösartigen Modelle leichter zu erkennen sind. Die zentralen Teile dieser Arbeit sind die Entwicklung des bestmöglichen Ansatzes, die Metriken im Modell einzuschränken und eine Untersuchung darauf, ob das System geeignet ist, Angriffe besser zu erkennen und welche Effekte das System sonst auf das Federated Learning hat.

Contents

1 Introduction **1**

2 Background **5**
2.1 Federated Learning . . . 5
2.2 Adversarial Federated Learning . . . 7
2.3 Backdoor Attack Types . . . 8
2.4 Multi-Task Learning . . . 9
2.5 Stochastic Gradient Descent with Momentum . . . 10
2.6 Stochastic Gradient Descent with Weight Decay . . . 10
2.7 Federated Learning Defense Metrics . . . 11

3 Related Work **13**
3.1 Defenses against Adversarial Federated Learning . . . 13
3.1.1 Detection and Filtering Approaches . . . 13
3.1.2 Influence Reduction Approaches . . . 14
3.2 Multi-Objective Optimization . . . 15

4 Approach **17**
4.1 Threat Model . . . 17
4.2 High-Level Overview . . . 17
4.3 Research Questions . . . 19
4.4 Constraining Metrics . . . 19
4.5 Experiments . . . 19
4.5.1 Baseline Measurements . . . 20
4.5.2 Determining the Best Strategy for Constraining Metrics . . . 20
4.5.3 Effectiveness of the Constraint System . . . 20
4.5.4 Different Setups . . . 21

5 Implementation **23**

6 Evaluation **27**
6.1 Experimental Setup . . . 27
6.2 Baseline Measurements . . . 27
6.2.1 Unconstrained Backdoor Accuracy . . . 28
6.2.2 Natural Metric Values . . . 30
6.2.3 Number of Epochs to Train . . . 31
6.3 Determining the Best Strategy for Constraining Metrics . . . 33
6.3.1 Weighting Method . . . 34
6.3.2 Weighting Approach . . . 37
6.3.3 Alpha . . . 39
6.3.4 Constraint Learning Rate . . . 42
6.3.5 Bagdasaryan Training or Individual Optimization . . . 44

6.4 Effectiveness of the Constraint System 47
6.4.1 Training Time Required for Constraints 50
6.4.2 Resilience of Backdoors Trained Under Constraints 51
6.5 Different Setups 52
6.5.1 Distributions 52
6.5.2 Learning Rate 53
6.5.3 Batch Size 55
6.5.4 Optimizer 57
6.5.5 Target Values 59
6.5.6 Momentum 61
6.5.7 Datasets 64
6.5.8 Models 65
6.6 Summary 66

7 Discussion **69**
7.1 Per-Layer Constraints 69
7.2 Other Areas of Machine Learning 70
7.3 Comparison with Other Approaches 70
7.4 Choice of Metrics 70
7.5 Improving Training Time 70

8 Conclusion **73**

List of Figures **75**

List of Tables **78**

Listings **79**

Acronyms **83**

Bibliography **85**

1. Introduction

Machine learning (ML) has the potential to revolutionize many areas of industry, research, and everyday life. Self-driving cars [1][2][3], medical image recognition [4][5][6], and voice assistants [7][8][9] are just some examples of applications that make extensive use of ML to bring the user significantly improved functionality. For some of these applications, implementing them using rule-based programs would be prohibitively complicated. Here, the approach of extracting knowledge from data, as used by ML, is what enables the applications to function.

To create a Neural Network (NN) for a certain task, large amounts of data are typically needed, which is then used to train the model utilizing a significant amount of computing power, specialized hardware like GPU accelerators, and electrical energy [10]. This makes ML a difficult and expensive task. Difficult, since gaining and preprocessing the required amount of training data is often a complicated and slow process that typically requires manual interaction of humans or engineers with every sample in the dataset, and expensive, since computing power and electrical energy are both not cheap.

In a traditional ML environment, all training data is collected in a single training system, where it is then processed. However, there are many scenarios where this approach has some drawbacks: When the data is collected directly from distributed systems, sending it to a central system could cause a large communication overhead [11], e.g., when dealing with a large number of high-resolution images. Having to send the data to a central trusted server controlled by a third party could also be harmful for privacy reasons, e.g., when multiple hospitals strive to train a collective classification system in the medical image processing domain [12]. In such a situation, without additional security considerations in the system setup, one cannot guarantee privacy, and therefore, ensure compliance with legal requirements, like the GDPR in the European Union [13] and the CCPA in California [14][12].

To enable applications where ML can make use of private data from multiple clients without sharing the plain data, *Federated learning (FL)* [15] has been introduced. In FL, the training itself happens on the client side of the devices participating in the federated system, while the central server aggregates and distributes the model updates after the local training by each device [15], i.e., by averaging the updates of all participating clients [15]. This means that in FL, the training data itself does not need to leave the individual device, so the central server and all other devices in the federation do not have direct access to the data. This significantly improves privacy around centralized ML. Also, since the central

server does not perform any ML itself, it does not need huge amounts of computational power and energy, significantly reducing the cost of operation for the operator of the server.

While FL solves many problems around classic *ML*, it also raises new challenges. In particular, it was shown that attacks on security and privacy are still possible. Especially, dealing with so-called *poisoning attacks* that can affect ML is more challenging in FL settings.

Poisoning attacks can be differentiated into *targeted* and *untargeted* poisoning attacks. In an untargeted poisoning attack, the goal of the adversary (i.e., the attacker participating in the FL network) is to reduce the usefulness of the model by lowering its prediction performance, while in a targeted poisoning attack, the goal is to cause the model to output a specific miss-prediction for a certain trigger in the input data [16]. Targeted poisoning attacks are also known as *backdoor attacks* in literature [17]. To conduct such attacks, the adversary needs to create malicious training data, for example, by intentionally mislabelling some of the data samples. This data then needs to be used for training by the central server. Since the server has access to the data, the attack can easily be detected by analyzing the input data. In FL, since the server does not have any direct knowledge of the user's training data, detecting poisoning attacks is much more difficult. This means that an attacker can perform poisoning attacks on the model he trains locally, which will then be part of the aggregated model distributed to other devices. Since the adversary also has the direct ability to change his own model completely (known as *Model Replacement* [17]), he can make this attack even more effective. By scaling the local model update, the adversary can increase the influence of his own contribution in comparison to other clients, increasing the probability that the backdoor is effective after his local model update has been aggregated with the model updates of the other participants. It has been shown that this approach can be used to create very effective backdoors that will survive in the aggregated model for multiple rounds of training, even when the backdoor itself is not renewed every training round [17].

These backdoors are hard to detect, as a detection mechanism built into the central server only has access to the attacker's model and not the data used to train that model. The client's local data are often not identically and independently distributed (non-IID), e.g., pictures of road signs in different countries can look significantly different, or one client can have mainly images from one class, e.g., stop signs, while another client possesses mainly images from another class, e.g, speed limit signs [18]. Since the data is not always IID, the model resulting from the training on this data can also differ significantly, reflecting the underlying data distribution. This means that even for benign local models, the individual updates can have significant differences, complicating naïve detection mechanisms based on outlier detection. Some defenses have been proposed to be able to detect malicious models [16][19][20], but, when the adversary knows about the characteristics that the defense evaluates, the so-called metrics, he can train the model to adapt to that metric. This adaption process has been shown to not result in a significantly reduced model performance [17]. The defense mechanisms also suffer from high false-positive rates.

One approach to improving backdoor detection is to constrain the local benign models in multiple metrics. The intuition is, that constrained updates enable a more robust detection for existing filtering approaches, e.g., the federation has less benign outliers. In ML, the training is guaranteed to reach a local minimum instead of a global minimum. This means that, intuitively, the constraints do not prevent the model from reaching a good performance, since it is still able to reach a local minimum. To achieve improvements in detection, the server must ignore all model updates that do not conform to the constraints. This is likely to force an adversary to train malicious models with respect to the constraints of the federated system. This results in multiple objectives being relevant to the adversary:

The main task for the purpose of producing a stealthy model, the backdoor task necessary to introduce the malicious behavior, and the constraints to be stealthy in the defense metrics. Ideally, this forces the adversary into a dilemma, where one needs to choose between training a backdoor while lowering model performance or training the backdoor with high model performance but under the risk of being detected due to the model deviating from the constraints. Determining that constraints on benign FL result in a safer FL environment is the central research question of this thesis.

Additionally, since such a constraint system is a significant change in the ML setup that every participant has to respect, it has a significant impact even on benign devices in the federated system. Researching these impacts is another important part of our research. Some central aspects of the ML operation are the performance reached by the NN and the amount of training required to achieve this. For the model's performance, we show that the constraints increase the achieved accuracy.

Our research thus focuses on the following aspects: First, the thesis shows that ML models can be constrained without worsening the performance of the model and which approaches work best for achieving this in Sections 4.5.2 and 6.3. Here, we evaluate approaches from adversarial ML, and from Multi-Task Learning (MTL). Next, in Sections 4.5.3 and 6.4, we show how this can be beneficial for detection of malicious models. Finally, Sections 4.5.4 and 6.5 show the influence that different configurations have on the quality of the adaption to the constraints. Section 6.5.6 shows how momentum can be used to lead the adversary into a strong dilemma, which prevents the injection of a strong backdoor into the global model.

2. Background

This chapter explains the background required for our work. We start with an overview of FL in Sect. 2.1, followed by an explanation of adversarial FL in Sect. 2.2 and types of backdoor attacks in Sect. 2.3. We're concluding with a short introduction to MTL and Multi-Objective Optimization (MOO) in Sect. 2.4.

2.1 Federated Learning

FL is an approach to ML that tries to solve many of the problems that arise due to the centralization of classical ML. These problems are:

- Large resource consumption by the central ML server, both in terms of required hardware and energy consumption [10],
- Requirement for training data to be available to the central server, which often means large amounts of required storage and a large communication overhead,
- Privacy aspects often make centralized training impossible.

FL improves on these problems by performing the training on the client-side, instead of on a centralized server [15]. As training data, the devices use their local data, for example in the form of sensor data of a mobile phone [21], cameras in a self-driving car [22], or medical images produced by a hospital [12]. This means that the training data itself does not leave the owner's device; instead, only the trained model (or the respective model update) is sent to a server. The central server then aggregates the model updates with the updates trained by the other participants and sends the aggregated model to all participants in the federation. The standard aggregation algorithm is *Federated Averaging (FedAVG)* [15], but there are byzantine robust alternatives like *KRUM* [23], *Median* [24] and *trimmed-Mean* [25]. In our research, we use the FedAVG algorithm, since it is most commonly used in research and thus allows for comparability with other approaches [16][26][19]. For the operator of the federation, this approach has the advantage that orchestrating the aggregation server requires significantly fewer resources than an ML server, while the other participants in the federation each typically only see an insignificant increase in resource usage. For a client device in the network, a significant advantage is that its data can be used in training (which can often mean a significant improvement in accuracy for this specific device) without having to make its training data known to any other party [15].

Depending on the number of clients, FL setups can be categorized as *cross-silo*, where a small number of clients provide a large amount of data, e.g., in a hospital scenario [12], or *cross-client*, where many clients each have a smaller amount of data, e.g., in the case of self-driving cars [1][27].

To evaluate the performance of a model, its capability of predicting the correct output for a given input is required. This value is called *Accuracy* of a NN and is defined as the probability of a random sample being classified correctly by the NN. The accuracy can be determined empirically by counting the number of correct predictions for a test data set and dividing it by the total number of samples in the test data set. During adversarial training, the accuracy achieved for the backdoor is called *Backdoor Accuracy* (BA), while the accuracy for the main task is called *Main-Task Accuracy* (MA).

Equation 2.1 shows the FedAVG algorithm with G^t as the global model in FL round t and L_i^t as the local model of device i in round t [16]. The algorithm works by adding the average of the individual model updates to the global model of the previous round, after weighting the average with the global learning rate δ [16].

$$G^{t+1} = G^t + \delta(\frac{1}{n}\sum_{i=0}^{n-1}(L_i^{t+1} - G^t)) \tag{2.1}$$

FL is round-based. At the beginning of each round, the server selects a random subset (of a specific size) of the devices in the federation. This subset then receives the current global model and each device trains that model with its local data. After the round of training is finished, the model updates are sent back to the server, which then computes a new global model from the local updates using an algorithm like FedAVG [15]. Usually, the client updates are weighed by the size of their datasets. In FL security research, the updates are weighed identically, since it is not clear if adversaries participate in the federation. Such adversaries could report false sizes, giving them the ability to increase the weight that is given to their contributions.

In FL, the distribution of the data in the client's local datasets is important, since a model always reflects the underlying data distribution in a classification task [1]. If the samples are *independent* from each other (i.e., the outcome of sampling from a dataset does not influence the outcome of the next sampling) and the samples follow the same distribution regarding their class label, the data is *identically and independently distributed (IID).* The opposite state, where the samples are not independent, don't follow the same distribution, or are disjoint, which means that some label classes are completely missing on some clients, is called *non-IID.* If the samples in a local dataset are IID between all of the available classes, the trained model is more likely to be able to classify samples of all classes with similar performance. If the clients in FL possess non-IID local datasets, the different models are specialized for specific label classes but might perform poorly on others. The aggregation transfers the knowledge of the specialized clients to a new global model and with this to the other clients. Further, non-IID can be defined between clients. Each of the clients can have a different sample distribution, e.g., one client can possess samples of 10 classes while another one has samples of just two classes or both possess the same labels, but the sample distribution differs. Such non-IID scenarios, within clients and between clients, are challenging problems for backdoor detection systems, especially, if the approaches rely on metrics that reflect the underlying data distribution and are based on majority assumptions, e.g., that 50% of the clients in a federation are benign. This is due to the difficulty of differentiating between models from disjoint datasets with non-IID sample distributions and adversarial models from poisoned datasets.

[1]We only consider classification tasks for the rest of the thesis.

In research scenarios, standardized datasets are commonly used as training data for ML. This ensures better comparability of the results gained in the research. For image classification, common datasets include:

- **MNIST** (Modified National Institute of Standards and Technology) Database, which is a dataset published by the US NIST, consisting of images of hand-written digits. Each sample consists of 28x28 grayscale pixels. In total, the dataset contains 60,000 training samples and 10,000 test samples [28].
- **CIFAR10** (Canadian Institute for Advanced Research) is a dataset containing images 32x32 pixel images from 10 different classes. For each class, there are 6,000 images, for a total of 60000 images [29]. There also exists an extended version called **CIFAR100**, containing images from 100 classes [29].
- **GTSRB** (German Traffic Sign Recognition Benchmark) is a dataset with roughly 50,000 images from more than 40 classes of German road signs [30].

In this thesis, we consider a FL system that classifies images from the CIFAR10 [29] dataset. For evaluating the system against other datasets, we also perform experiments with MNIST [28].

2.2 Adversarial Federated Learning

Adversarial federated learning is FL with an adversarial goal. In a *targeted poisoning attack*, the adversary wants to manipulate a NN in a way that makes specific inputs lead to wrong, predefined outputs. An adversary can inject these so-called *backdoors* into a NN to achieve a certain goal, e.g., to inject advertisements into the next word prediction of a smartphone's keyboard [17]. An *untargeted poisoning attack* is an attack with the goal of decreasing the global model's accuracy.

In non-federated ML, the only entity that can influence the NN training is the central ML server since the devices using the NN do not have a way of distributing their (backdoored) models to other devices. The only attack vectors for an adversary to introduce a backdoor into this setup are thus to compromise the central server or to manipulate the input data. In FL, every device can influence the NN trained in the federation. This means that an adversary only needs control over one of the devices to introduce backdoors into the NN. Additionally, since the devices do not share their individual training data with other devices, the backdoors can only be detected through analysis of the model updates. This results in adversarial ML being a significant risk in FL [17].

There are multiple approaches an attacker can use to add a backdoor to an existing ML model. The basic approach is to prepare training data to include the desired backdoors and train the model using an otherwise normal training process. To speed up the backdoor training, the adversary can increase the learning rate for the malicious samples. Since the adversary has full control over his local model, he can also replace this model entirely. This can be used to ensure that the backdoor remains intact after the aggregation algorithm created the new global model [17].

Defense mechanisms against adversarial FL can be roughly categorized by the used approach. Detection and Filtering (DF) defenses work by using a metric to find adversarial model updates and excluding those updates from the aggregation process. Influence Reduction (IR) methods try to remove adversarial behavior by limiting the influence a local model has on the new global model [16].

Figure 2.1 shows a possible defense setup. The model updates are first passed through a DF defense, followed by a local IR layer, the aggregation, and a global IR layer. Due to

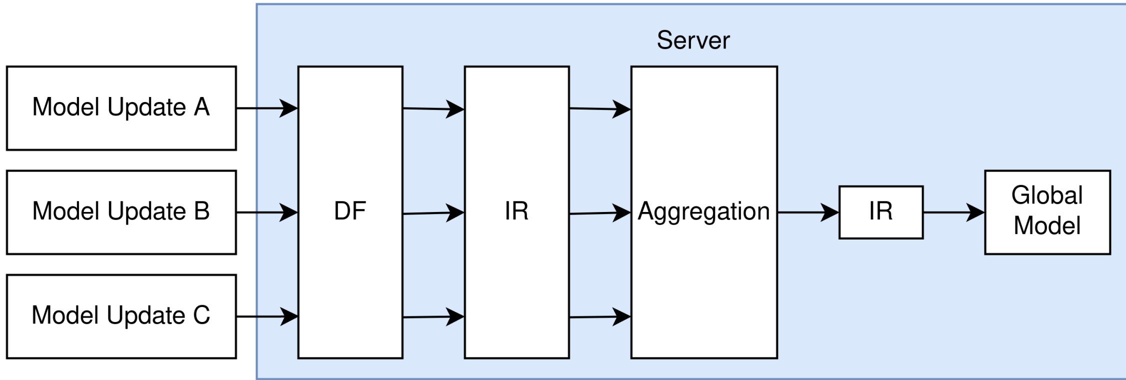

Figure 2.1: A possible defense setup using both DF and IR mechanisms.

the adversarial dilemma, the combination of IR and DF mechanisms is promising to be effective in filtering out many backdoors.

When training a backdoor into a model, an attacker can choose how aggressively he wants to train the model. When choosing an aggressive approach (e.g., a high learning rate), the adversarial model is more likely to differ significantly from the benign models, meaning that a DF mechanism based on outlier detection can find the backdoor easier [20]. If the attacker chooses a less aggressive approach to train a backdoor into the model, this backdoor is more likely to not be very effective, which means that an impact-reducing defense mechanism can prevent the backdoor from being effective at all. When using a defense mechanism that combines a detect-and-filter approach with an impact-reducing approach, the defender should thus be able to prevent many backdoors in the model. This means that an attacker has no clear way of performing a backdoor attack that will not be prevented through the use of a defense mechanism. This problem (to the adversary) is called *adversarial dilemma* [20].

An adversary can circumvent a DF defense mechanism by training the model against this mechanism. One possible approach to do this is the *constrain-and-scale* method [17]. This approach works by first training the model using MTL, with the individual tasks being the normal training of the model with the backdoor and the adherence to the metric enforced by the defense mechanism. The adversary can train on these tasks simultaneously by using a weighted linear combination of the individual task's loss functions as seen in Equation 2.2, where $\mathcal{L}_{class}$ is the loss for the underlying data, i.e., the main task and the backdoor task, and $\mathcal{L}_{defense}$ is the loss for adherence to the defense mechanism [17]. Then, to ensure that the trained model updates are not canceled out during aggregation, the model updates are scaled up by a hyper-parameter before sending them to the server. The scaling factor must be chosen to be large enough for the backdoor to remain effective while being low enough to not trigger a detection mechanism [17]. In this thesis, we show that this approach to constraining the model (called *Bagdasaryan Training* in the thesis) works well, by comparing it to a different approach. We also determine exactly how Bagdasaryan Training should be performed to lead to the best results.

$$\mathcal{L}_{model} = \alpha\mathcal{L}_{class} + (1-\alpha)\mathcal{L}_{defense} \tag{2.2}$$

2.3 Backdoor Attack Types

Backdoors attacks can be split into two categories. In the first category, the adversary only needs to manipulate the labels, and the data itself remains unchanged. These methods

include:

- *Label swap backdoors*, where all samples of one class are mislabelled to an incorrect class [16],
- *Semantic backdoors*, where images with certain features are mislabelled as a different class, e.g., classify green cars and cars in front of vertically striped walls as dogs [17],
- *Edge-Case backdoors*, where samples with very unlikely features (edge-cases) are mislabelled as another class. Because of their rarity in the training data, these samples would typically be mislabelled even in benign training [31].

In the second category, triggering the backdoor requires the adversary to change both the labels and the data itself. This category includes:

- *Pixel-Pattern backdoors*, where images containing certain patterns of pixels, e.g., an entirely red pixel in each corner of the image, are mislabeled [17]. In some cases, the pattern does not consist of a small number of pixels, but a smaller image, like a sticker, embedded in the image [32],
- *Distributed backdoor attacks*, where the pattern triggering the backdoor is split into multiple parts, with multiple adversarial clients training the backdoor, each being responsible for a specific part [33].

For this thesis, we only consider Pixel-Pattern backdoors. Evaluation of other backdoor types would have to be performed as part of future work.

2.4 Multi-Task Learning

MTL is a variant of ML, where a single model is trained to achieve good results in multiple tasks [34]. MTL can lead to improved results since it allows for the reuse of parts of the models that are not specific to the individual task. This means that when training for one of the tasks, the accuracies for other tasks also improve since the common parts of the NN are trained. For example, for image recognition, the first parts (layers) of the NN typically detect general structures in the images, while the last layers use this structural information to perform task-specific calculations. When using MTL here, the first layers are shared between the individual tasks. This means an increase in training data for these layers, resulting in better accuracies for each individual task [34]. Figure 2.2 shows an example of a MTL network with three shared layers, followed by individual layers for each task. When training an MTL model for a specific task, the loss for this task is back-propagated through the network, where the task-specific parameters for this task are adjusted first, which then propagates further back to the shared layers [34]. In our research, we consider constrained ML an MTL problem, with the main-task training and each constraint as an individual task.

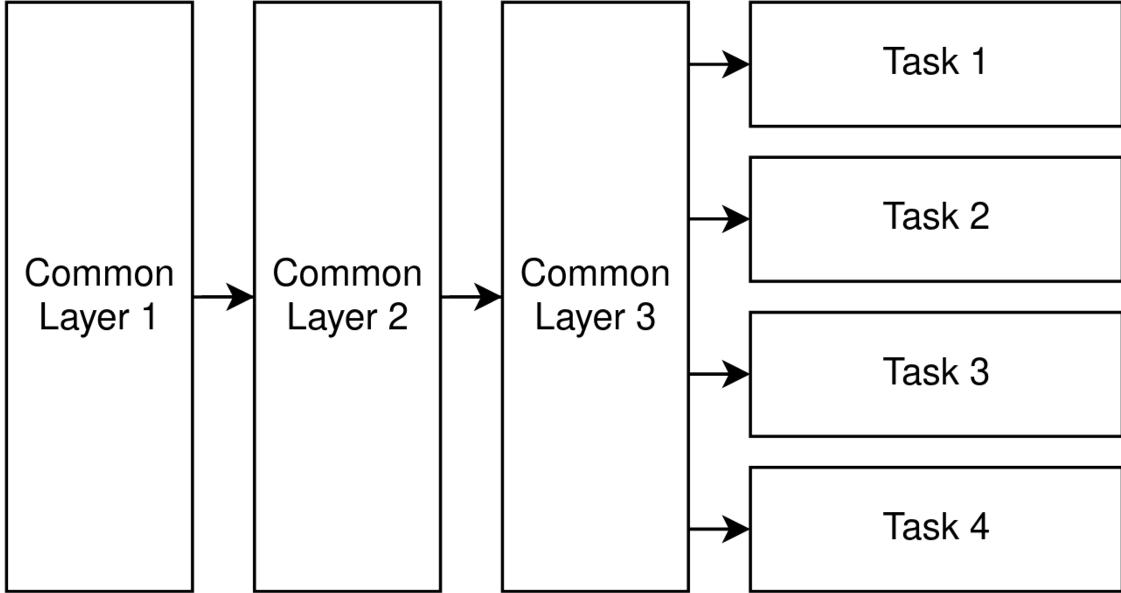

Figure 2.2: A basic MTL setup. The first 3 layers of the NN are shared between the tasks, while each task also has an individual layer

2.5 Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent (SGD) is an optimization algorithm that is often used when training machine learning models. SGD works by approximating the gradient of the loss function using the gradient at a small number of samples and using this approximated gradient to calculate the new weights for the model.

Momentum extends SGD by adding the gradient from the previous iteration to the current gradient, weighted by a factor μ. It is often used in machine learning to achieve faster improvements. When using momentum, the update from the previous iteration (Δw) is remembered and applied to the next round as shown in Formula 2.3, where w are the weights of the NN, Q_i is the loss function for the i-th sample, ∇Q_i is the gradient of the loss function, and η is the learning rate [35][36]. First, the new weight update is calculated as the difference between the previous weight update and the gradient of the loss function (weighted with their respective weight factors μ and η). This value is then added to the weights to calculate the new weights and stored for the next iteration of the optimization algorithm. Bagdasaryan et al. [17] uses momentum for all training, without evaluating its necessity first. In this thesis, we evaluate the influence it has on benign and adversarial training with constraints.

$$\begin{aligned} \Delta w &= \mu \Delta w - \eta \nabla Q_i(w) \\ w &= w + \Delta w \end{aligned} \tag{2.3}$$

2.6 Stochastic Gradient Descent with Weight Decay

Weight Decay is an extension to SGD that aims to keep the overall weights in the NN small. This is achieved by adding an additional factor to the loss function that is equal to the L2 norm of the weights [37], as shown in Formula 2.4. Here, Q_o is the original loss function, Q is the new loss function, λ is the factor determining the strength of the weight decay (with larger values leading to smaller weights), and w_i are the weights in the model.

Similarly to Momentum, Weight Decay is used in existing work on adversarial FL. In this thesis, we thus also evaluate the influence it has on the training of backdoors into ML models.

$$Q(w) = Q_o(w) + \frac{1}{2} * \lambda * \sum_i w_i^2 \tag{2.4}$$

2.7 Federated Learning Defense Metrics

Some defense mechanisms perform outlier detection on ML models, with the goal of detecting malicious models. To achieve this, the dimensionality of the models needs to be reduced. This can be done by extracting metric values from the models, thereby reducing the problem from many dimensions to the single dimension of the metric values. There are metrics that quantify relations between vectors. These can be used for analyzing the model parameters, which can be interpreted as vectors. For metrics that operate on multiple vectors, e.g., the cosine distance, a reference model is used as the second vector. In our evaluation, we use the metrics found in MESAS [38], since it's the defense mechanism that evaluates most metrics simultaneously. By using metrics from a real-world defense mechanism, we ensure that our system applies well to existing mechanisms.

MESAS [38] considers these metrics:

- Cos: The cosine of the angle between two vectors.
- Euclid: The Euclidean distance between the vectors.
- Var: The variance of the elements of the vector.
- Count: The number of parameter values increased from the global model.
- Min: The smallest element in the vector.
- Max: The largest element in the vector.

3. Related Work

In this thesis, we evaluate how existing defenses against adversarial federated learning can be improved. To do this, we're using an approach based on MTL. The related work relevant for us thus comes from these two areas. We start this chapter with an overview of some existing defense mechanisms in Sect. 3.1, followed by a look at work on how MOO can be used in MTL in Sect. 3.2.

3.1 Defenses against Adversarial Federated Learning

Many approaches for defending against poisoning attacks in FL have been proposed. Table 3.1 gives an overview of the metrics used by the defense mechanisms.

Table 3.1 The presented defense mechanisms and the metrics they use to detect backdoors.

Defense Mechanism	Detection Metrics
CrowdGuard [16]	HLBIM (COS, EUCLID)
DeepSight [20]	DDifs, NEUPs, Threshold exceedings
FoolsGold [39]	Cosine similarity (COS)
Auror [19]	K-Means Clustering
Adaptive Federated Averaging [40]	Cosine similarity (COS)
(Multi-)KRUM [23]	Majority-Based Squared-Distance
BaFFLe [41]	Client Feedback
MESAS [38]	COS, EUCLID, VAR, MIN, MAX, COUNT

3.1.1 Detection and Filtering Approaches

DeepSight [20] is a DF mechanism that introduces three new metrics for finding adversarial models. The metrics used include *Division Differences (DDifs)*, which measure the differences between the predicted scores of the local and global models [20], *NEUPs*, which measure the magnitude of the updates of the neurons, and *Threshold exceedings*, which measure the homogeneity of the labels in the training data [20]. One problem of DeepSight is that especially the DDifs assume that adversarial models were trained with fewer labels than benign models [20]. Additionally, as for most papers, non-IID data between multiple clients is not supported [20].

Neural Cleanse [42] is a method intended to not only find backdoors in a model but also be able to "patch" the backdoor out of the model. This can be useful if removing malicious models is infeasible, e.g., when it is not possible to use a different model or when removing a model update would have a significant impact on the global model's accuracy, like in cross-silo FL [42]. In cross-client FL, removing a single model update is assumed to not have a significant negative impact on the accuracy of the network. Neural Cleanse has the disadvantage that it can only find simple, pixel-pattern backdoors in networks with a small number of classes, which means that it is not useful for many real-world tasks [17].

CrowdGuard [16] is a mechanism that uses the other client's training data to evaluate the maliciousness of a participant's individually trained network. While this requires sending locally trained networks to other participants, which is generally not desirable due to privacy concerns, this is mitigated by using secure enclaves to guarantee that the devices do not use the models for anything but the intended purpose [16]. A new metric called *HLBIM* is introduced to detect backdoors based on the DNN's hidden layers [16].

A similar approach is taken by BaFFLe [41]. Using the data that is available client-side, a feedback mechanism is created, i.e., a system where the clients evaluate whether model updates are malicious. In contrast to CrowdGuard, it only evaluates the presence of a backdoor based on the last layer's output, whereas CrowdGuard also considers the hidden layer outputs [41][16].

Zhao et al. [43] also follows this approach, with clients inspecting model updates created by other clients. In contrast to CrowdGuard, it only considers the model's performance on the local data, i.e., it only uses the output data. This means that it is less powerful than CrowdGuard since an attacker can hide the backdoor in the intermediate layers [43][16].

FoolsGold [39] is a mechanism for detecting *Sybil attacks* in FL, meaning attacks where multiple devices in the network collude to achieve a common goal (i.e., insert a backdoor). It assumes that the models submitted by adversarial actors would look similar to one another, but different from non-adversarial model updates. Hence, the models are clustered by their distances to find the adversaries [39].

Auror [19] is an approach that works by assuming that benign models have a similar distribution to each other, while adversarial models differ in their distributions. It uses K-Means to cluster specific features of the models and determine whether those features are indicative of an adversarial model [19]. It has the drawback that it depends on multiple adversaries training towards the same backdoor, which is not necessarily the case in FL [19].

Adaptive Federated Averaging [40] is an approach that uses a new model averaging algorithm that dynamically scales the influence of an individual update by determining a relevance for the update based on a hidden Markov model and the cosine similarity [40]. While this approach is noteworthy for not completely discarding "dubious" model updates, the metric it uses is rather simple and the results will thus be limited if the adversary adapts to the respective metric [40]. This approach is a combination of both IR and DF approaches, i.e., it uses the detection from DF methods and the reduction from IR methods.

MESAS [38] uses 6 metrics to force the attacker into a situation where adaption to all constraints is infeasible. In our research, we use these metrics for constraining our models.

3.1.2 Influence Reduction Approaches

IR approaches include 2DP-FL [44], which works by adding noise to the individual model updates before aggregation and to the new global model after aggregation. Different configurations can be used depending on the desired trade-off [44].

McMahan et al. [45] introduces an approach that trains language models for improved security using stochastic gradient descent. The approach promises that it does not cause a decrease in accuracy, but an increase in training time [45]. When training for the same time as without this approach, the accuracy will be worse.

The central part of the FL concept is the mechanism used for combining the local models from the clients into a global model. In adversarial ML, this is critical, since models with backdoors should be excluded at this point. The mechanism can also influence convergence speed and the influence of individual updates.

Krum [23] focuses on improving byzantine resilience, i.e., resilience against multiple attackers colluding to introduce a backdoor. To achieve resistance against such attacks, the method chooses one reference model that is closest to its neighbors based on a combination of a *majority-based* and a *squared-distance* method [23]. The Multi-Krum variant combines Krum with averaging by interpolating between them, allowing to combine the resilience properties of Krum with the faster convergence of averaging [23].

Sun et al. [46] introduce *Decentralized Federated Averaging*, which removes the need for a centralized server for creating the new global model. While being able to remove this instance has many advantages, e.g., the resiliency of the federation against outages of the network, it also brings increased complexity to each client [46]. While we do not expect this change in averaging algorithm to have an impact on an attacker's ability to inject backdoors into a model, it could worsen defense mechanisms due to the complexity added to the system.

Many of these defense mechanisms could benefit from the constraint mechanism we aim to develop in this thesis, because of their reliance on the metrics listed in Table 3.1. We evaluate if benign clients can be constrained in some of these metrics (Sect. 4.5.3), to make existing defenses more effective (Sect. 6.5.6) and to harden the adaption process for the adversary (Sect. 4.5.3). When an adversary circumvents defense mechanisms by training to conform to the respective metrics, he must handle a MOO problem. Our research goes beyond this, by enforcing multiple constraints on the ML model even in benign training. We do not focus on an individual, specific defense but rather on the general approach of using multiple constraints and how it can improve the quality of outlier detection DF approaches. Unlike other defense methods, we do not come up with a new metric but come up with an approach such that the existing defense mechanisms would be improved, which has not been done before.

3.2 Multi-Objective Optimization

A central question for the constraint-learning approach is how the constraints can be enforced during training. The constraint system is a multi-objective optimization problem, so methods from MOO can be used here. Here, we use the terms "task" and "objective" interchangeably. A common approach here is to create a loss function from a weighted linear combination of the loss functions of each task. While there are algorithms, from research around gradient-based multi-object optimization [47], that achieve better results than the weighted linear combination, these are not generally suited to MTL, since the algorithms do not scale well with the number of dimensions and the number of tasks [48]. Sener et Koltun [48] introduces an algorithm that is suitable for usage in MTL scenarios that returns a *Pareto optimal solution*, i.e., a solution for which there is no different solution that has better results for **all** individual loss functions.

Our scenario is a special form of MTL since our models only contain shared layers and no task-specific layers. Usually, in MTL, there are individual outputs for each task, with each output being connected to layers specific to the respective task. In our case, the additional

tasks being trained on do not correspond to any output; instead, we use them only to enforce certain metrics in the model. This means evaluate whether research focusing on the architecture for MTL is adaptable to our use case.

4. Approach

In this chapter, we first present the threat model for our research in Sect. 4.1, followed by an overview of the approach we present in Sect. 4.2, the research questions we answer in this thesis in Sect. 4.3, the strategy we're using for constraining the metrics in Sect. 4.4, and the experiments conducted in order to answer our research questions in Sect. 4.5.

4.1 Threat Model

We consider a standard FL setup as described in Sect. 2.1. In this setup, the adversary controls fewer than half of the devices in the federation. We assume the attacker has full access to the devices, including privileged access. We further assume that the adversary does not have special access to the aggregation server, meaning that the server is trusted.

The goal of the adversary is to perform backdoor attacks on FL using data poisoning, malicious training, and model replacement attacks. For targeted poisoning attacks, the adversary tries to inject a backdoor leading to the misclassification of samples with a specific pixel-pattern trigger. For untargeted poisoning attacks, the goal of the attacker is to reduce the model's performance as much as possible.

For training, the adversary has access to more computing power than the benign devices in the federation. We generally assume the training data to be non-IID.

4.2 High-Level Overview

Our setup is based on a standard FL scenario, as introduced in Sect. 2.1. We extend this setup by constraining the training of benign models in metrics commonly used by DF mechanisms. Figure 4.1 shows an example of such a system in two dimensions: While the values in the left figure are not constrained in any way, the right figure limits values in their angle between the x1 and x2 axis. When enforcing such constraints, the model updates will be more similar under the metrics that were constrained. Due to these constraints, the influence of a single model on the global model lowers, causing adversarial models with a high influence to be more easily detectable. This leads to a situation where the adversarial models cannot adapt to the constraints as well as the benign models, meaning that the metric values for the adversary would differ from those for the benign models, causing the defense mechanism to be able to filter out the adversarial models.

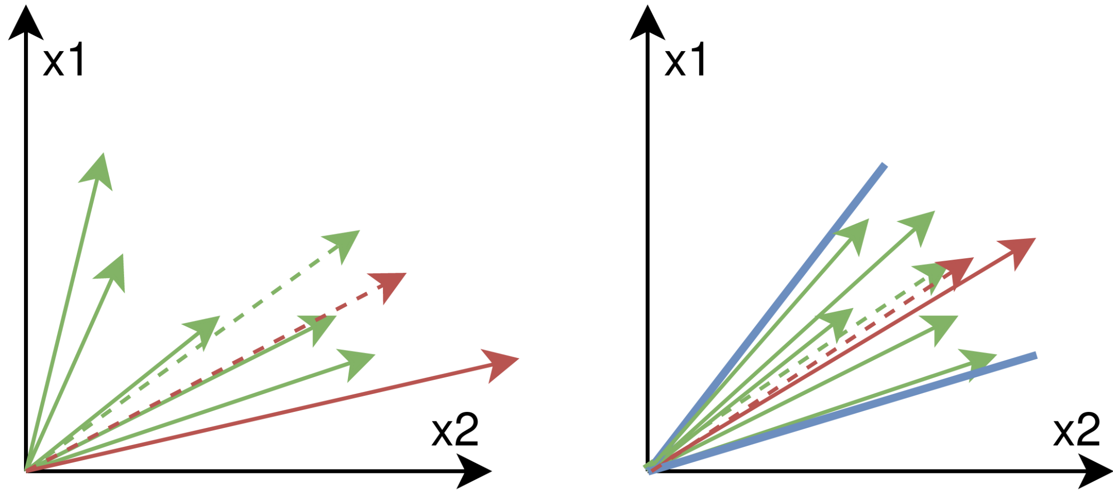

Figure 4.1: An example of a two-dimensional constraint system. In the left figure, there are no constraints. In the right figure, the vectors are constrained to an angle between the blue lines. Green arrows correspond to benign model updates while the red arrow corresponds to adversarial model updates. The dotted arrows represent the aggregated model without (green) and with (red) the adversarial model update. One can see that in the left figure, the difference between the benign aggregated model and the adversarial aggregated model is larger than in the right figure.

This mechanism adds a small number of additional hyper-parameters to the FL setup: The values for the metrics to be constrained towards. These values are determined by calculating the median of the metric values for models trained during normal, unconstrained, FL. By using the median, we ensure that the resulting metric values are in areas suitable for the benign models, since our threat model assumes more than 50% of models to be benign. Using other values is also possible and is explored in Sect. 6.5.5. This results in the following process for each round of FL: First, the aggregation server chooses a subset of clients for the next round and sends the current global model to these clients. Then, the clients train the model as usual, without constraints. After training, the local models are sent back to the server, which calculates the median of each metric value. It then communicates this value back to the clients, which then train their model to adapt to these constraints. The clients send the adapted model back to the server, which aggregates them after passing a defense mechanism. While this means that our setup brings a high network overhead, since the model needs to be transferred from the clients to the server twice, the security benefit achieved by using our system outweighs the disadvantage caused by the additional overhead. Sect. 7.5 explains how the system's overhead could be reduced by performing some modifications. The system further causes a low storage overhead, since the metric values need to be stored on the clients, and a high computation overhead, since an additional training step needs to be performed as part of each round of FL. Sect. 7.5 introduces some concepts that might be able to reduce this overhead.

In our experiments, this setup is implemented in a simplified form. This is done to reduce the effort for implementation and execution of the experiments. Instead of the split between the unconstrained training and the following training for adaption to the constraints, only the adaption step is performed. Since our system uses the unconstrained training step as source for the metric target values, we replace these with values derived from training benign models and calculating their mean metric values.

4.3 Research Questions

The questions that arise around the constraint system can be roughly split into two categories: First, questions around the mechanism used for constraining and the influence the constraints have on local, benign, and malicious training. This leads us to **RQ1** and **RQ2**, which focus on the exact setup, and the influence this has on the backdoors during local training. Second, on the FL side, the questions focus on the effectiveness the constraint system has for reaching the goal of improving defense mechanisms against poisoning attacks. Here, **RQ3** and **RQ4** focus on specific characteristics of the constraint system, while **RQ5** is more open-ended about whether the system fulfills its main purpose.

RQ1: How should constraints be applied and how does that influence the accuracy and convergence speed of the trained model?

RQ2: How does the accuracy of a backdoor change with constraints on the machine learning model?

RQ3: Are backdoors trained in a constrained environment effective after the server-side aggregation?

RQ4: What is the effect on the resilience of backdoors trained in a constrained environment?

RQ5: Is the effectiveness of existing defense mechanisms improved by the use of constraints during the learning process?

Our research questions have a strong experimental focus. The rest of the thesis focuses on experimentally finding their solutions. Sect. 4.5, gives an overview of how the research questions are resolved, with the central part of the thesis then being their evaluation in Chapter 6.

4.4 Constraining Metrics

The central part of our system is a method by which the metrics of a model can be constrained during training. This can be achieved in multiple ways. First, the approach used by Bagdasaryan et al. [17] for passing DF mechanisms can be used. As explained in Sect. 2.2, when using this approach, an additional loss value is added to the loss function for each constraint that gives high loss values when the constraint is not fulfilled and low loss values when the constraint is fulfilled (see Equation 2.2).

A different approach to constraining ML models is to apply the loss functions for all tasks, i.e., normal training and the constraints, individually, in separate optimization steps. We call this approach *Individual Optimization* for the rest of this thesis. For this approach, each loss function that is optimized uses an individual optimizer instance, each responsible for one constraint and hence one loss function. For each batch of samples, the program iterates over all loss functions (i.e., the normal loss function for the main-task/backdoor training and the loss functions for optimizing the constraints), calculates the output of the model for the current input, and the loss function for the current model and outputs. Finally, the backward pass for the loss value is calculated and the optimization step is executed.

4.5 Experiments

The following sections give an overview of the experiments performed for evaluating our research questions. The first experiments, as described in Sect. 4.5.1 focus on baseline

measurements, which we use to evaluate the performance of the proposed approach. Afterwards, in Sect. 4.5.2, we describe the experiments for finding the best approach to constraining metrics. Next, Sect. 4.5.3 describes the evaluation of the effectiveness of the constraint system. The chapter concludes with an overview of further experiments on the constraint mechanism in Sect. 4.5.4.

4.5.1 Baseline Measurements

To properly evaluate the effectiveness of an approach, some baseline values are needed for comparison. These values are important for the evaluation of all research questions. They are determined experimentally and include:

1. The (aggregated) backdoor accuracy that is reached by unconstrained adversarial training and Bagdasaryan training. We use these as comparison values for the backdoors trained under the constraint system.
2. The values that the metrics take during unconstrained training. We use these as part of the configuration of the constraint system.
3. The amount of training required for (constrained) training to reach "stable" metric values. We require this to ensure that our experiments are adequately configured.

4.5.2 Determining the Best Strategy for Constraining Metrics

To evaluate **RQ1**, we first need to determine the best strategy for constraining the metrics. We achieve this by comparing two approaches for constrained ML. The first approach is **Bagdasaryan Training**, as described in Sect. 2.2. The second approach is **Individual Optimization**.

Before the approaches are compared directly, several hyper-parameters of each approach are evaluated to ensure that each approach is executed in the way that leads to the best possible results. For Bagdasaryan Training, these hyper-parameters are the method used for weighting the individual loss values, the point in time at which the weighting is performed, and the alpha value that is used to weight the main-task loss against the constraint losses. For Individual Optimization, the only hyper-parameter that is evaluated is the learning rate that is used for optimizing the constraint losses. This evaluation then concludes with a direct comparison of the approaches.

4.5.3 Effectiveness of the Constraint System

To evaluate whether the system is helpful in differentiating between benign and adversarial models (and thus answer **RQ5**), we determine whether it can create a situation where the benign models are more similar to each other than to the adversarial models. We evaluate this by applying the previously developed system as it would be applied in a real-world scenario and observe how it influences the values taken by the constrained metrics. In addition to the quality of the adaption to the constraints, an important part of the effectiveness of the system is the increase in training time caused by the constraints. We evaluate this in comparison to the time required for unconstrained training.

The evaluation of the effectiveness concludes with determining the answer to **RQ4**. Here, we research the *resilience* of the system, i.e., the number of rounds of training the backdoor remains effective while it is not being renewed by an adversary. Resilience is important since it influences the cost and complexity of an attack. The results of this experiment also answer **RQ2** and **RQ3**.

4.5.4 Different Setups

After evaluating the effectiveness of the constraint system, we determine its behavior when various hyper-parameters of the system change. Some of these can be changed arbitrarily by either the client or the server in the FL setup, e.g., the learning rate or the batch size, while others are more fundamental properties of the system or the client, e.g., the distribution of the client's training data. This means that the evaluation here has the purpose of both determining how our configuration could be changed to lead to different results, and how it would behave in different environments. It is thus mainly relevant for answering **RQ1** and **RQ5**. The hyper-parameters we evaluate here are different distributions, learning rates, batch sizes, optimizers, constraints target values, momentum values, datasets, and models.

5. Implementation

The experiments are implemented in Python [49] using a custom experimentation framework on top of PyTorch [50].

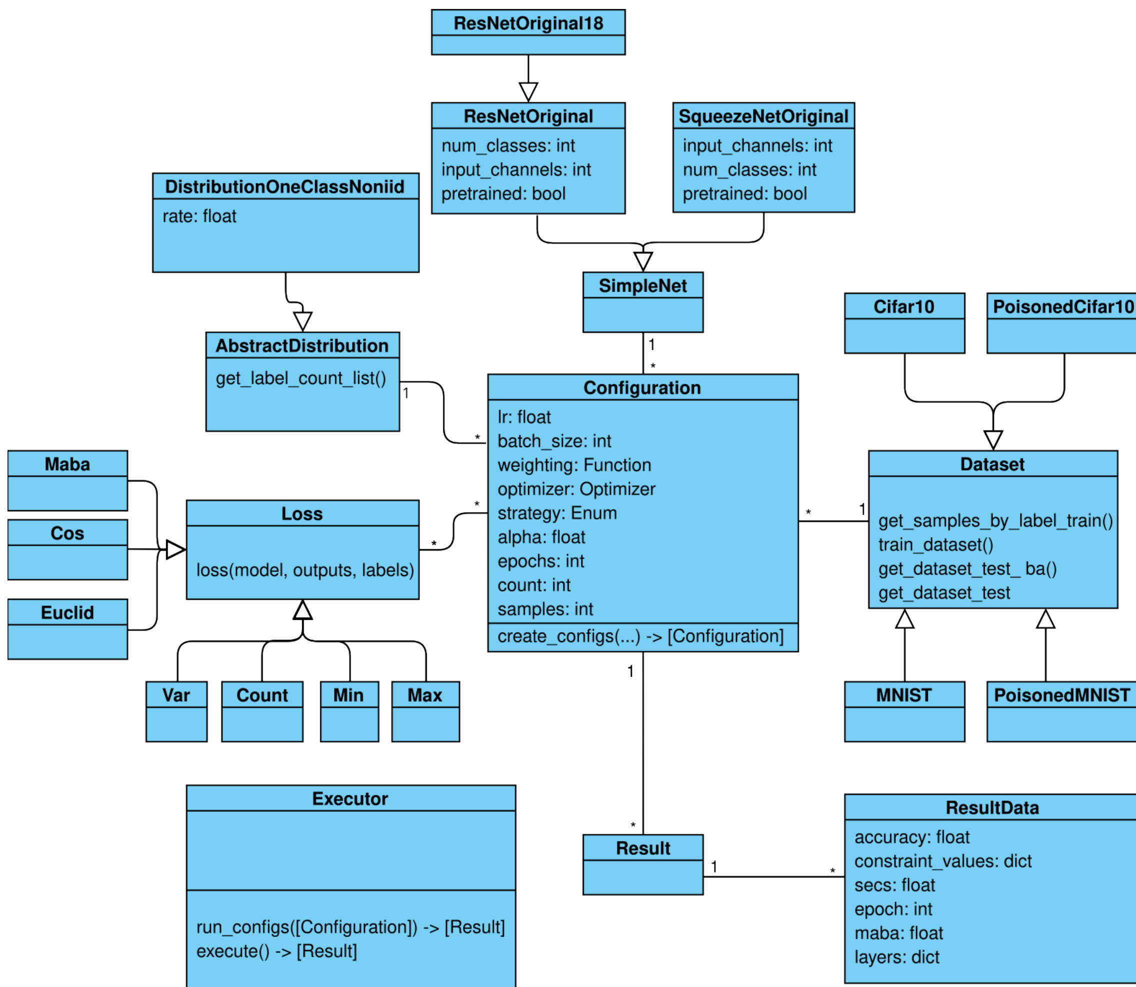

Figure 5.1: Class Diagram for the core parts of the Machine Learning framework.

Figure 5.1 shows a class diagram of the core parts of the framework. The central class

is `Configuration`, which describes a setup by collecting all relevant hyper-parameters. Typically, configurations are created from lists of alternatives for each hyper-parameter, where all combinations of alternatives are expanded to new configurations. As an example, an experiment with two alternative learning rates, three batch sizes, and two weighting approaches would lead to 12 configurations being created. When a configuration should be run multiple times (i.e., 6 times as is most common in our experiments), only one configuration for it is created, which is then run multiple times. In some cases, it is necessary to manually change configurations after they were automatically created. This can be useful when not the full combinatorial set of configurations is needed.

The results are stored in `Result` objects, which are generated by the `run_configs` function of the `Executor` class. When a configuration is executed multiple times, multiple result objects are created. The results are then stored in a JSON file and various renderings for metrics are created using MatPlotLib [51]. When multiple configurations are run in the same invocation of `run_configs`, the results are visualized in the same diagram.

After training, the `Evaluator` class measures and collects various metrics for the model, notably, the main-task accuracy, backdoor accuracy (if applicable), metric values, and training time.

All objectives (i.e., main-task/backdoor-task training and metric training) are modeled as child classes of `Loss`. The loss calculation itself is done in the `loss(model, outputs, labels)` function that objective implementations need to implement. In practice, metric losses only require access to the `model` parameter, while `outputs` and `labels` are relevant for the main-task training.

For main-task (and backdoor task) learning, the objective is implemented in the `Maba` class using PyTorch's built-in `CrossEntropyLoss` function.

Listing 5.1: General pattern for the implementation of loss functions. First, the absolute difference between the metric value and the targeted metric value is calculated. Subtracting half of the range and passing the value through a ReLU leads to values smaller than half of the range being set to 0.

```
1 distance = ...
2 loss = ReLU(|distance-target_value| - value_range / 2)
```

For the metrics, the implementation typically follows the pattern shown in Listing 5.1. The central part of the loss value - a value that is small, when the current distance is close to the target and large otherwise - is calculated as the absolute difference between distance and target value; using the `range` parameter, the objective can be "weakened", i.e., the calculation can be modified to accept values inside a certain range as perfect fulfillment of a constraint. The ReLU function is only significant when using a non-zero range, since it ensures in this case, that the loss value is never smaller than 0.

Listing 5.2: Implementation of the loss function for the Cosine metric. At its core, the `cosine_distance_for_backpropagation` function iterates over the model's layers and sums up their cosine distances.

```
1 def cosine_distance_for_backpropagation(
2         model_one, model_two_dict: dict)
3             -> torch.Tensor:
4     summed_cosine = torch.zeros(1).to(cuda_default())
5     for layer_name, layer in model_one.named_parameters():
6         if layer_name not in model_one
7             .get_named_parameters_for_distance_calculations():
8                 continue
```

```
 9            layer2 = model_two_dict[layer_name]
10            cs = torch.ones(1).to(cuda_default())
11                - cosine_similarity(
12                    layer.to(cuda_default()).view(-1).reshape(1, -1),
13                    layer2.to(cuda_default()).view(-1).reshape(1, -1)
14                )
15            summed_cosine = summed_cosine + cs
16        return summed_cosine.reshape([])
17
18    def loss(self, model, outputs, labels):
19        distance = cosine_distance_for_backpropagation(
20            model, self.ref_model.state_dict())
21        return r(abs(distance - self.ref_value)
22            - self.value_range / 2)
```

Listing 5.2 shows the implementation of the COS metric as an example. Here, the core work of calculating the metric is handled in `cosine_distance_for_backpropagation`. The result of this calculation is then handled as in Listing 5.1. During the distance calculation, a `for` loop iterates over the layers of the model, calculating the cosine similarity using PyTorch's built-in `cosine_similarity` function and summing the results to get the cosine distance. The calculation of the metric values is assisted by some utility functions in the `DistanceHandler` class.

Instances of these objective classes can then be added to the list of objectives in a configuration object. By default, the framework does not measure any metrics, even when they are added to this list. To measure a certain value, the objective objects must be added to the `evaluate_losses` list of the configuration object. Using this approach, the optimized objectives can be treated entirely separately from the measured objectives.

As an optimizer, the built-in SGD implementation is typically used. Since the creation of an optimizer object requires the model and the learning rate to be passed to the constructor, the object itself is not created during the configuration of the setup. Instead, the optimizer is passed indirectly: the configuration stores a function that returns an optimizer object, which is then invoked internally. This allows for keeping the configuration of the learning rate separate from the configuration of the optimizer. When an optimizer with momentum or weight decay is needed, it can be created using a similar function (`sgd_momentum`, which takes as parameters the desired Momentum and Weight Decay values).

The weighting methods are implemented as functions with the signature `weigh(losses: [Tensor]) -> [float]` that return the weights after calculating them from the losses. This function is stored as part of the configuration.

The weighting approach (weighting for each batch, weighting once for each model, or using the individual optimization approach) is configured using an enum and implemented using different functions in the `Executor` class. The implementation of the Individual Optimization algorithm is shown in Listing 5.3. The program iterates over the training data (line 3) and the loss functions (line 4). It then resets the gradients of all optimizers (lines 5-6), and performs the usual calculation of the loss value, backward pass, and optimization (lines 7-10).

Listing 5.3: Implementation of the *Individual Optimization* approach. For each batch, the code iterates over all loss functions and performs the normal training steps for each.

```
1 def train_individual_optimization(
```

```
2        model, loss_functions, optimizers, training_data):
3            for inputs, labels in training_data:
4                for loss_function in loss_functions:
5                    for key, opt in optimizers.items():
6                        opt.zero_grad()
7                    outputs = model(inputs)
8                    loss = loss_function.loss(model, outputs, labels)
9                    loss.backward()
10                   optimizers[loss_function].step()
```

Models are implemented as classes that wrap the classes built-in in PyTorch. For the `ResNet18` implementation, either a randomized model or a pretrained model can be created. The model is automatically adapted for usage with the CIFAR10 dataset. This is done by replacing the last layer with a fully-connected layer with 10 outputs and the first layer with a convolutional layer with 3 input channels. Both layers start with randomized weights.

Datasets are implemented in a similar way, with classes wrapping the infrastructure available from PyTorch. Before usage, the datasets must be manually post-processed by calling the `postprocess` function The dataset classes also perform adequate transforms on the dataset. For the CIFAR10 [29] dataset, this includes resizing the images, performing some randomized changes to the image's size and orientation, and normalizing it. Adversarial training is automatically done when the `PoisonedCifar` dataset is used. The backdoor that will be implanted is hard-coded into the `PoisonedCifar` class. When evaluating a different backdoor, it can easily be changed there. In case different backdoors should be compared directly, the backdoor has to be added as a separate parameter of the `Configuration` object.

Training-data distributions are implemented as subclasses of `AbstractDistribution` and are applied internally after being configured in the configuration object. Other hyper-parameters (e.g., learning rates, alpha values, poisoned data rate (PDR)) are implemented using adequate basic data types and passed to the configuration.

For aggregating models using the FedAVG algorithm, the `aggregated_stats()` function is used. It returns the average model while also evaluating the model and saving some statistical data to a file. Some utilities for scaling are implemented in the `ScalingHandler` class. The processing device used for training the models is determined in the `cuda_default` function. It can be changed by changing the return value of this function.

Experiments are implemented on top of this framework in individual Python files, each being an individual program that can be executed directly. Setting up a new experiment generally involves the following steps:

1. Creating the file.
2. Configuring the various hyper-parameters in the form of lists and variables.
3. Creating the `Configuration` objects by calling `create_configs`, passing the hyper-parameters as parameters.
4. Executing the configurations by calling `run_configs`
5. (Aggregating the models using `aggregated_stats`)

For more advanced experiments, e.g., when additional evaluation of the model or second-stage training is to be performed, the configurations and results can then be further used.

6. Evaluation

In this chapter, we first present the experimental setup used for our experiments in Sect. 6.1, followed by the evaluation of the experiments conducted in order to answer our research questions in Sections 6.2, 6.3, 6.4, and 6.5. It then concludes with a summary of the results in Sect. 6.6.

6.1 Experimental Setup

We evaluate our approach on a machine containing 4 Nvidia A16 GPUs, 128 GB main memory, and an AMD Epyc 7413 processor. The experiments are implemented in Python using the PyTorch framework [50] and executed on one of the GPUs using CUDA [52].

For the experiments, unless explicitly evaluating different domains via different datasets and models, hyperparameters, and different loss weighting methods, a standard ML scenario is used. The default scenario is training models to classify images from the CIFAR10 [29] dataset using a ResNet18 [53] model. The implementation is done in PyTorch [50]. The data is distributed as one-class non-IID with a non-IID parameter of 0.3, meaning that 30% of the training data is distributed identically between all classes except one, and 70% is coming from the remaining class. The PDR is 0.3. SGD with a momentum of 0.9 and decay of 0.005 is used as an optimizer. Training is done on 2560 samples per client dataset and for 20 epochs and a batch size of 64. The Bagdasaryan constraining approach (cf Sect. 2.2) is used, with the individual loss functions being weighed once to their smallest value and the main-task accuracy being individually weighed with $\alpha = 0.7$. For experiments focusing on local models, 6 benign and 6 malicious models are trained[1], while for experiments focusing on aggregated models, 6 benign and 5 malicious models are trained. For purely benign experiments, 6 models are trained.

6.2 Baseline Measurements

The next three subsections focus on determining some baseline values important for our research. First, Sect. 6.2.1 evaluates the accuracy reached by backdoors trained without constraints, followed by measuring the metric values reached during normal training in Sect. 6.2.2 and the number of epochs of training required to reach stable results in Sect. 6.2.3.

[1]By training as many malicious models as benign models, we test for a case worse than our threat model allows.

Table 6.1 Aggregated main-task and backdoor accuracies of FL setups with different momentum (M) and decay (D) values and six benign and five malicious models.

Configuration	MA	BA
M = 0; D = 0	68.8%	16.3%
M = 0.9; D = 0	66.3%	38.5%
M = 0.9; D = 0.005	66.6%	40.6%

Table 6.2 Average Main-Task and backdoor accuracies of the five malicious local models with different momentum (M) and decay (D) values.

Configuration	MA	BA
M = 0; D = 0	60.8%	57.7%
M = 0.9; D = 0	55.0%	75.0%
M = 0.9; D = 0.005	55.0%	75.1%

6.2.1 Unconstrained Backdoor Accuracy

We start our evaluation with an investigation into the effectiveness of classic adversarial training and Bagdasaryan Training as introduced in Bagdasaryan et al. [17]. Here, we first create an FL setup, train this with some of the clients as adversaries, aggregate the models using FedAVG, and measure the backdoor accuracy achieved by the aggregated model. We then use this backdoor accuracy value as a first comparison value for later experiments and to determine whether more experiments, using additional adversarial techniques are needed to train an effective backdoor when using our experimental setup. As part of this experiment, we also evaluate the influence momentum and weight decay have on the backdoor accuracy achieved by the aggregated model. We do this by performing this experiment with and without momentum and weight decay and evaluating the resulting backdoor accuracy values.

Table 6.1 shows the main-task (MA) and backdoor accuracy (BA) achieved by the aggregated models. For the MA, we see that all configurations reach a similar value, with the configuration without momentum or weight decay reaching the highest value, at 2.5 percentage points (pp) higher than the configuration with momentum and 2.2pp higher than the configuration with momentum and weight decay. This indicates that momentum and weight decay only have a minor influence on the main-task accuracy. For the backdoor accuracy, the situation is different: without momentum, the adversary fails to implant a useful backdoor into the aggregated model. When adding momentum to the experiment, the achieved backdoor accuracy rises drastically, to more than double the previous value. Even then, it is still significantly below what would be considered a strong backdoor, with the backdoor accuracy not even reaching 40% when, e.g., an accuracy above 60% is desired. The addition of weight decay leads only to a minor additional improvement to the backdoor accuracy of 2.1pp.

To determine the reasons for the backdoor accuracy being low, we also evaluate the average main-task and backdoor accuracy of the local models, shown in Table 6.2. Here, we see that the local backdoor accuracies are significantly higher than the aggregated backdoor accuracies for the respective configuration. Similarly to the previous evaluation, the models trained with momentum reach a significantly higher backdoor accuracy than the models trained without momentum, while the addition of weight decay only has a minor influence on the backdoor accuracy. The relation between local backdoor accuracy and aggregated backdoor accuracy leads to the conclusion that very strong local backdoors (e.g., with a backdoor accuracy larger than 90%) would be required to reach acceptable backdoor

Table 6.3 Accuracies of per-layer constraint training with and without scaling.

Configuration	Accuracy	Backdoor Accuracy
Per-Layer Constraints	66.6%	46.5%
Per-Layers Constraints with Scaling	35.7%	88.5%

accuracies. This leads us to the discovery that momentum has significant importance for adversarial training.

To increase the backdoor accuracy achieved for adversarial training, a special form of Bagdasaryan training can be performed. Here, the Euclidean metric of the individual layers of the models is constrained to the values the respective layer has in the global model. This *per-layer constraint* approach causes the respective layer of all adversarial models to be more similar to each other, resulting in the overall adversarial models being more similar, which leads to a higher backdoor accuracy after aggregation since all similar layers strongly influence the aggregation towards the same model. We evaluate this approach by repeating the last experiment while constraining the Euclidean metric in all layers. For this experiment, we change our standard experimental setup by not weighting the losses at all. Due to every constraint only being limited to a single layer, the individual loss functions do not compete against each other. This means that the "fairness" created by weighting the loss functions equally is not required. By not weighting the losses, we can also avoid possible effects that might arise due to significantly different loss values. We also raise the learning rate to 0.01, since this leads to improved results with the larger number of constraints.

As the first row of Table 6.3 shows, this increases the backdoor accuracy to 46.5% percent. While this is an improvement, it is still not high enough to be considered an effective backdoor. Further increases to the backdoor accuracy can be achieved by performing *scaling*, as is part of the Constraint-and-Scale approach described by Bagdasaryan et al. [17]. Here, the adversary calculates the model update (i.e., the difference between the last global model and the adversary's current model) and multiplies all values with a factor. Since FedAVG calculates the new global model by averaging all values, this increases the influence of the adversarial model on the new global model. While this approach can thus increase the backdoor accuracy in the aggregated model significantly, it changes the weights drastically, resulting in significant changes to the model's metrics, which reduces the model's stealthiness. This means that an attacker needs to carefully evaluate whether scaling is desirable or possible, considering the defense mechanisms in place [17]. We repeat the previous experiment and scale the adversarial model updates with a factor of 1.5 before aggregation. As the second row of Table 6.3 shows, this now leads to a very strong backdoor being implanted in the aggregated model. While the BA is now very high, the MA decreased significantly. This can be problematic to the adversary, since it could allow a detection mechanism to detect the malicious models. In a real-world attack, the attacker would then be required to choose a lower scaling factor, leading to a higher MA, at the cost of a lower BA.

Figure 6.1 shows the cosine metric of the local models before and after scaling. It shows that the metric values change drastically during scaling. It is also apparent that the change does not seem to follow an exact ratio: while the values after scaling are similar in value, some of the values that were above average before scaling are below average after scaling. Table 6.4 lists the values of all MESAS [38] metrics before and after scaling for one model. As the last row indicates, some metrics follow a predictable pattern during scaling: Euclid, Min, and Max scale exactly with the scaling factor, while the count metric does not change at all. There is no (obvious) pattern for Cos and Var.

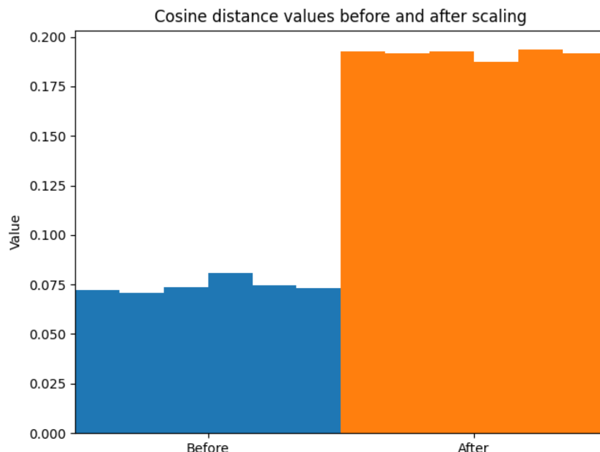

Figure 6.1: Values of the cosine distance metric before (blue) and after (orange) scaling the local models.

Table 6.4 Metric values of one local model before and after scaling.

	Cos	Euclid	Var	Count	Min	Max
Before	4.888	4.611	$7.767 * 10^{-6}$	0.510	$9.155 * 10^{-14}$	0.218
After	10.488	6.918	$6.976 * 10^{-6}$	0.510	$1.373 * 10^{-13}$	0.326
Ratio $After/Before$	2.14	1.5	0.898	1	1.5	1.5

6.2.2 Natural Metric Values

As part of our mechanism, we need values that the metrics can be constrained towards. During this evaluation, we use the average of the values that the metrics take when training multiple identical models in benign, unconstrained training. These values are also the values that are targeted in Bagdasaryan training to "hide" the adversarial models between the benign models. Since these are the values the metrics "naturally" take, these reference values will be called *natural metric values*. We measure them for several reasons: First, knowing the natural values and how much they are spread apart is useful for comparing the performance of different constraint mechanisms, i.e., how good they are at keeping metric values at a certain point in a small range, against the intrinsic behavior of the model. Second, as stated before, they are potential target values for the constraints. Intuitively, since the average of the natural metric values has the smallest combined distance to all individual values, it is the value "easiest" to reach for all individual models during benign training, meaning that the least amount of effort is required for reaching this point. This means that more training effort can be spent on achieving values *closer* to this value, or that less effort is spent on the constraints entirely, allowing for higher focus on the main-task training. We determine these values experimentally for all MESAS [38] metrics. We also measure the influence momentum and weight decay have on the values. The results are then used both as part of the configuration of other experiments and for the evaluation of momentum and decay. These experiments are only conducted as a basis for our own experiments. In a real setup of our system, the metric values are determined by performing normal, unconstrained FL on the clients and calculating the median of the respective target value.

Tables 6.5 and 6.6 show the average metric values and the range (i.e., the difference between the largest and smallest metric value) respectively. Together, they give some insight into the metrics' behavior regarding momentum and decay. **Cos** and **Euclid** behave similarly,

Table 6.5 Average of natural metric values for different configurations.

Configuration	Cos	Euclid	Var	Count	Min	Max
M = 0; D = 0	1.014	0.292	$1.21 * 10^{-5}$	0.196	$1.184 * 10^{-14}$	0.04
M = 0.9; D = 0	2.073	1.174	$1.211 * 10^{-5}$	0.199	$2.072 * 10^{-14}$	0.188
M = 0.9; D = 0.005	2.073	1.175	$1.20 * 10^{-5}$	0.509	$1.13 * 10^{-15}$	0.188

Table 6.6 Range of natural metric values for different configurations.

Configuration	Cos	Euclid	Var	Count	Min	Max
M = 0; D = 0	0.874	0.013	$9.80 * 10^{-9}$	0.019	$7.11 * 10^{-15}$	0.008
M = 0.9; D = 0	1.046	0.096	$7.71 * 10^{-8}$	0.019	$1.42 * 10^{-14}$	0.034
M = 0.9; D = 0.005	1.047	0.097	$7.67 * 10^{-8}$	0.017	0.0	0.034

with momentum leading to a significantly increased average and range in both, while the addition of weight decay only leads to a minor additional change. The **Var** metric stays mostly uninfluenced by momentum and weight decay. For the **Count** Metric, the addition of weight decay leads to a significant change in its average value, while its range doesn't change significantly. Both the **Min** and **Max** metrics are significantly influenced by the addition of momentum.

6.2.3 Number of Epochs to Train

An important characteristic of our system is the amount of training required for the constraint mechanism to reach the targeted value. Since, in general, the metric values measured at the beginning of training are not equal to the targeted metric values, enough training needs to be performed to allow the constraints to reach their targets. This is important both in our experiments since we want to measure stable results and not results that are still heavily changing, and in the real-world application of our system, where the number of epochs in a round of FL must be chosen at least as large as is required for the metrics to adapt to the constraints. To evaluate the behavior of the constraints over time, we train benign and adversarial models for 50 epochs, recording the current metric values after each epoch of training. We compare the metric behavior for constrained and unconstrained training by performing this experiment with unconstrained, Bagdasaryan-constrained, and Individual Optimization-constrained models.

Figure 6.2 shows the progression of the cosine metric values over the training epochs, with the y-axis showing the metric values and the x-axis showing the training epochs. All constrained configurations approach their target value (roughly 2.0) quickly, reaching it after the first epoch of training and then remaining relatively constant for the other epochs. It also shows that the convergence behavior of both constraint mechanisms is very similar, with the lines for both approaches being almost indistinguishable. Notably, there is a difference in the cosine values reached by unconstrained benign (green line) and unconstrained malicious (red line) models. After an initial phase of roughly 3 epochs, this difference is relatively constant at roughly 0.4. This difference could be used by basic DF mechanisms to find malicious models.

The behavior of the Euclidean metric is shown in Figure 6.3. The initial "chaotic phase" is slightly longer than for the cosine metric, at roughly 5 epochs. During this phase, the constraint approaches behave differently, but both reach their target point at the same time and then do not change significantly once there. Notably, the Euclidean metrics of unconstrained benign and malicious models are much closer together than the cosine values and seem to diverge.

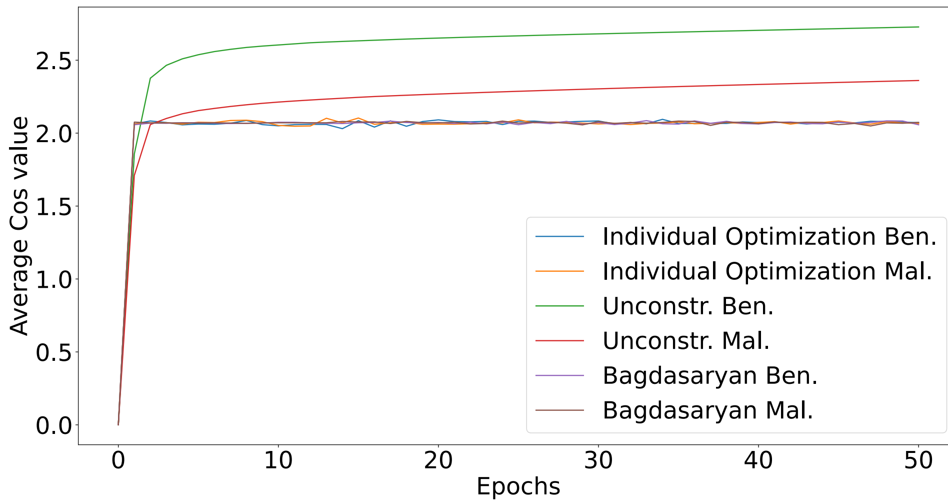

Figure 6.2: The average value of the Cos metric after each round of training. The two uppermost lines show values for unconstrained models, where the values diverge, while the values for constrained models stay close to the targeted values after as little as one epoch of training.

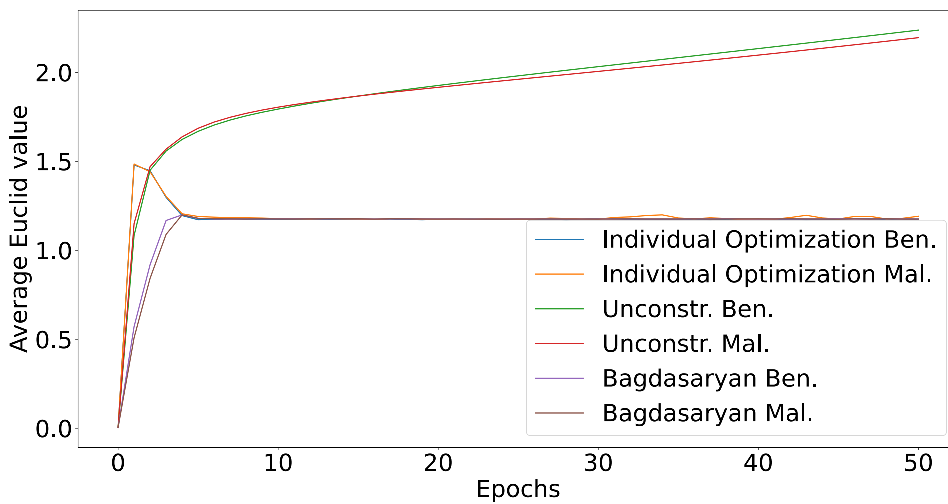

Figure 6.3: The average value of the Euclidean metric after each round of training. At the top, the values for the unconstrained models can be seen diverging, while the values for the constrained models stay close to the targeted value after roughly 5 epochs of training.

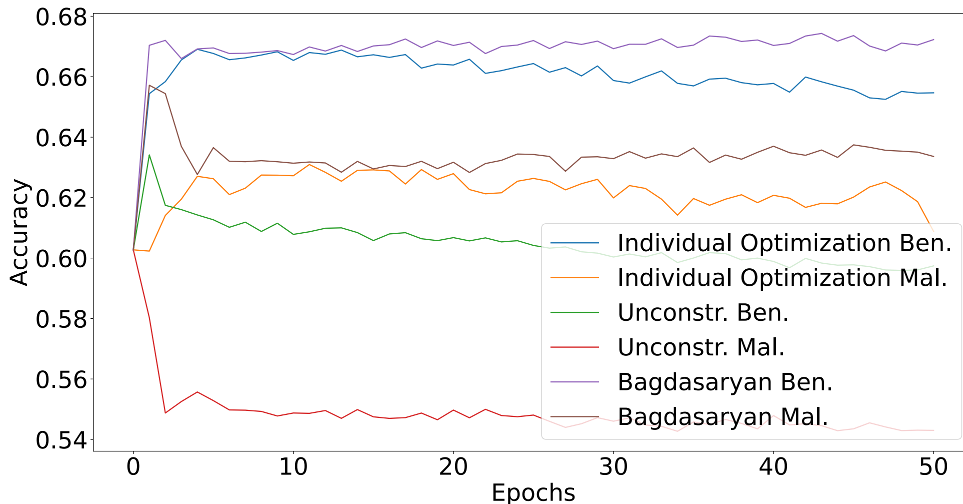

Figure 6.4: The main-task accuracy after each round of training. Benign models always reach better accuracies than the respective malicious model, with the constrained models reaching better accuracies than the unconstrained models. Notably, for all models, the accuracy does not increase significantly after the first few rounds of training.

For the main-task accuracy in Figure 6.4, we get the somewhat unintuitive result that constrained models reach a better accuracy than unconstrained models. We also see that benign models always reach better values than the corresponding malicious model. Figure 6.4 also shows that the constrained models seem to converge faster: After roughly 5 epochs, the values of the constrained models (i.e., the upper 4 values) appear more stable than the unconstrained benign value, which continues to decline slightly.

From this analysis, we see that most major changes to the model's measurements happen in the first 10 epochs of training. This means models trained for 20 epochs will show the desired adaption to the metric and longer training is not required for our experiments. It also gives a lower bound of roughly 5 epochs of training for the size of one round of FL when using our approach.

6.3 Determining the Best Strategy for Constraining Metrics

In the next sections, we execute and evaluate the experiments necessary for determining the best constraining approach, starting with the configuration of Bagdasaryan training, followed by the configuration of the Individual Optimization approach and a comparison of both approaches. This is the central part of our research.

Criteria for a good approach are:

- The accuracy achieved by training with constraints is high.
- The constraints are fulfilled well, i.e., the values of the metrics converge well to the target values.
- The training time (both in terms of epochs required for the model to converge and the time required to train each epoch) is low.

Since there is no universal answer for how to rank different strategies when the criteria differ (i.e., when one approach achieves a higher accuracy while a different approach achieves

lower constraint values), we generally assume a higher accuracy to be more important than the fulfillment of constraints and consider good adaption to the constraints to be more important than a lower training time.

For Bagdasaryan training, questions arise about how exactly the loss function should be constructed. In general, the loss function is a weighted sum of the constraint losses and the main-task/backdoor loss. In our research, we determine the answer to three questions around this function. First, how the weights should be calculated. Second, at which point during training the weights should be recalculated, and last, how strong the main-task/backdoor loss should be weighed against the other loss values.

The Individual Optimization approach allows for a high degree of flexibility during training, since all loss functions use an individual optimizer. It is possible to use a different kind of optimizer for each loss, or different learning rates, momentum, or decay values. For example, it would be possible to train the main-task loss with an SGD optimizer with momentum 0.9 and decay 0.005, while training for the constraint losses without momentum or decay. Whether this leads to beneficial results in practice would have to be researched in future work. For this approach, we only evaluate which learning rate should be used for the constraints in order to lead to the best results.

6.3.1 Weighting Method

When using the Bagdasaryan approach to adapting to constraints, a weighted sum of the different losses (i.e., constraint losses and main-task/backdoor loss) is created, which is then used as the loss function. The method used for weighting the losses in this sum thus has a significant influence on the constraint values and accuracy achieved by the trained model. During constrained training, the main-task/backdoor loss is still the most significant loss, since without a good main-task (and, during adversarial training, backdoor) accuracy, a good adaption to the constraints is not useful. For this reason, after weighting all losses, we treat this loss separately and weight it against the constraint losses using a hyper-parameter α. We thus arrive at the following Equation 6.1 for the loss function, where w_m is the weight for main-task training, $w_c^1, ..., w_c^n$ are the weights for the constraint losses, l_m is the main-task loss, $l_c^1 ... l_c^n$ are the constraint losses and $L(x)$ is the combined loss function.

$$
\begin{aligned}
w_m, w_c^1, ..., w_c^n &= weight(l_m, l_c^1, ..., l_c^n) \\
L(x) &= \alpha * w_m * l_m(x) + (1 - \alpha) \sum_{i=1}^{n} w_c^i * l_c^i(x)
\end{aligned} \tag{6.1}
$$

While there are infinitely many weighting functions, we can only evaluate a limited number of them. In general, it makes sense to weight the different constraint losses equally, since there is no intrinsic motivation to constrain one metric more than another. This leads us to evaluate these weighting functions:

1. **max**: Set the weights so that all losses have the magnitude of the largest one.
2. **min**: Set the weights so that all losses have the magnitude of the smallest one.
3. **no**: Do not perform any weighting, i.e., set all weights to 1.

We evaluate minimum weighting, maximum weighting, and no weighting by training benign and malicious models for all methods.

Figure 6.5 shows the absolute difference between the measured and targeted cosine metric values for this experiment. Here, low values correspond to a good adaption to the

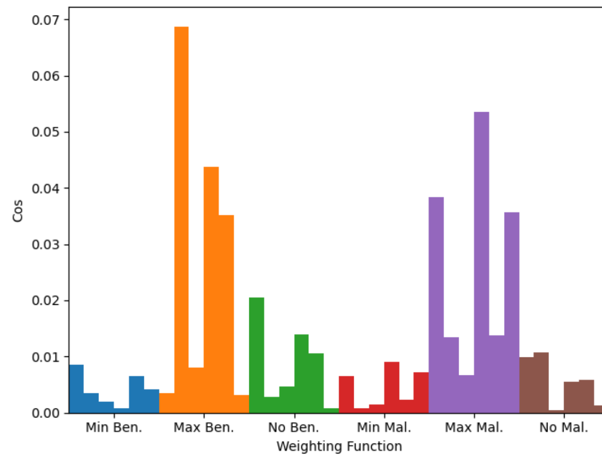

Figure 6.5: The absolute difference between measured and targeted cosine values for local models trained with different weighting functions. Lower values mean a better adaption to the constraint.

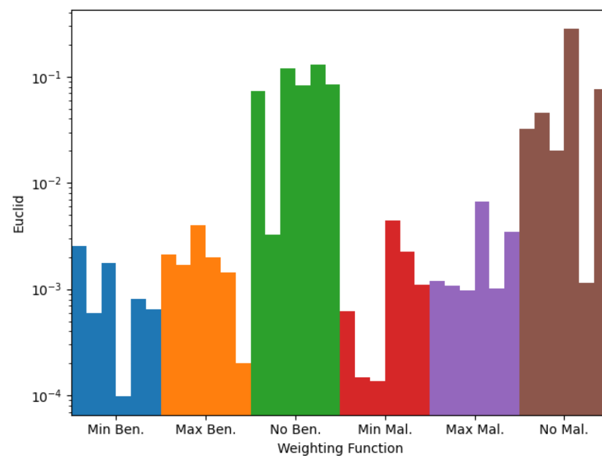

Figure 6.6: The absolute difference between targeted and measured Euclid values for local models trained with different weighting methods. Lower values correspond to a better adaption.

constraint. We can see that the maximum-weighting function leads to the worst results, since the orange and violet bars (for benign and adversarial training, respectively), take the highest values, with some of the values almost two orders of magnitude higher than the best values (e.g., the fourth blue bar and the third brown bar). When comparing minimum-weighting and no-weighting, the results are less clear, though, on the benign side, three of the green (i.e., no-weighting) measurements are higher than the highest measurements for the blue (minimum weighting) function. This leads to the conclusion that for the cosine metric, minimum weighting leads to the best results, followed by not weighting the constraints at all. Max-weighting leads to the worst results.

Figure 6.6 shows the absolute difference between the measured and targeted values for the Euclidean metric for each trained model. Again, smaller values are desired, since they correspond to a better adaption to the constraints. In contrast to Figure 6.5, not weighting at all performs worst here, as can be seen from the fact that almost all green and brown values are higher than the values for other weighting functions. For the remaining functions, on the benign side, the first five orange models perform worse than almost all blue models, meaning that min-weighting performs better here than max-weighting. On

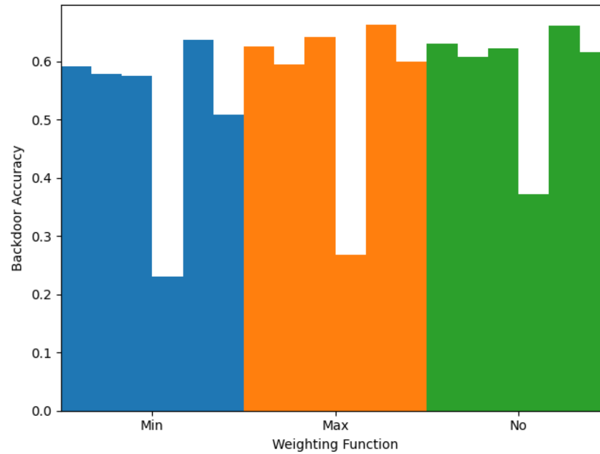

Figure 6.7: Backdoor accuracy of local models for different weighting methods. Higher values are better for the adversary. The fourth models have significantly worse backdoor accuracy due to their non-IID class being equal to the target class of the backdoor attack.

the malicious side, the situation is equivalent, with only red (i.e., minimum weighting) models 4 and 5 performing significantly worse than the best violet models. The evaluation of the Euclidean metric values thus confirms the previous evaluation of the cos metric about the min-weighting function performing best, while coming to a different result about which function is the second-best and which function performs worst.

Continuing with Figure 6.7, we see the backdoor accuracy achieved for the different weighting functions. Here, high values are desirable to the adversary, while lower values are advantageous for defenses against backdoors. Notably, for all functions, the fourth model has a significantly worse backdoor accuracy. This is caused by the non-IID class of these models being the same as the target class of the backdoor attack [2]. It is thus not an inherent property of the models and is not relevant here. Not accounting for these outliers, all bars reach roughly similar values, i.e., all weighting functions lead to similar backdoor

[2]This is caused by the significant focus of benign training on this class conflicting with the adversarial training on this class.

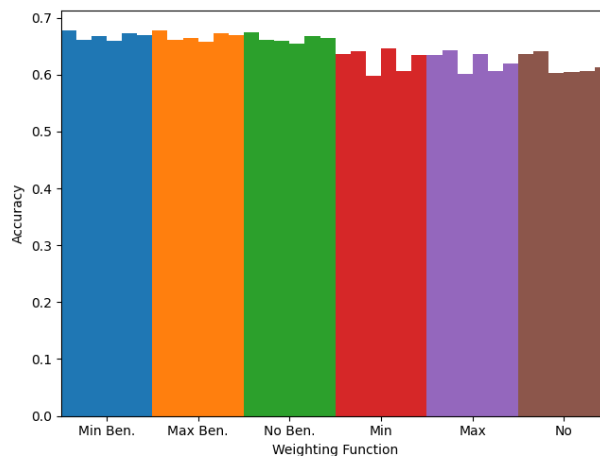

Figure 6.8: Main-task accuracy of local models trained with different weighting methods. Higher values are better. Max-weighting reaches significantly worse results than the other methods.

accuracies. While all models reach backdoor accuracies around 60%, the green (no weighting) models reach a slightly higher backdoor accuracy than the orange ones (maximum weighting), which again reach slightly higher accuracies than the blue ones (min weighting). This can be seen particularly well from the fourth model of each weighting method, where the green model performs significantly better than the orange and blue ones. This means that all weighting functions are suitable for reaching a high BA.

Figure 6.8 shows the main-task accuracy achieved by the models trained using different weighting methods. Here, the only noticeable difference between the different configurations is between the benign models (on the left) and the malicious models (on the right), with the benign models reaching a somewhat higher accuracy than the malicious models. There is no significant difference in main-task accuracy between the benign (and malicious models, respectively) models with different weighting methods. This can be seen from all benign (and malicious) values being roughly identical.

We are thus in a situation where both the Euclidean and the Cosine metric are best constrained with the minimum weighting function, while this is also not disadvantageous to the main-task training. This means that for our experiments, we continue with the minimum weighting function. An additional benefit (from the defender's perspective) is the slightly reduced backdoor accuracy this weighting function brings. This partially answers **RQ1**: When using Bagdasaryan Training, the best weighting function function is minimum-weighting.

6.3.2 Weighting Approach

In addition to the weighting function, a relevant aspect of the setup is the point in time at which the weights are calculated. This could be done at many points in time, e.g., for each batch that is calculated, at the start of each epoch, when loss values have significantly changed, or once at the beginning of training for each model.

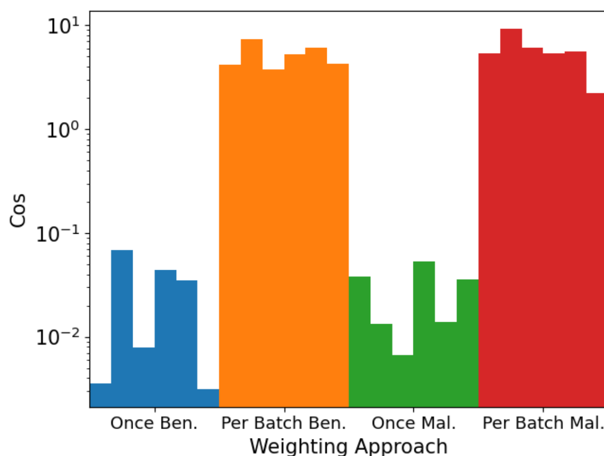

Figure 6.9: The absolute difference between targeted and measured Cosine values for local models trained with different weighting approaches. Lower values are better. Blue and green values are significantly smaller than red and orange ones. Note the logarithmic scale.

We evaluate two approaches here. The first is to weight each batch that is being calculated. This means that for each batch, the different losses are assigned the same importance, even if one of the losses has previously already been optimized significantly better than a different one. The second is to calculate the weights once, at the beginning of training, and use these weights for the rest of training. To determine which approach works better,

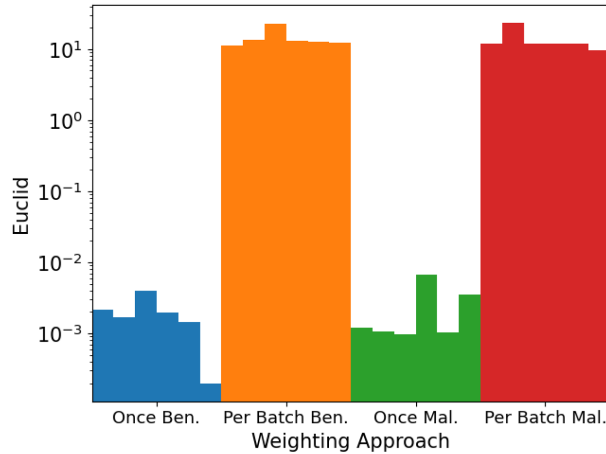

Figure 6.10: The absolute difference between targeted and measured Euclid values for local models trained with different weighting approaches. Lower values are better. Blue and green values are significantly smaller than red and orange ones. Note the logarithmic scale.

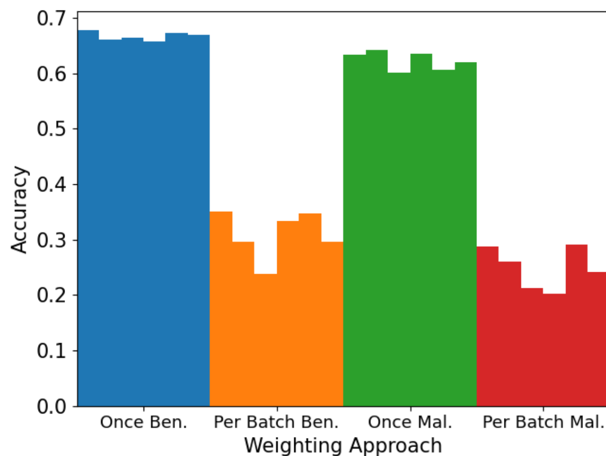

Figure 6.11: Main-task accuracy for local models trained with different weighting approaches. Higher values are better. Once-weighting (blue and green values) reaches a significantly higher accuracy than batch-weighting (orange and red values).

we evaluate this both for benign and malicious training and determine the better approach based on the achieved main-task accuracy, metric adaption, and backdoor accuracy.

Figure 6.9 shows the absolute difference between targeted and measured values of the cosine metric for the two approaches. Here, lower values are desired, since they correspond to a better adaption. It is clear that using the once-weighting approach leads to better results, with the blue and green values being significantly lower than the orange and red values. For the equivalent data for the Euclidean metric in Figure 6.10, the same result manifests. Even more pronounced than for the cosine metric, once-weighting reaches significantly better results than batch-weighting by several orders of magnitude.

Figure 6.11 shows the main-task accuracy achieved when using the two approaches. Here, higher values are better. The results show a drastic difference between the accuracy achieved when weighting once, at the beginning of training and when weighting for each batch, with the values for once-weighting (blue and green) being roughly twice as high as the values for batch-weighting (orange and red). This means that for reaching a high MA,

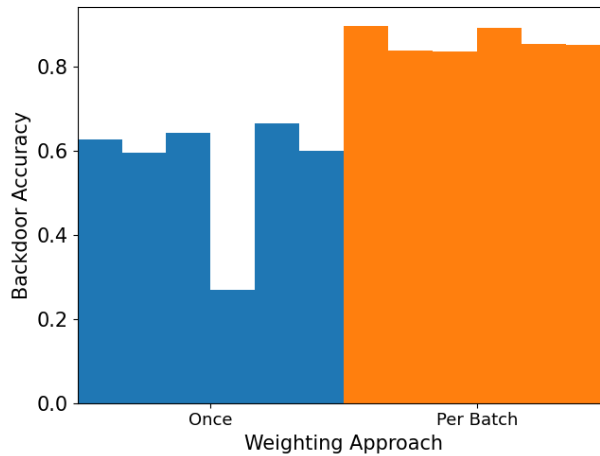

Figure 6.12: Backdoor accuracy for local models trained with different weighting approaches. Higher values are better for the adversary.

once-weighting is the better approach.

Figure 6.12 shows the backdoor accuracy, where high values are desired by the adversary, while low values would be beneficial to defenders. Here, we see that all orange (i.e., batch-weighting) values are significantly higher than all blue (i.e., once-weighting) values, meaning that per-batch weighting leads to stronger backdoors.

The evaluation of the two weighting approaches has shown conflicting results: While weighting only once, at the beginning of training for each model, is the only approach leading to a high main-task accuracy and a good adaption to the constraints, it comes at the cost of a worse backdoor accuracy than when weighting individually for each batch. For benign learning, this situation is ideal: for all relevant criteria, once-weighting is the better approach. For adversarial learning, this result poses a problem. While this additional backdoor accuracy would be important for reaching a good backdoor accuracy after federated averaging, it does not lead to an adequate adaption, making batch-weighting unsuitable.

Based on these results, it is clear that creating weights once for each model, at the beginning of training, is the better approach to training models and that the other approach should not be further considered. This is a partial answer for **RQ1**: When using Bagdasaryan Training, weights should be calculated once, at the beginning of training, and then used for the rest of training.

6.3.3 Alpha

The α value is used to weight the constraint losses against the MA/BA loss during Bagdasaryan training, with higher α values giving a stronger focus to the MA/BA loss. The choice of α value is generally a trade-off between achieved accuracy (both main task and backdoor) and the achieved constraint values. This means that there is potentially a range of acceptable α values, depending on the exact scenario. Here, we determine the influence the α value has on the achieved main-task accuracy, the quality of the metric adaption, and the backdoor accuracy. To do this, we train models with α values between 0 and 1 in step sizes of 0.2 and evaluate the data. As a result, we gain insight into the behavior of our model under different α values and knowledge about which values lead to acceptable results.

Figure 6.13 shows the values of the cosine metric for models trained using the different α values. Here, low values correspond to a better adaption. For $\alpha = 0.0$ (blue models), all

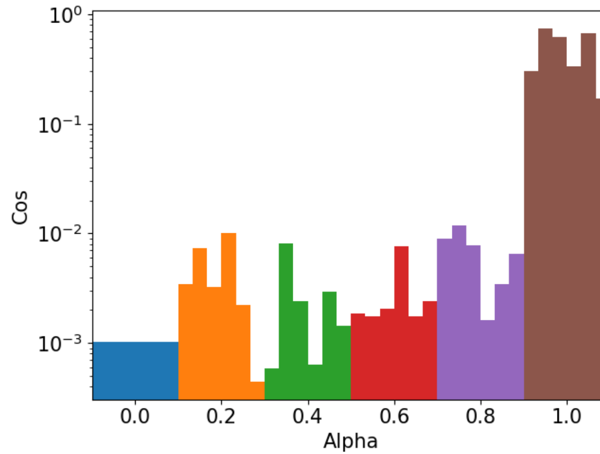

Figure 6.13: The absolute difference between targeted and measured Cos values for models trained with different α parameters. Lower values are better.

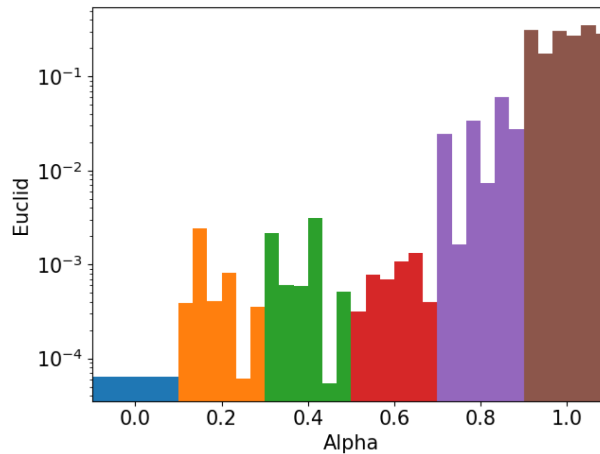

Figure 6.14: The absolute difference between targeted and measured Euclid values for local models trained with different α parameters. Lower values are better.

models adapted exactly equally well. This is expected, since for this α value, no main-task training is done at all. This result is thus not relevant in practice. For $\alpha = 1.0$ (brown models), the metric values are by far the largest, which is also expected, since the metrics are not constrained in this configuration. For the other values (0.2, 0.4, 0.6, and 0.8), there is no strong difference in cosine values, with the worst individual models for each α value having roughly equal values. For a more exact analysis of the adaption quality of the cosine metric, the average values are given in Table 6.7. It shows that the cosine metric adapts best for α values 0.4 and 0.6, while both lower (0.2) and higher (0.8) values lead to worse results. This means that there is no strong correlation between adaption quality and α value at least for α values in this range.

For the Euclidean metric, as shown in Figure 6.14, the results are somewhat different. Here, we see that for $\alpha = 0.8$ (violet values), the adaption quality is already significantly worse than for the smaller values, with almost all individual values being worse than any individual value of the other configurations. For the values between 0.2 and 0.6, we again see no significant difference, with all individual values being roughly similar. The averages of the Euclidean values for the different α values are also shown in Table 6.7. This data confirms our evaluation: $\alpha = 0.2$ and $\alpha = 0.6$ lead to the best results, with $\alpha = 0.4$ only slightly behind. $\alpha = 0.8$ already leads to a significantly worse average Euclidean metric

Table 6.7 Average and range of difference between targeted values and achieved values for local models.

α	Cos Average	Euclid Average
0.0	0.001	$6.46 * 10^{-5}$
0.2	0.004	0.0007
0.4	0.002	0.001
0.6	0.002	0.0007
0.8	0.006	0.026
1.0	0.472	0.284

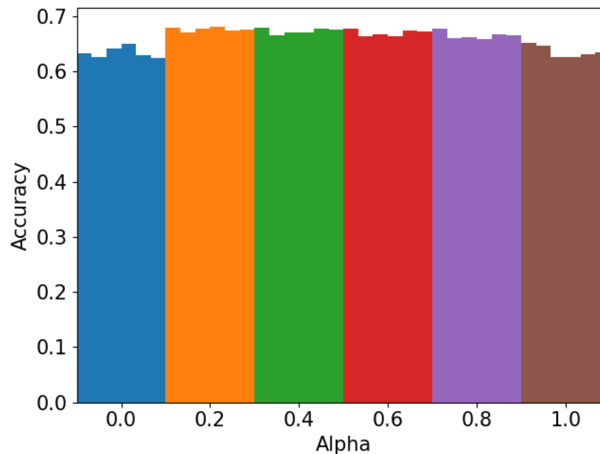

Figure 6.15: Main-task accuracy of local models trained with different α parameters. Higher values are better.

value. Again, there is no correlation between adaption quality and α value for α values between 0.2 and 0.6. While we have seen this result here, this does not mean that the same range of α values is applicable to all situations. In different scenarios, the trade-off achieved with different α values might look significantly different.

The main-task accuracy is shown in Figure 6.15. Here, higher values are desired. It shows that there is no significant difference in main-task accuracy for the α values between 0.2 and 0.8, with all individual values being roughly equal in size. The values are only smaller for $\alpha = 0.0$ (blue values), where no training is performed at all, and $\alpha = 1.0$, where the model is trained normally. The main-task accuracy being worse for normal training than for constrained training is notable since it could be considered an additional benefit of constrained training.

The backdoor accuracy is shown in Figure 6.16. Here, high values are beneficial to the adversary, while low values are advantageous to the defenders. We can see a strong, roughly linear correlation between the α value and the achieved backdoor accuracy, meaning that higher α values also lead to higher backdoor accuracy. This means that an adversary should choose a higher α value (e.g., 0.7) since this brings an efficient backdoor while still adapting well to the constraints.

From this analysis, we have seen that a wide range of α values, roughly between 0.2 and 0.6 can be chosen without having a major effect on the adaption or the main-task accuracy. For adversarial training, a higher α value (e.g., 0.7) should be chosen to lead to a high backdoor accuracy. This is again a partial answer of **RQ1**.

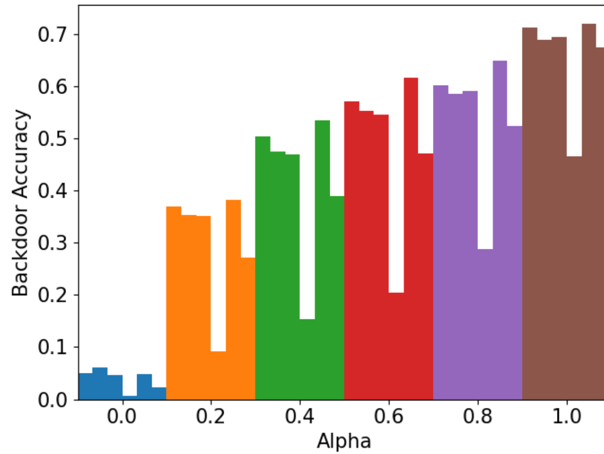

Figure 6.16: Backdoor accuracy of local models trained with different α parameters. Higher values are better for the adversary.

6.3.4 Constraint Learning Rate

One of the advantages of the Individual Optimization approach is that training for the main-task (and backdoor) and the constraints is more decoupled than during Bagdasaryan Training. This allows us to use a different kind of optimizer, different learning rates, momentum, weight decay, etc. While this leads to a potentially infinite number of configurations, our focus is on the learning rate used for training the constraints.

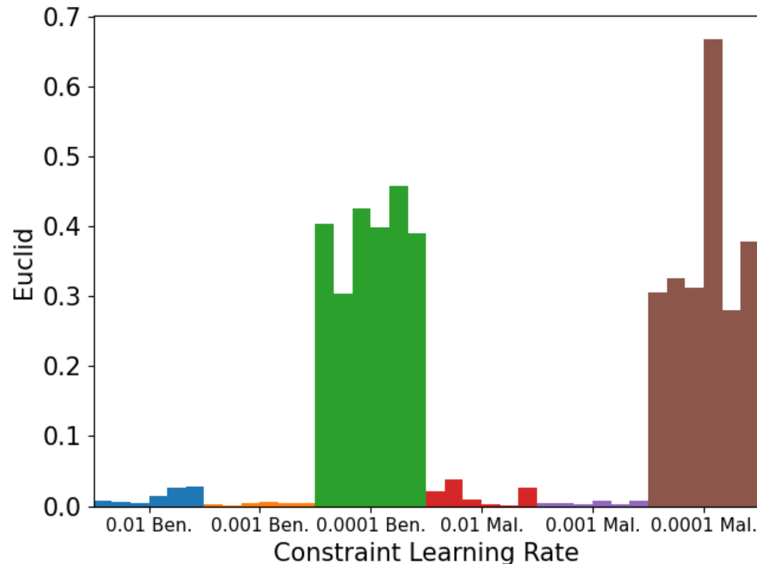

Figure 6.17: The absolute difference between targeted and achieved Euclidean values for local models trained with different constraint learning rates. Lower values are better.

Intuitively, a higher learning rate here means reaching the targeted metric values faster, at the cost of the metric values not remaining as close to the targeted value once the value has been reached. For a smaller constraint learning rate, the opposite situation would happen: the optimization is slower at reaching the target value but adapts better if it reaches the target value at all. We determine experimentally whether this assumption holds and which learning rate values reach the best adaption. We evaluate this for benign and malicious training by training models with constraint learning rates 0.01, 0.001, and 0.0001. The

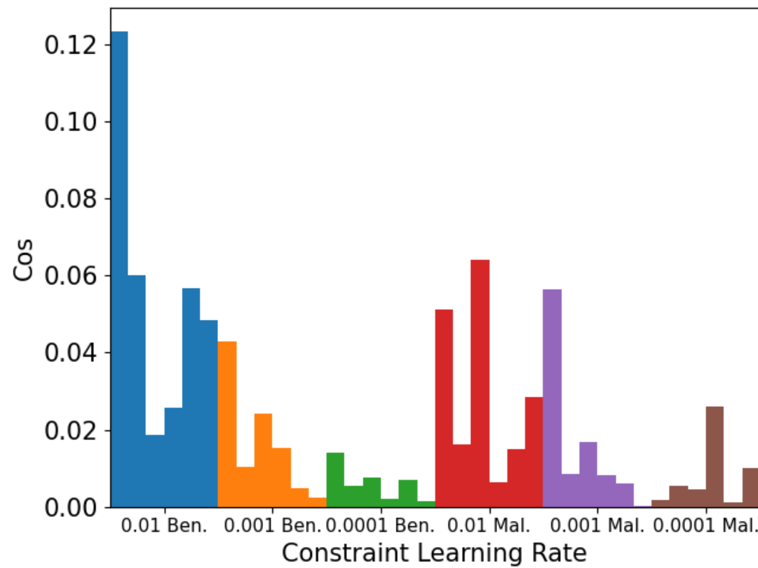

Figure 6.18: The absolute difference between targeted and achieved cos values for local models trained with different constraint learning rates. Lower values are better.

result is then evaluated on the achieved main-task accuracy, constraint adaption, and backdoor accuracy to determine the best constraint learning rate.

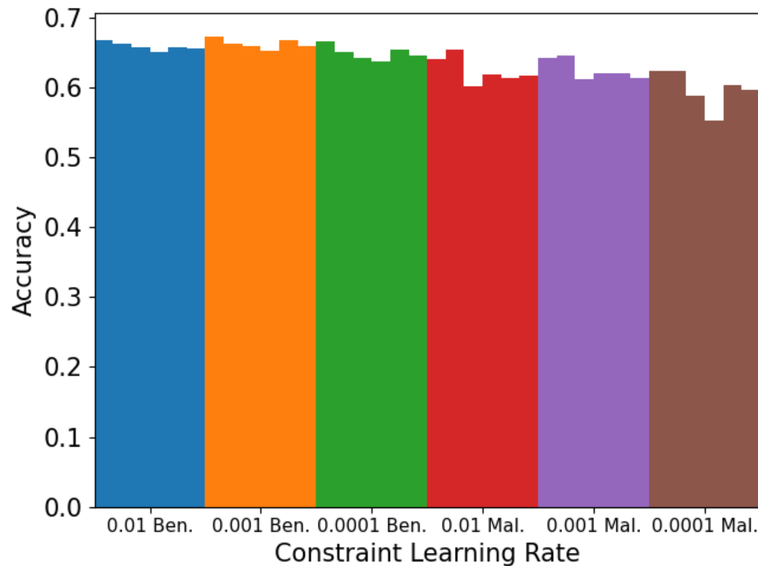

Figure 6.19: Main-task accuracy for local models trained with different constraint learning rates. Higher values are better.

Figure 6.17 shows the absolute difference between measured and targeted values of the Euclidean metric. Lower values are better. We see the effect that was assumed in Sect. 4.5: for the larger two constraint learning rates, the metrics adapt well, with the models with constraint learning rate 0.001 adapting best. For the smallest learning rate, the metrics fail to adapt well, as indicated by the high values.

Figure 6.18 shows the absolute difference between measured and targeted values of the cosine metric. Lower values are better. Here, the results are somewhat different from the results for the Euclidean metric, with the best adaption being achieved by the models with the lowest constraint learning rate. We do, however, observe a correlation between smaller

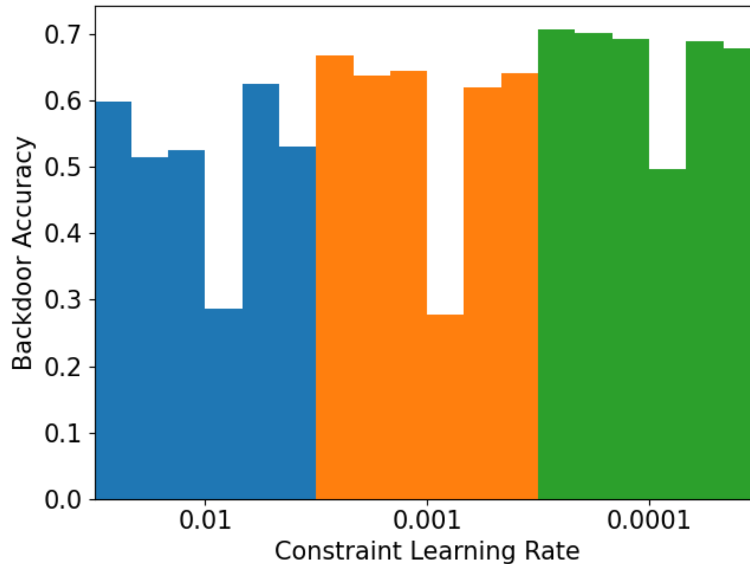

Figure 6.20: Backdoor accuracy of local models trained with different constraint learning rates. Higher values are better for the adversary.

constraint learning rates and better adaption, with the models for the largest constraint learning rate achieving the worst adaption and the models for the medium constraint learning rate following behind.

The main-task accuracy, as shown in Figure 6.19, shows no strong influence caused by the constraint learning rate: all benign models take similar main-task accuracy values, with the adversarial models reaching slightly lower values. The backdoor accuracy, as shown in Figure 6.20, improves significantly with a lower constraint learning rate, as can be seen from all green models (lowest constraint learning rate) achieving higher backdoor accuracies than all orange models (medium constraint learning rate), which in turn achieve higher backdoor accuracies than all blue models (highest constraint learning rate), when not considering the outliers (fourth model for each configuration).

For benign training, this means that the choice of constraint learning rate should be 0.001, since this results in the best adaption while having no significant influence on the achieved accuracy. For adversarial learning, the adversary would have to choose this value as well, since, while a lower learning rate would provide a better backdoor, the Euclidean metric would not adapt well enough to pass a detection mechanism. This would be a partial answer of **RQ1**, if Individual Optimization was used as the constraining approach in our mechanism.

6.3.5 Bagdasaryan Training or Individual Optimization

In the previous sections, we have developed two approaches to adapting to certain constraints. We now determine which of the two approaches performs better at adapting to the metrics when comparing them directly.

As before, we do this by evaluating the main-task accuracy, adaption quality, and backdoor accuracy reached when training using the respective approach. Figure 6.21 shows the absolute difference between targeted and measured values of the cos metric for both approaches. Lower values are desired, as they correspond to a better adaption to the constraint. For this metric, both approaches lead to a roughly similar adaption, with the Bagdasaryan approach resulting in a slightly better adaption, as visible from the slightly lower values especially on the benign side (blue and orange models).

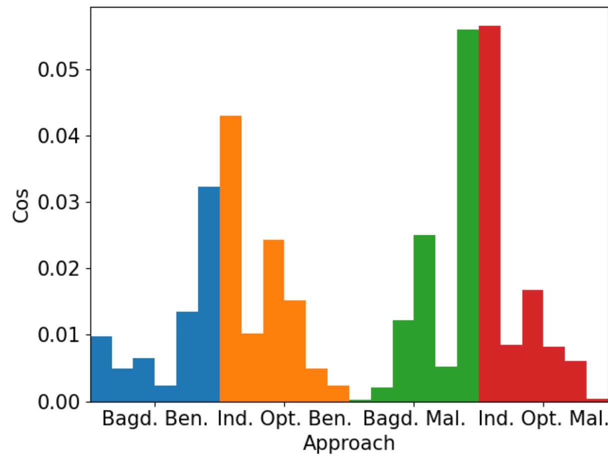

Figure 6.21: The absolute difference between targeted and measured cosine values for local models trained with both approaches. Bagdasaryan training constrains the metrics slightly better. Lower values are better.

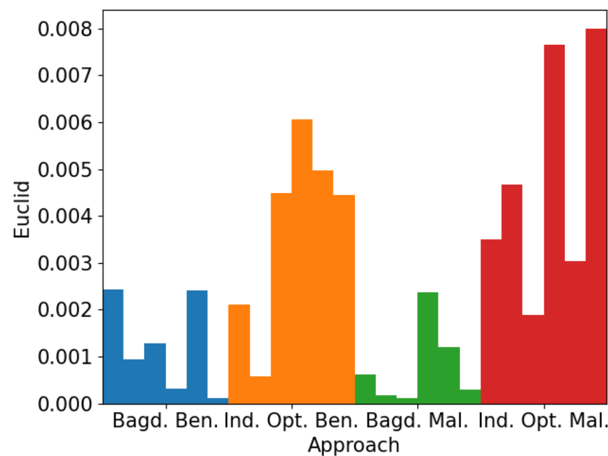

Figure 6.22: The absolute difference between targeted and measured Euclidean values for local models trained with both approaches. Bagdasaryan training performs significantly better at constraining the values. Lower values are better.

Figure 6.22 shows the equivalent data for the Euclidean metric. Here, the results are clearer, with almost all models trained using individual optimization (orange and red) having higher values than the models trained using Bagdasaryan training (blue and green models). Bagdasaryan training thus performs significantly better here.

For the main-task accuracy, shown in Figure 6.23, we see a similar situation to the evaluation of the main-task accuracy in most previous experiments, with no significant difference caused by a different choice of approach. This can be seen by all benign models taking roughly identical main-task accuracy values. The same is true for the adversarial models, while they reach slightly lower values than the benign models.

Figure 6.24 shows the backdoor accuracy reached by the models. As before, a higher value here is beneficial to an adversary, while defenders desire lower backdoor accuracies. In this experiment, the individual optimization approach leads to slightly higher backdoor accuracy values than Bagdasaryan training, as can be seen from almost all orange models reaching a higher backdoor accuracy than the best blue model. While this means that the individual optimization approach would be better for the adversary, the same situation seen previously occurs: the adversary can't switch to this approach, since the worsened

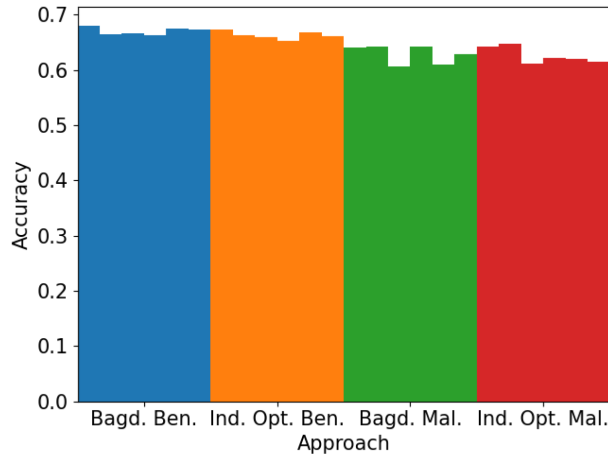

Figure 6.23: Main-task accuracy for local models trained with both approaches. No significant difference is visible between Bagdasaryan training and Individual Optimization. Higher values are better.

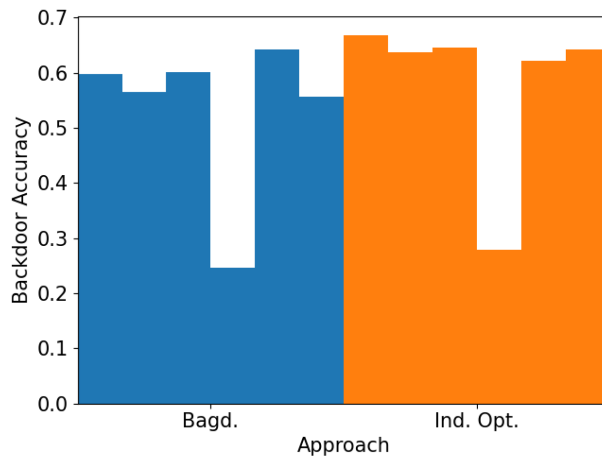

Figure 6.24: The backdoor accuracy of local models trained with both approaches. Individual Optimization reaches slightly better backdoor accuracy values. Higher values are better for the adversary.

adaption would allow a DF mechanism to detect the adversarial models.

Figure 6.25 shows the time required to train a single epoch using both approaches. Lower values are better since they correspond to a faster training process. Here, we see that Bagdasaryan Training requires, on average, roughly 5 seconds to train each epoch, while Individual Optimization requires roughly 5.5 seconds. This increase in training time of roughly 10% would be acceptable if Individual Optimization performed better for constraining the metrics. For setups where a larger number of metrics are constrained, we expect the training time required for Individual Optimization to increase more rapidly than that for Bagdasaryan Training since Individual Optimization involves more resource-intensive steps (forward pass, backward pass, optimization steps) than Bagdasaryan Training.

This leads to the conclusion that the Bagdasaryan Training approach works best for constraining the Euclidean and Cosine metrics. While individual optimization was not able to perform as well as the Bagdasaryan approach, it might still be useful for other areas of ML, as a method of performing MTL. Future work could also focus on finding ways of improving the adaption reached when using individual optimization, which could surpass the adaption reached when using Bagdasaryan training. This completes the answer to

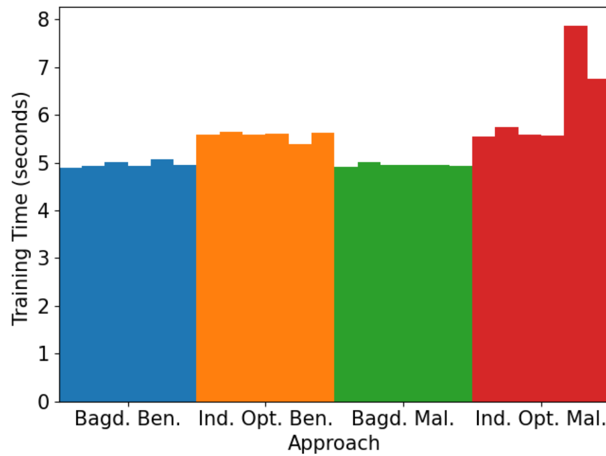

Figure 6.25: Time required to train one epoch under both approaches. Lower values are better. Bagdasaryan Training trains roughly 0.5 seconds faster than Individual Optimization. There is no significant difference between benign and malicious training.

RQ1 by showing that Bagdasaryan Training should be used.

6.4 Effectiveness of the Constraint System

We start this evaluation with unconstrained training, where adversarial models can often be detected due to some metric taking an unusual value.

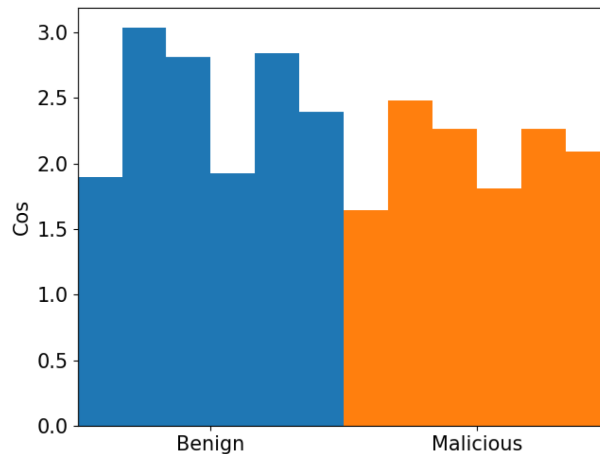

Figure 6.26: Cosine values of benign and malicious local models trained without constraints.

Figure 6.26 shows the cosine metric values taken by benign and malicious models. Higher or lower values here are not inherently better or worse, but significantly different values can be a sign of a malicious model, which could be used for detection by a DF approach. In Figure 6.26, we see no such outlier. This indicates that defense mechanism would be unable to detect adversarial models based in the Cosine values here.

Figure 6.27 shows the Euclidean metric values for the same experiment. Here, the fourth adversarial model has a significantly different value for the Euclidean metric. It would thus be filtered out by a DF mechanism.

To prevent the detection, Bagdasaryan et al. [17] started constraining metric values, with the goal of preventing defense mechanisms from being able to detect the backdoored mod-

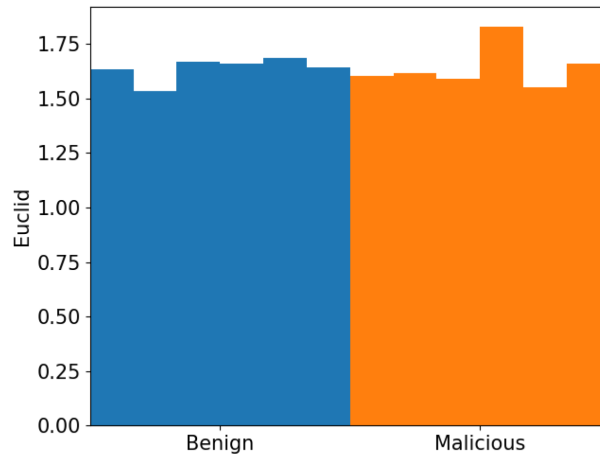

Figure 6.27: Euclid values of benign and malicious local models trained without constraints. While most adversarial values are similar to benign ones, the fourth value is significantly different.

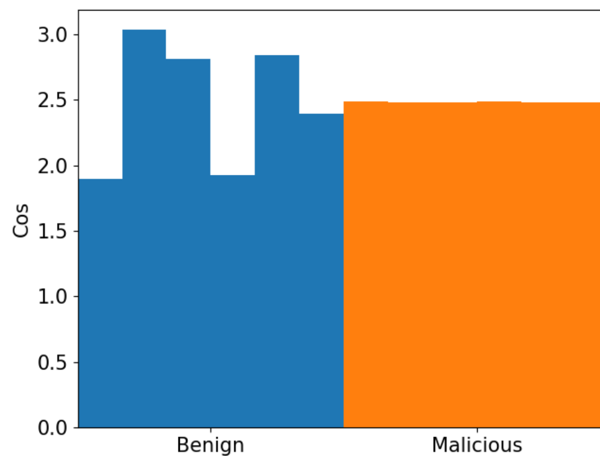

Figure 6.28: Cosine values of unconstrained benign local models and constrained adversarial local models. The cosine values for the adversarial models are almost identical and similar to the benign ones.

els. An example is shown in Figure 6.28. Here, the cosine metric values for all malicious models are almost identical and take similar values to those for the benign models. There are thus no outliers anymore and a defense mechanism would be unable to detect the models.

Our approach now performs the same process on the benign side. In the same way the adversary constrains the model's metrics during adversarial training, we constrain the training of the benign models. Figure 6.29 shows cosine metric values for this setup. Due to the smaller scale, we now again see significant differences between the values for the individual models. When comparing the benign to the adversarial models, we see no difference that would allow a DF mechanism to detect adversarial models, i.e., the adversarial side adapted better to the constraints than the benign side and there are no outliers in cosine metric value on the adversarial side. Figure 6.30 shows the same data for the Euclidean metric. Here, we see some outliers on the adversarial side (orange models 3 and 5). A DF mechanism might thus be able to filter these out due to our mechanism.

As shown in Chapter 4.5.1, training a model with a strong backdoor might require scaling the final model update. When performing this scaling process, the metric values change,

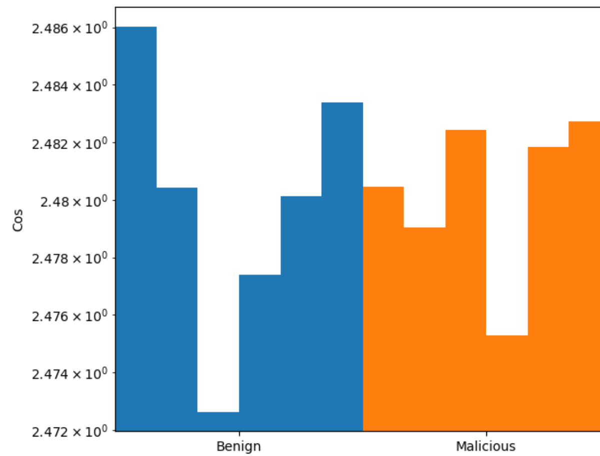

Figure 6.29: Cosine values of benign and malicious local models trained with constraints.

since the weights of the model change. This means that, if the adversary does not adapt the metric targets, the models will have significantly different metric values, making detection trivial, as seen in Figure 6.31. Here, clustering the models into "benign" and "malicious" would be trivially possible.

The adversary must thus change the targeted metric values to respect the scaling factor. For the Euclidean, Min, and Max metrics, this is trivial: the correct target value can be calculated by dividing the target value after scaling by the scaling factor. The Count metric does not need to be changed. For the Cosine metric, the value can be determined empirically, by training the model as usual, scaling it, determining the average Cosine value, and dividing this value by the desired target value.

Figure 6.32 shows the cosine metric values for models trained with adjusted target values. Here, detecting the adversarial models would be easy, as all cosine metric values for malicious models are smaller than those for benign models. The adversary might be able to improve this, by manually compensating for this effect, but even then, the larger range of values (compare, e.g., values for the second and fifth malicious model) would allow a DF mechanism to detect some malicious models. This larger range is caused by scaling itself since the increase in metric values also spreads them further apart. Our system can thus improve the detection of adversarial FL, when the adversary has to perform scaling as part of the attack. This a partial answer to **RQ5**.

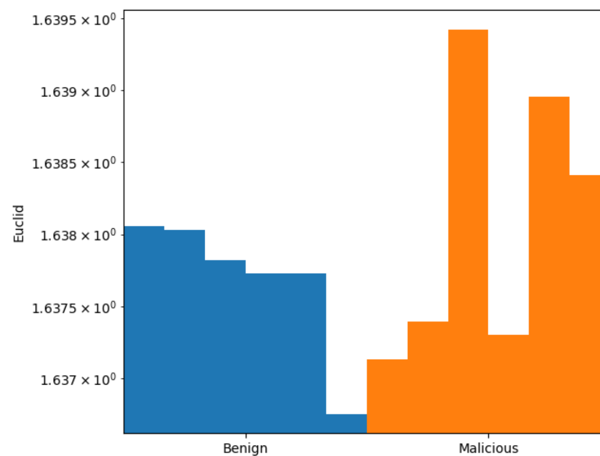

Figure 6.30: Euclid values of benign and malicious local models trained with constraints.

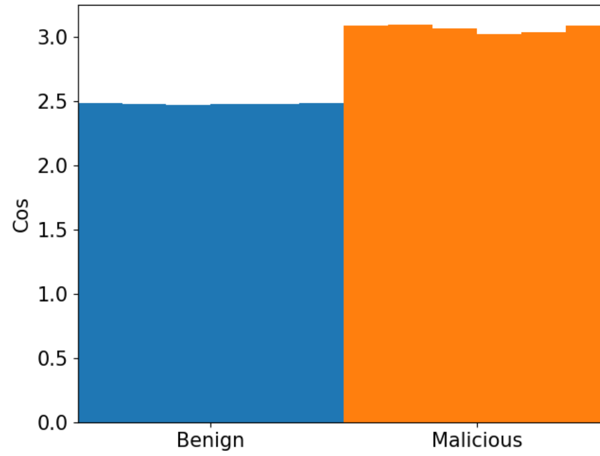

Figure 6.31: Cosine values of benign and scaled malicious local models trained with constraints. The values for the malicious are consistently higher than those for the benign models.

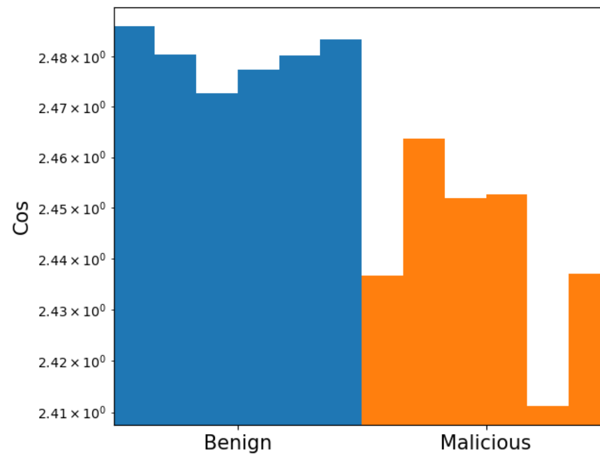

Figure 6.32: Cosine values of benign and scaled malicious local models trained with constraints and adjusted target values. Even with correct constraint targets, scaling causes the metric values for malicious models to differ from those for benign models.

6.4.1 Training Time Required for Constraints

When training the constraints, more calculations are required for each epoch that is being trained. In Bagdasaryan Training, this is largely the calculation of the metric values. For Individual Optimizations, this would also have included additional optimization steps, forward passes, and backward passes. This means that our constraint mechanism increases the training time required. To evaluate the effect of the constraint mechanism on the training time, we compare the training time for unconstrained and constrained training of one epoch. Figure 6.33 shows that constrained training takes a significantly longer time than unconstrained training, with the time required for a constrained epoch being roughly five times larger than the time required for an unconstrained epoch. This means that our system imposes a significant performance penalty on the FL setup. Whether this prohibits using the system in a real-world application must be considered specifically for each use case. Further work could evaluate ways of speeding up this process, for example by improving the speed of the metric functions using hardware acceleration.

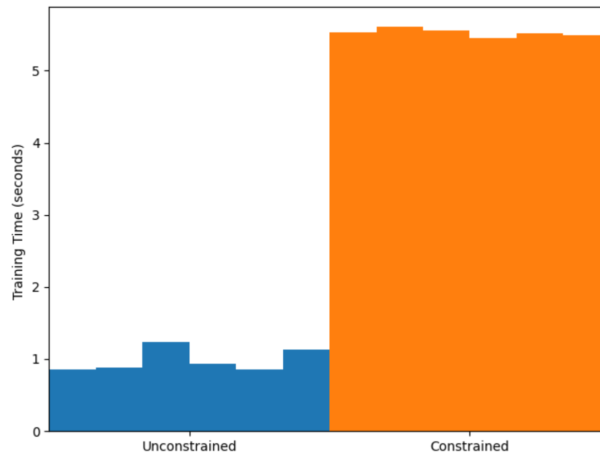

Figure 6.33: Training time required for one epoch of unconstrained and constrained training. Lower values are better. While unconstrained training takes roughly one second, constrained training requires 5 seconds.

6.4.2 Resilience of Backdoors Trained Under Constraints

RQ4 focuses on one important characteristic of a backdoor: its *resilience*, i.e., the number of rounds of training the backdoor remains effective while it is not being renewed by an adversary. This is relevant since it determines the cost and complexity of an attack. Cost, since the adversary does not have to perform as much training if a backdoor is effective for a longer time, and complexity since the adversary might not have to maintain permanent attack infrastructure if the backdoor is resilient enough. We evaluate this by training a backdoor into an FL model and determining how the backdoor accuracy behaves during the next epochs of benign training and comparing this behavior for constrained and unconstrained training. We start this evaluation by training an unconstrained global model using our standard setup. Next, the experiment is repeated under constrained training with an otherwise identical setup.

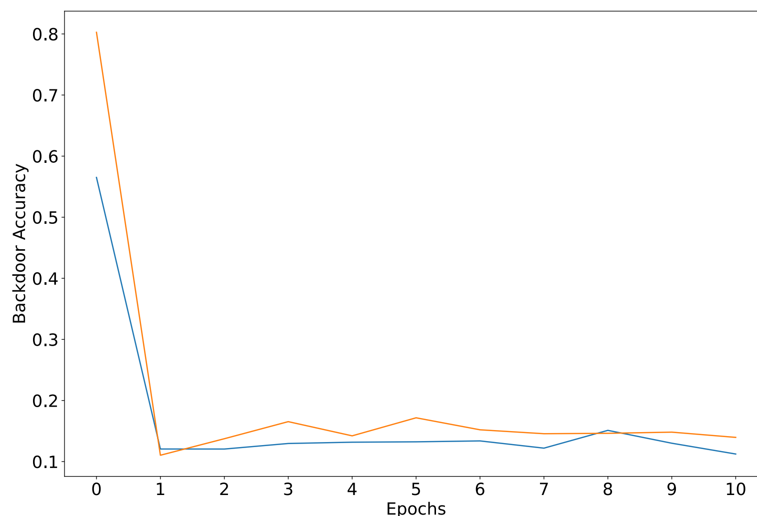

Figure 6.34: Backdoor accuracy of aggregated models with backdoors trained with and without constraints during benign training. Higher values are better for the adversary. The backdoor accuracy falls to irrelevant levels immediately.

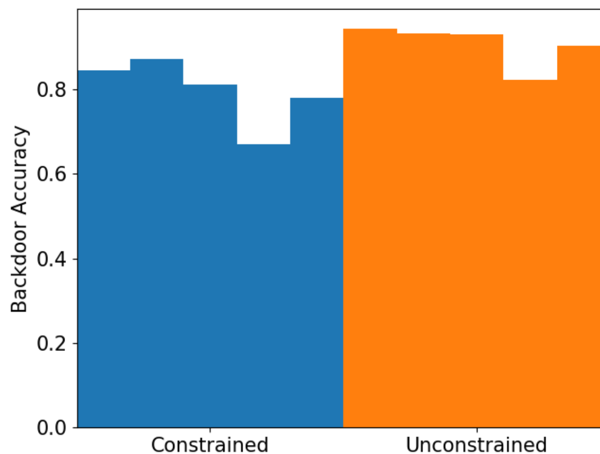

Figure 6.35: Backdoor accuracy of local models with backdoors trained with and without constraints. Higher values are better for the adversary. Constrained models reach moderately worse backdoor accuracies than unconstrained models.

As Figure 6.34 shows, both in the constrained and in the unconstrained model, the backdoor fails to achieve any sort of resilience: the backdoor accuracy rapidly lowers to insignificant levels for both models and does not significantly change from there. This means that we can not make any reasonable statements about the backdoor's resilience. As an answer to **RQ4** this leaves the fact that neither constrained nor unconstrained backdoors are resilient in our scenario. For **RQ3**, we have shown here that backdoors trained under constraints are effective, albeit less than normal backdoors. This could be worked around using scaling.

Figure 6.35 shows the local backdoor accuracy of the models trained in this experiment. Here, we see that constrained training also reduces the accuracy of the backdoors before aggregation. This was expected from the evaluation of the aggregated backdoors, where the constrained models also performed worse. This answers **RQ2**: the backdoor accuracy of local models decreases due to the constraints, but is still relatively high.

6.5 Different Setups

The experiments in this section explore the influence of various hyper-parameters on the effectiveness of the constraint system. Some of these are hyper-parameters that can be changed arbitrarily, e.g., learning rate or batch size, while others are intrinsic properties of the system or of each client, like the distribution of the dataset. For the hyper-parameters that can be changed, we can also differentiate between those that are determined as part of the FL setup and cannot be changed for each client, e.g., the model, and those that can be changed by each user, e.g., the learning rate. These could allow an adversary to change a hyper-parameter from the intended setup to lead to improved results for adversarial training.

6.5.1 Distributions

In contrast to many other hyper-parameters, e.g., learning rate, batch size, or optimizer, the distribution of the dataset is an intrinsic part of each ML scenario that can not be changed by the implementor in real-world scenarios. Especially in FL, distributions can be chaotic, causing the simplicity of a mathematical approximation (e.g., one-class non-IID with a fixed parameter) to be too limiting, for example, if the training data distributions of different participants differ not only in some parameters but also in fundamental shape.

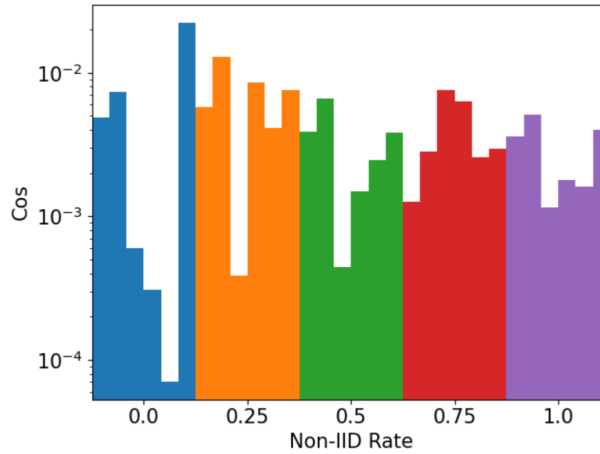

Figure 6.36: The absolute difference between targeted and measured values for the Cosine metric of local models with different non-IID rates. Lower values are better. The data shows that higher non-IID rates do not lead to a better adaption, but do lower the range of values for each configuration.

While a larger study of more realistic distributions is out of scope here, we can gain some insight into the behavior of our system by performing some experiments with different distributions. For this experiment, we train models with different non-IID rates and evaluate whether this influences the the adaption to the constraints. In contrast to most other experiments, we do not perform this evaluation with the goal of determining a "best" distribution - since the distribution cannot be changed in the real world. The evaluation is merely supposed to gain some intuition into the behavior of the system under different distributions.

Figure 6.36 shows the absolute difference between the targeted and measured values for the cosine metric. Here, smaller values correspond to a better adaption. It shows that the adaption quality is not significantly influenced by the non-IID rate since the average value for each configuration is roughly similar. There is, however, a significant influence on the range of values: for higher non-IID rates, the distance between the largest and smallest value (e.g., second and third model for the last configuration) is significantly smaller than for lower non-IID rates (e.g., fifth and sixth model for the first configuration). This is expected, since larger non-IID rates mean that the datasets for the individual models are more similar, leading to the trained models being more similar to each other.

6.5.2 Learning Rate

In ML, the learning rate of a system corresponds to the speed with which a model is trained. A higher learning rate typically leads to faster improvements, but also to more unstable training, ultimately often ending in worse accuracy.

During adversarial training, a larger learning rate is often chosen, since this allows for quicker training of a backdoor (at the cost of a worsened main-task accuracy). Since the learning rate has a significant influence on the development of the weights in the network and these weights are being used to determine the metrics for our constraints, we evaluate whether it has a significant influence on our constraint system by comparing the main-task accuracy, adaption quality, and backdoor accuracy of training with learning rates 0.1, 0.01, 0.001, and 0.0001. Since each client can choose the learning rate freely, we also evaluate whether there is potential for an adversary to gain an advantage by using a different learning rate.

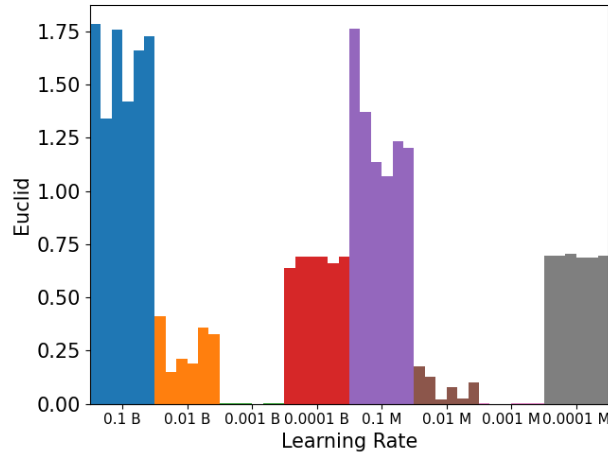

Figure 6.37: The absolute difference between targeted and measured Euclid values of local models trained with different learning rates. Lower values are better. Benign models are marked `B`, while malicious models are marked `M`. Learning rate 0.001 leads to the best results.

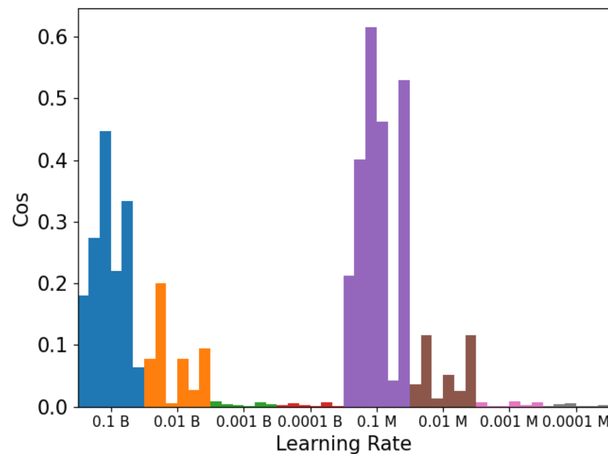

Figure 6.38: The absolute difference between measured and targeted Cos values of local models trained with different learning rates. Benign models are marked `B`, while malicious models are marked `M`. The smallest learning rates lead to the best results (i.e., lowest values).

Figure 6.37 shows the absolute difference between measured and targeted values of the Euclidean metric for different learning rates. Here, smaller values are desired, since those correspond to a better adaption. The data shows that adaption works best for a learning rate of 0.001 (i.e., green and pink models). The models for all other learning rates only reach values higher than roughly twice these values.

Figure 6.38 shows the same data for the cosine metric. Here, we see similar results as before, with the learning rate of 0.001 reaching the lowest values. In contrast to the evaluation of the Euclidean metric, the lower learning rate of 0.0001 reaches values roughly identical to these. This indicates that the lower learning rate would also be suitable when constraining only the Cosine metric.

Figure 6.39 shows the main-task accuracy for different learning rates. We see that the best values are achieved for the two lowest learning rates, where the values are indistinguishable on the benign side. On the malicious side, the values are slightly higher for the lowest learning rate. For the largest learning rate, the results are significantly worse, being

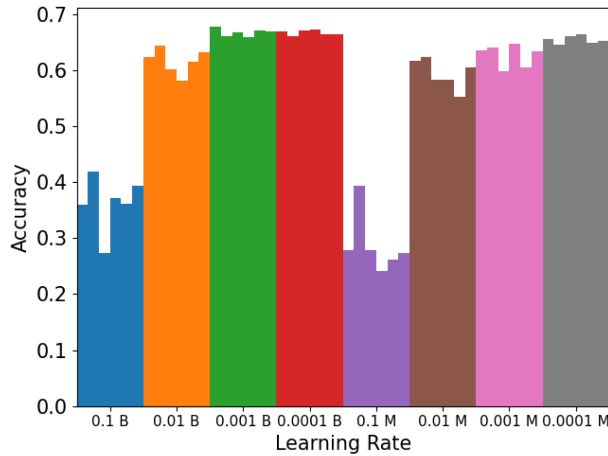

Figure 6.39: Main-task accuracy of local models trained with different learning rates. Higher values are better. Benign models are marked `B`, while malicious models are marked `M`.

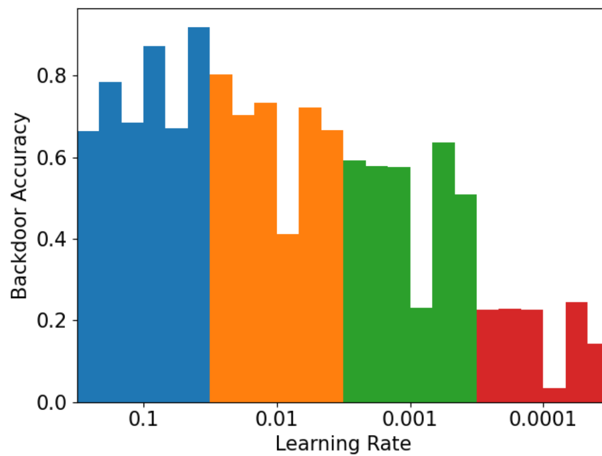

Figure 6.40: Backdoor accuracy of local models trained with different learning rates. Higher values are better for the adversary.

roughly 50% lower than for the best learning rates. For the backdoor accuracy, as shown in Figure 6.40, the best values are achieved with the highest learning rate, with the values decreasing the lower the learning rate gets. As we have seen before, this puts the adversary in a suboptimal situation: one configuration (i.e., learning rate 0.001, green models) would lead to a good adaption, while a different configuration (i.e., learning rate 0.1, blue models) would be required to train a strong backdoor. This evaluation confirms our previous choice of 0.001 as the standard learning rate to use for our experiments. When only constraining the Cosine metric, a lower learning rate could also be chosen.

6.5.3 Batch Size

The batch size determines how many samples are trained in the network simultaneously. Larger batch sizes lead to faster training, at the cost of less improvement for each sample.

The adversary is also able to change the batch size, so might be able to gain an advantage this way. This leads us to evaluate the influence of different batch sizes on the adaption quality, while also considering the backdoor and main-task accuracy. To evaluate the influence of the batch size on our constraint system, we train benign and malicious models with batch sizes 32, 64, and 128 and compare the results.

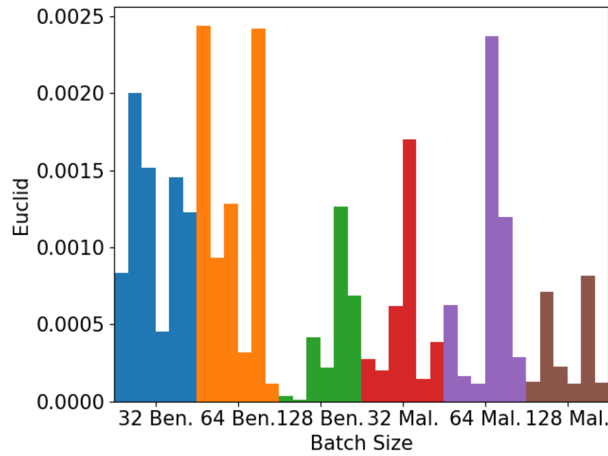

Figure 6.41: The absolute difference between measured and targeted Euclid values of local models trained with different batch sizes. Lower values are better. The largest batch size (128) leads to the best results (i.e., the lowest values).

Figure 6.41 shows the absolute difference between the measured and targeted values for the Euclidean metric. Here, we see that the values for the largest batch size (128, green and brown models) are significantly lower than those for the other batch sizes. This means that this batch is best suited for adapting to the constraints. This is contrary to the configuration of our standard experimental setup, where a batch size of 64 is used. This highlights an interesting property of our system: there is no inherent benefit to a better adaption. While we have now seen that the batch size of 64 did not lead to the best results, it was perfectly adequate as long as both benign and adversarial were trained with this batch size. Only if the adversary trains with a configuration that leads to better results than the configuration used by the benign clients can the system be circumvented. Once the benign participants learn of the advantage the adversary has, due to some better configuration, the benign side can adopt this configuration, eliminating the adversary's advantage. This makes our system robust against improvements to the adaption quality. It is only important that the setup with the best-known adaption quality is used.

Figure 6.42 shows the same data for the cosine metric. Here, the situation is slightly different, with the smallest batch size (32, blue values) reaching the best values on the

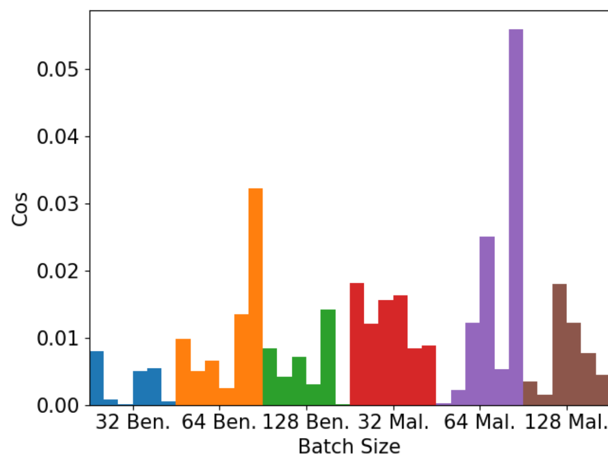

Figure 6.42: The absolute difference between measured and targeted Cos values of local models trained with different batch sizes. Lower values are better. The smallest batch size (32) leads to the best results (i.e., the lowest values).

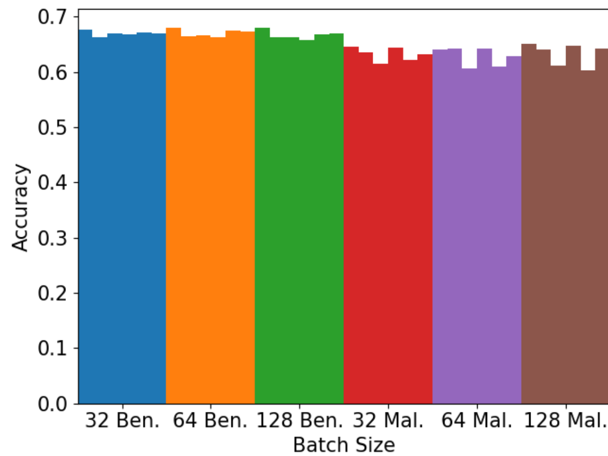

Figure 6.43: Main-task accuracy of local models trained with different batch sizes. Higher values are better.

benign side. As Figure 6.43 shows, the choice of batch size does not have any noticeable influence on the main-task accuracy achieved by the model. For the backdoor accuracy, shown in Figure 6.44, we see no significant influence by the batch size. All models, not accounting for the outliers, reach roughly similar values.

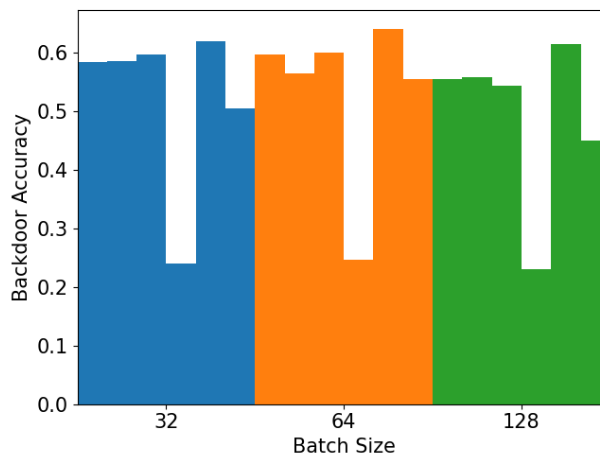

Figure 6.44: Backdoor accuracy of local models trained with different batch sizes. Higher values are better for the adversary.

For the batch size, our evaluation has thus shown, that a larger batch size can be advantageous for adapting to some metrics. Whether this applies to the concrete metrics used in a setup would have to be evaluated individually. We have not seen an influence on the backdoor accuracy by the batch size.

6.5.4 Optimizer

Optimizers are one of the central parts of each ML setup. While SGD is most commonly used, modern alternatives like ADAM [54] often lead to better results. Since ADAM uses features like momentum, it can be assumed to behave differently than SGD when training our constraints. We evaluate the influence of the ADAM optimizer on benign and adversarial constrained training. As before, our evaluation also focuses on any advantages the different optimizers could have for the adversary. We evaluate different optimizers by comparing the adaption reached by SGD with the adaption reached by Adam.

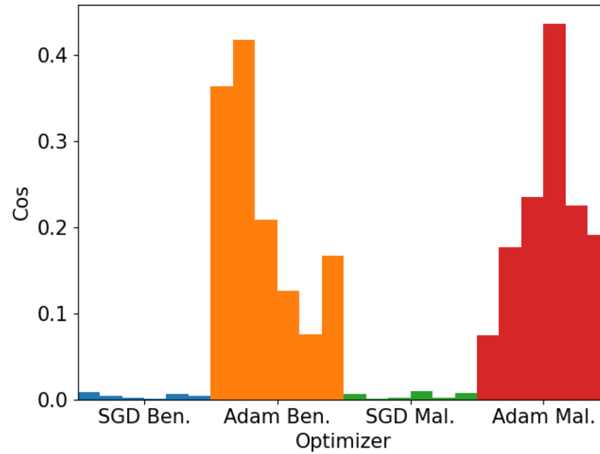

Figure 6.45: The absolute difference between targeted and measured Cos metric values of local models trained with SGD and Adam optimizers. Lower values are better. The models trained by SGD reach significantly better values.

Figure 6.45 shows the absolute difference between measured and targeted values for the Cosine metric. Here, we see that all values for models trained using the SGD optimizer (blue and green models) are significantly smaller than the values for the best models trained using Adam. This indicates that the Adam optimizer is not suited for our constraint mechanism.

For the main-task accuracy, shown in Figure 6.46, we see that the models trained using the SGD optimizer (blue and green models) consistently reach significantly higher main-task accuracy values than the models trained with the Adam optimizer. Figure 6.47 shows that the models trained using the Adam optimizer reach a significantly higher backdoor accuracy than the models trained using the SGD optimizer. While this would be an advantage to the adversary, it is not a feasible option, since the models would not adapt to the constraints, as seen previously. The analysis of the optimizer leads to the conclusion that SGD works well for the constraint system, while Adam does not. Other optimizers would have to be evaluated separately.

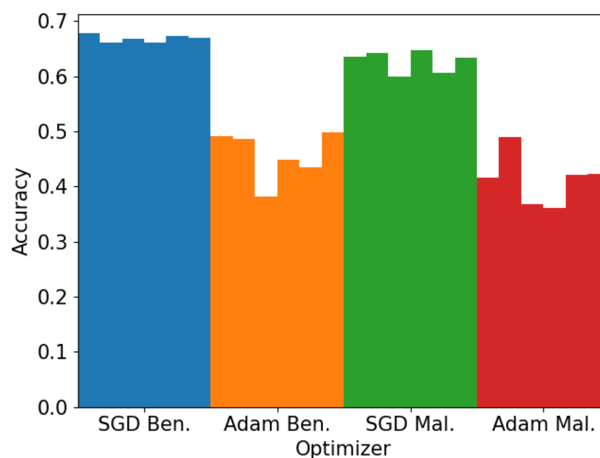

Figure 6.46: Main-task accuracy of local models trained with SGD and Adam optimizers. Higher values are better. The models trained using SGD reach better main-task accuracy values.

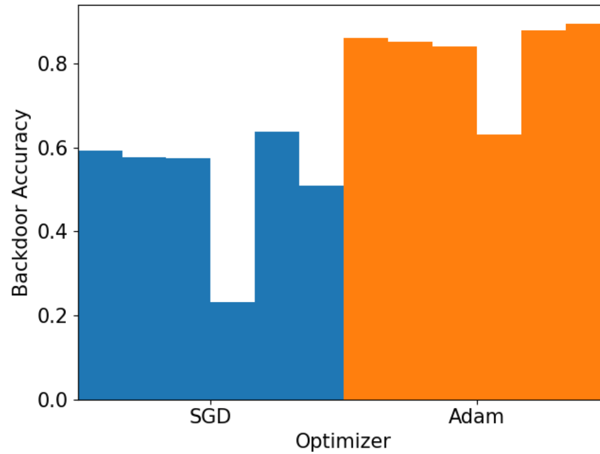

Figure 6.47: Backdoor accuracy of local models trained with SGD and Adam optimizers. Higher values are better for the adversary. The models trained using the Adam optimizer reach significantly higher backdoor accuracies.

6.5.5 Target Values

One significant aspect of constraining the metrics is which value the metrics should be constrained to. So far, we have used the average of the values the metrics reach without any constraints. These values are shown in Table 6.5. Since these values would require the smallest change in each individual value, they would need the least "effort", which might lead to the best results at the lowest cost to the other objectives. In a real application of our system, the median of the values would be used, since this ensures benign values under our threat model.

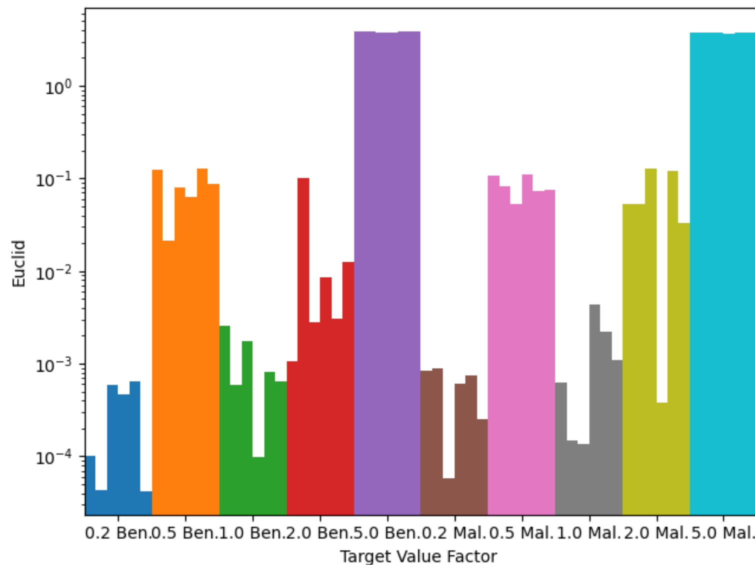

Figure 6.48: Absolute difference of measured and targeted Euclid values of local models trained with different constraint target values. Lower values are better.

On the other hand, constraining to different values can lead to a situation where a benign model can still adapt to the constraints while keeping a good main-task accuracy, but the adversary cannot do so, while also training a backdoor into the model. We thus determine the influence of the target values on the adaption quality, backdoor accuracy, and main-task accuracy experimentally for benign and malicious training. We base our evaluation of the target values for the constraints on the natural metric values we have determined

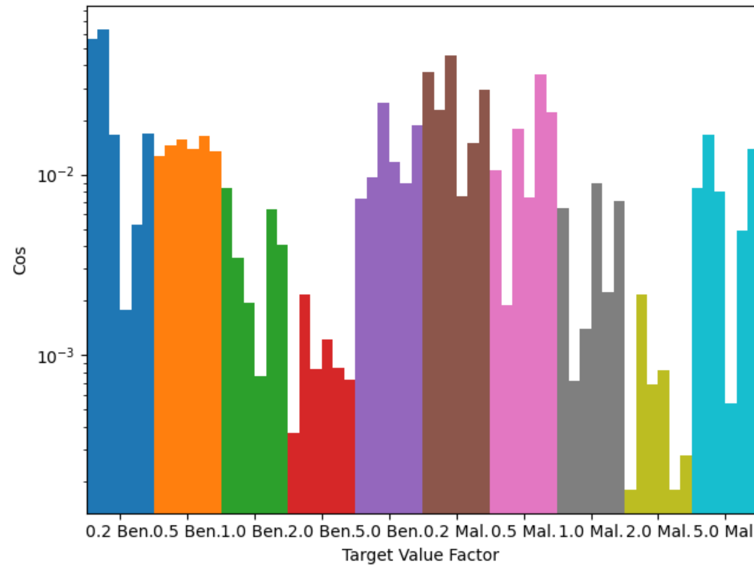

Figure 6.49: Absolute difference of measured and targeted Cos values of local models trained with different constraint target values. Lower values are better.

previously. To evaluate the influence of the target values, we train benign and adversarial models with target values based on multiples of these values. The multiples are 0.2, 0.5, 1.0, 2.0, and 5.0.

Figure 6.48 shows the absolute difference between measured and targeted values for the Euclidean metric. Here, lower values are desired, since they correspond to a good adaption. We see that the lowest values are reached for the lowest target values (0.2, blue models), with the natural target values (1.0, green models) reaching the second-best values. 0.5 (orange) and 2.0 (red) perform worse, with 5.0 (violet) performing worst.

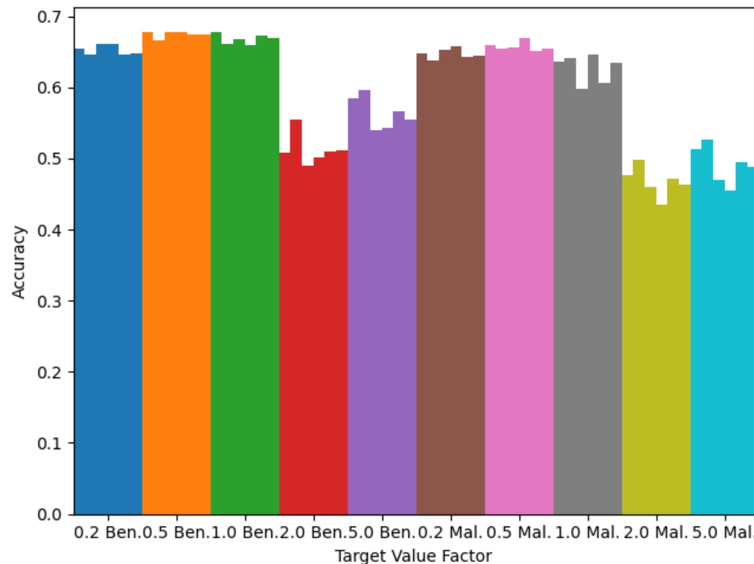

Figure 6.50: Main-task accuracy of local models trained with different constraint target values. Higher values are better.

Figure 6.49 shows the same for the Cosine metric. Here, we see that the best adaption is reached by multiplier 2.0 (red models); Multiplier 1.0 again reaches the second-best values. 0.2, which performed best for the Euclidean metric, performs by far worst here.

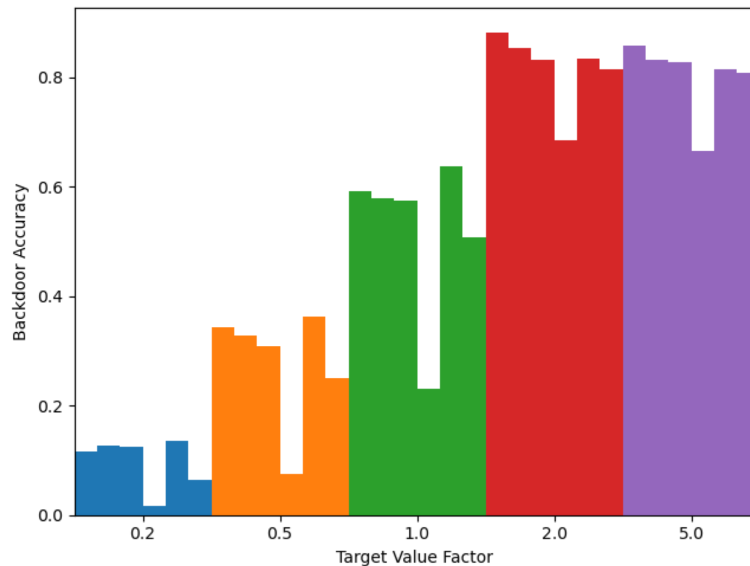

Figure 6.51: Backdoor accuracy of local models trained with different constraint target values. Higher values are better for the adversary.

This indicates that our choice of using the natural metric values (i.e., multiplier 1.0) in our system was ideal.

For the main-task accuracy, shown in Figure 6.50, we see no significant difference between the models trained with constraints targeted to values smaller than or equal to the natural target values. For larger values (i.e., red and violet models on the benign side), the main-task accuracy sinks significantly.

For the backdoor accuracy, shown in Figure 6.51, we observe a correlation between higher target values and higher backdoor accuracy. For multiplier 0.2, no significant backdoor accuracy is reached, while multiplied 2.0 can reach a strong backdoor. Since the target value of the constraints is a central property of the entire FL system when using our approach and the adversary cannot change it, this is not an effect the adversary can abuse. This leads us to the conclusion that a target value equal to the metric's average natural value is ideal for constraining the metrics.

6.5.6 Momentum

Momentum is an extension to the SGD optimizer that is often used to achieve improved results. Since momentum has a significant influence on the optimization process, it also influences the values taken by the metrics during constraining. We evaluate this by training models with and without momentum and evaluating the influence this change has on the adaption quality, main-task accuracy, and backdoor accuracy. For this experiment, it is relevant to use different metric target values for each configuration, i.e., the values determined in Table 6.5 for the respective momentum value.

Figure 6.52 shows the absolute difference between the targeted and measured values for the Euclidean metric when training with and without momentum. Low values correspond to a good adaption here. It is apparent that a better adaption is reached when using momentum, with the values shown for models without momentum (orange and red) being more than twice larger than values for models with momentum.

Figure 6.53 shows the same for the Cosine metric. Here, we again see that the models trained with momentum adapt better to the constraints than the models trained without momentum. On the benign side, the difference in adaption quality between the models

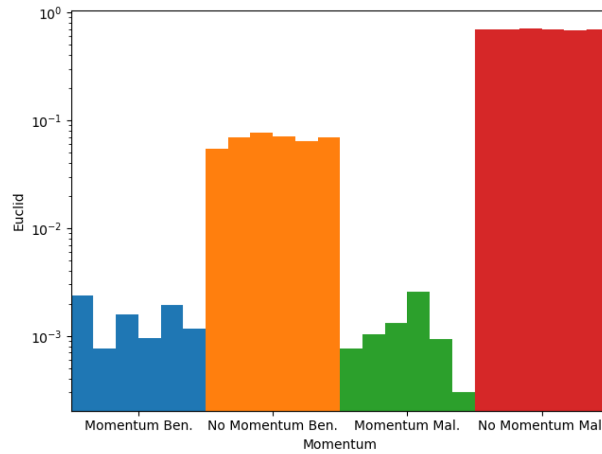

Figure 6.52: The absolute difference between targeted and measured Euclid values of local models trained with and without momentum. Lower values are better.

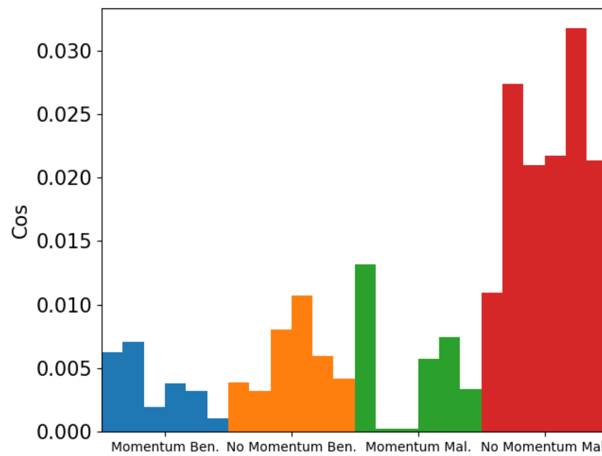

Figure 6.53: The absolute difference between targeted and measured Cos values of local models trained with and without momentum. Lower values are better.

trained with momentum and the models trained without momentum is much smaller than for the Euclidean metric.

For the main-task accuracy, shown in Figure 6.54, we see no significant influence caused by the usage of momentum. The backdoor accuracy, shown in Figure 6.55, is significantly reduced when training without momentum.

This data indicates that in general, momentum is required both for achieving good constraints and for reaching a high backdoor accuracy. Considering the relatively good adaption of the Cosine metric for benign training without momentum and the problems adversarial training faces when not using momentum, interesting results might be achievable if the adversary can be forced to train without momentum.

This can be done by performing benign training without momentum, with the correct constraint target values. Since the adversary then, when using momentum, has to constrain to a different value than the natural value, the adaption might be worsened. If, on the other hand, the adversary chooses not to use momentum, the backdoor accuracy would suffer significantly. Here, the results of the previous experiment, indicating that smaller target values lead to worse backdoors, are relevant. This would put the adversary into a dilemma. We evaluate this dilemma by repeating the last experiment, with the target values determined for training without momentum.

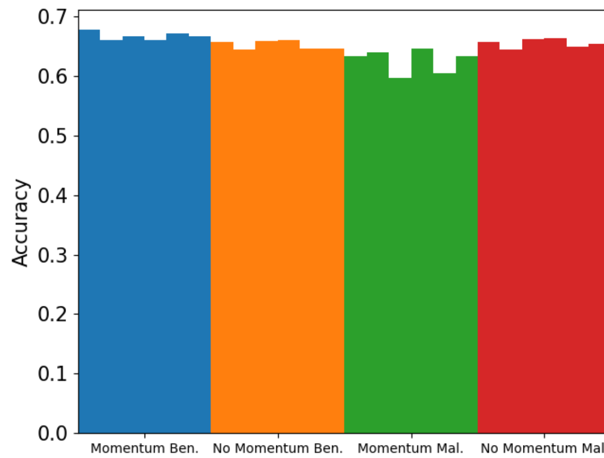

Figure 6.54: Main-task accuracy of local models trained with and without momentum. Higher values are better.

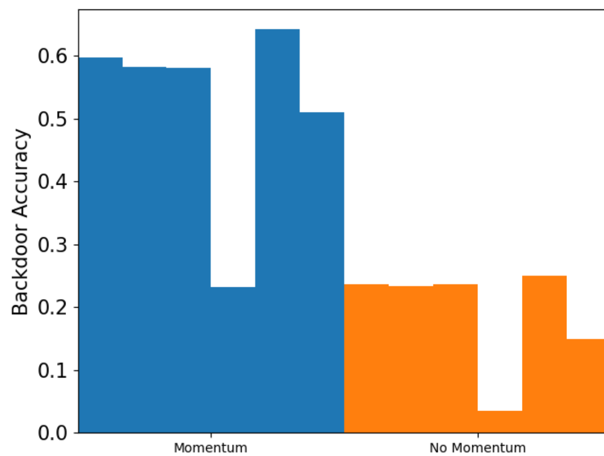

Figure 6.55: Backdoor accuracy of local models trained with and without momentum. Higher values are better for the adversary.

Figure 6.57 shows the backdoor accuracy reached by the adversary in this configuration, with and without momentum. Both values are significantly lower than usual, due to the different constraint target values (see 6.5.5). When training without momentum, the backdoor accuracy is even worse, as we have determined that momentum is important for reaching a strong backdoor.

Figure 6.56 shows the absolute difference between measured and targeted values for the Cosine metric. Here, our main focus is on the benign models without momentum. This is the configuration that benign models would use in this setup. The adversary now has the choice between training without momentum (red models), reaching a good adaption and thus passing the detection - with a weak backdoor (see Figure 6.57), or training with momentum (green models), reaching a better backdoor, but not passing the detection mechanism due to the significantly worse adaption. This puts the adversary into a strong dilemma, with no feasible way of inserting a useful backdoor in the model. This answers **RQ5**: by using this approach, a defense mechanism like a clustering defense would achieve improved results. We have shown that our system is effective for preventing an adversary from inserting a backdoor in a FL environment. To work around this, the adversary would need to find a way of constraining the malicious model being trained with momentum, which is not trivial, as our experiments has shown, or find a way of achieving a high

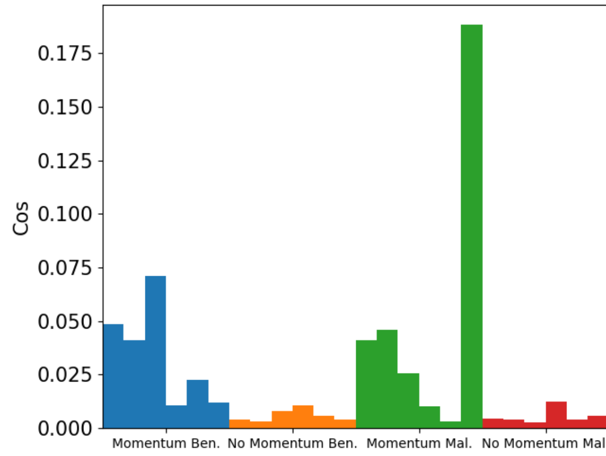

Figure 6.56: The absolute difference between targeted and measured cosine values of local models with and without momentum, for the target values of training without momentum. Lower values are better.

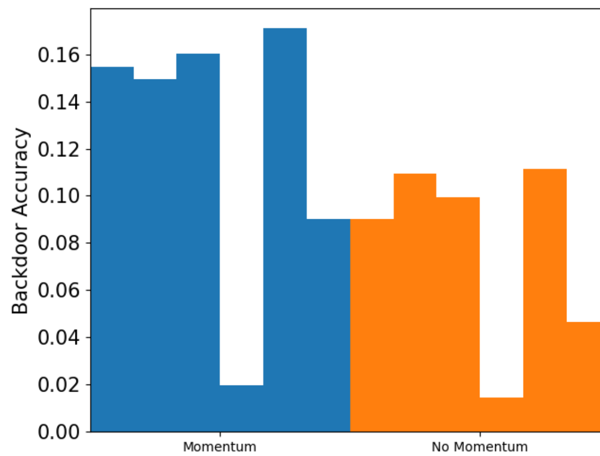

Figure 6.57: Backdoor accuracy of local models trained with and without momentum, for the target values of training without momentum. Higher values are better for the adversary.

backdoor while training without momentum.

6.5.7 Datasets

CIFAR10 [29] is just one of many datasets that might be used for training a model. In the real world, training would likely not be done on such a predetermined, well-known dataset at all, but, for example, on images as they are processed by a camera. While we cannot evaluate our mechanism on such a live dataset, the dataset used for our experiments can be changed with a dataset like MNIST [28] to at least gain some insight into the behavior of the approach when using a different dataset. As with the evaluation of different distributions, the goal of this evaluation is not to determine which dataset to use, but merely to gain some knowledge about the behavior of the model when using a different dataset. The model is adjusted as needed for the new dataset and for this experiment, the model is only constrained in the Euclidean metric.

Figure 6.58 shows the values of the Euclidean metrics when training on the MNIST [28] dataset. As the data shows, the models manage to adapt to the constraints in the Euclidean metrics, though not as well as in previous experiments, as seen by the larger fluctuations

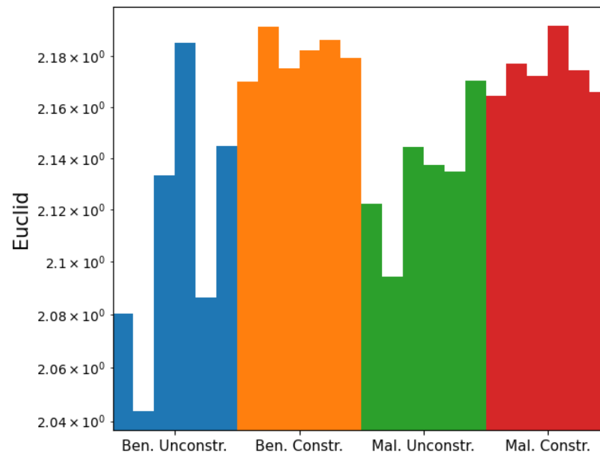

Figure 6.58: Euclidean values for local models trained on the MNIST [28] dataset. Euclidean values can be constrained when training on the MNIST dataset, though they do not adapt as well as when using the Cifar10 [29] dataset.

in the orange and red values. Especially on the malicious side, the constraints do not seem to limit the values of the metric significantly. This could be an intrinsic property of the MNIST [28] dataset, or due to our previously well-working configuration not being ideal here. Future work could look into determining how the ideal configuration needs to be changed for improvements here.

6.5.8 Models

Similarly to our evaluation of the influence of other hyper-parameters, we must consider the influence a different model has on the effectiveness of our defense system. To evaluate this, we replace the model with a SqueezeNet. We then evaluate the effectiveness of our system by considering the adaption quality, the achieved main-task accuracy, and the backdoor accuracy.

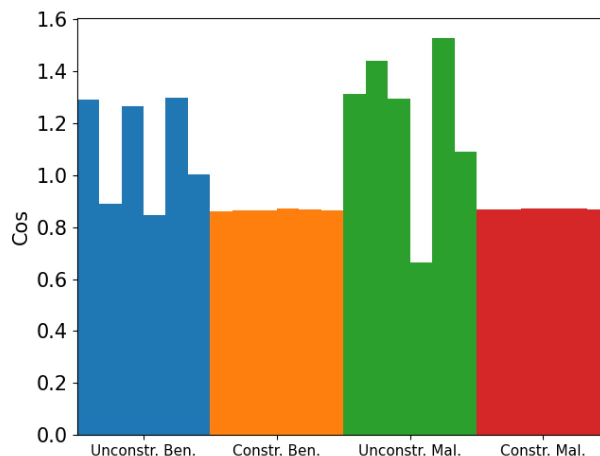

Figure 6.59: Cosine values of local SqueezeNet models.

In our categorization of hyper-parameters between ”operator-controllable” and ”predetermined”, the model takes a special place: While choosing a specific model because it performs better for our defense mechanism is possible, it’s likely undesirable, since more important characteristics, for example, size, training time, and reachable accuracy, also depend on the choice of model. We should thus not assume that choosing a different

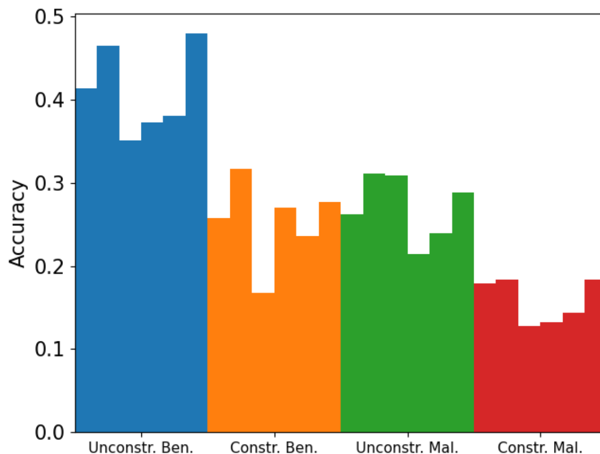

Figure 6.60: Main-task accuracy of local SqueezeNet models. Higher values are better.

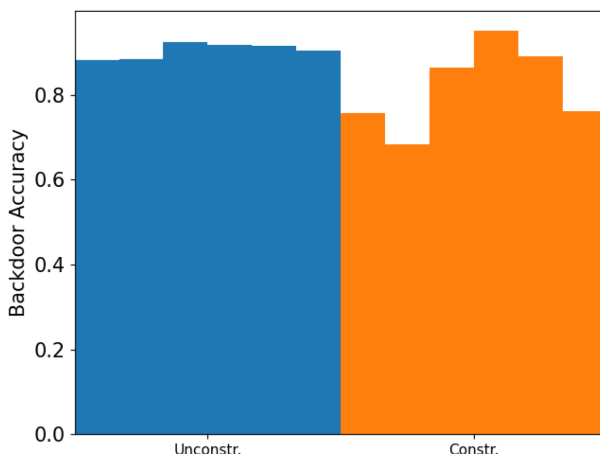

Figure 6.61: Backdoor accuracy of local SqueezeNet models. Higher values are better for the adversary.

model is a reasonable part of our defense mechanism. It is also not a choice that each client can make individually, so we do not have to consider specific advantages an attacker might gain. For this experiment, we have to change several hyper-parameters to achieve a realistic learning process. First, we raise the number of epochs to train 30. Then, the number of samples per epoch is doubled to 5120.

Figure 6.59 shows the values of the cosine metric for constrained and unconstrained benign and adversarial training. As the data shows, the metrics constrain well to their target value (see orange and red models). Figure 6.60 shows the main-task accuracy for constrained and unconstrained benign and adversarial training on a SqueezeNet. Here, we see that the constraints (orange and red models) cause a significant reduction in main-task accuracy. This could be due to the SqueezeNet model used here being significantly less trained before the experiment than the ResNet18 that was used previously. It could also be an intrinsic property of the SqueezeNet; Determining this would be future work. Similarly, evaluating the constraint system for any other model could be part of future work. For the backdoor accuracy, shown in Figure 6.61, we see a moderate reduction caused by the constraints.

6.6 Summary

In the last sections, many aspects of the constraint system have been evaluated. This section summarizes the most significant findings.

While the results from the baseline measurements were mainly intended to be helpful during other experiments, they have also shown that constrained models reach better accuracy values than unconstrained models, thereby partially answering **RQ1**. They have also indicated that the constraints might cause a minor improvement in the convergence speed of the models. This effect was expected, since intuitively, the constraints prevent the model from making larger, sudden changes, which leads it into a local minimum of the loss function faster. The evaluation of the different constraint approaches has shown that the best approach is Bagdasaryan Training with weighting to the minimum values once at the beginning of training and an α value roughly in a range of 0.2 to 0.6. This answers the rest of **RQ1**.

For **RQ2**, we have seen that constraints cause a penalty on the accuracy of the backdoors of local models. The same experiment also answered **RQ3**: backdoors trained under constraints are still effective, albeit less so, than unconstrained aggregated backdoors. For **RQ4**, we have seen that both constrained and unconstrained models lose their backdoors immediately after beginning benign training. We can thus not make any other useful statement on the influence of backdoors on the resilience of backdoors.

For **RQ5**, our experiments have shown that generally, the adversarial side can adapt to the constraints just as well, or better than, benign clients. This means that in this "basic" setup, our system fails to improve the detection capability of DF mechanisms. We have seen, however, two scenarios where the constraint system can be useful. First, when the attacker has to scale the model update, this worsens the adaption, giving the benign models an advantage. Second, the attacker can be tricked into a dilemma, where the choice is between successfully implanting a weak backdoor, or trying to implant a strong backdoor, but failing to adapt to the constraints. To do this, benign training must be performed without momentum. In this scenario, if the adversary also trains without momentum, implanting the backdoor fails, since momentum is required to reach a significant backdoor accuracy. This means that, in contrast to most other setups we have evaluated, where the adversary can reach a similar adaption than the benign models by also constraining the training, the adversary cannot reach a good backdoor here.

7. Discussion

In this chapter, we discuss open questions around the constraint system. We start with per-layer constraints, which apply the constraints to specific parts of the models, in Sect. 7.1. Then, we briefly consider how the system could behave in different areas of ML in Sect. 7.2. Next, Sect. 7.3 focuses on comparing our approach with existing defense mechanisms. Finally, Sect. 7.4 explains how specific choices of metrics to constrain might be beneficial, and Sect. 7.5 introduces some approaches for improving the training time required when training with constraints.

7.1 Per-Layer Constraints

Our system always constrains metrics across the entire model. We only chose a different approach than this most basic one during baseline measurements, where layers were constrained individually to achieve a better backdoor accuracy. Constraining layers individually was beneficial since it allowed the models to be more similar in ”internal structure”, resulting in a better backdoor after averaging. Individual layer constraints could also result in our system being improved since it would further limit the attacker’s ability to hide a backdoor in the model. In our current system, a metric in a malicious model could deviate significantly from the benign models in one layer, with a different layer’s metric deviating in the opposite direction, thus resulting in an identical metric for the complete model. By constraining each layer individually, this could be prevented. This approach might, however, lead to an increased performance penalty of our system even in benign training. Future work should evaluate the approach of constraining individual layers and the effect it has on benign and adversarial adaption capability and the cost it has. Further questions around layer constraints are whether all layers should be constrained equally or whether constraining just some layers could be enough. Since the first layers in a model are typically responsible for detecting broad structures in the input data, with only the last layers (typically a fully connected layer) performing the main part of the task, the effect of adversarial training would most likely be strongly focused on the last layer. This means that constraining only this layer could also lead to useful results, while not hindering accuracy improvements in the other layers. This would also prevent the problem of ”hiding” the backdoor in differences between metrics in individual layers, as described earlier since only the metric for a single layer is considered.

7.2 Other Areas of Machine Learning

Image classification is, of course, only one possible application of ML. Other areas deal with text, sound, videos, or other data. While we were able to show that our system leads to interesting results in one image classification setup, further research should also evaluate whether it is equally applicable to other such applications. We have seen that our system is effective because momentum is required for achieving a high backdoor accuracy. This behavior could be different when using a different model, which would typically be required when operating in a different domain. This further ties into our result that the adaption quality is highly dependent on the exact setup, meaning that for each scenario, the best adaption setup must be individually researched and developed.

7.3 Comparison with Other Approaches

The evaluation of our approach has focused only on this system, without considering other defense mechanisms. On the DF side, our system cannot be directly compared against existing mechanisms, since it is not a stand-alone defense mechanism, but rather extends other systems. Future work should investigate how our system can improve the results achieved by existing DF mechanisms. Nevertheless, the performance penalty, in terms of achieved accuracy and required training time, can be compared to existing DF and IR mechanisms to evaluate whether it is generally reasonable since this is hard to determine without values to compare it to.

7.4 Choice of Metrics

Our research has also not focused on the selection of the metrics to constrain, instead using metrics that are commonly used by existing defense mechanisms. By constraining the Cosine and Euclidean metrics in our experiments, we have chosen a metric measuring an "angular" property and a metric measuring a "distance", thereby restricting the model more than if we were using two more similar metrics. This approach generally seems beneficial. By analyzing the behavior of different metrics under constraints, improved results could be achieved. Future work could investigate whether different constraints (or a specific combination of constraints) can lead to improved results.

7.5 Improving Training Time

The evaluation has shown that for the two constraints we have typically used, the training time increased by a factor of 5. This increase might be considered too large for our system to be effective. Fundamentally, there seem to be two possibilities of improving the training time. First, the implementation of the constraints could be improved. In our implementation, the largest part of the time is spent in calculating the metric value itself, with only minor overhead caused by other calculations. This means that speeding up the metric implementation, for example by using hardware acceleration, could lead to improved results. The other approach to improving the training time would be to choose metrics that can be calculated faster than the metrics that are currently being used. It seems unlikely that this is a realistic option since any metric should depend on all values of the network, to prevent an attacker from "hiding" the backdoor in the unused weights. When the metric depends on all values, it is likely to be slow to compute, since all values need to be used during computation.

Some of the overhead of our system is due to needing to train the models twice during each round of FL, once unconstrained to determine the metric values and once constrained

to adapt the metric values. This could be reduced by modifying the system. For the first round of FL, the process would remain unchanged, with unconstrained training followed by constrained training. In the next round of FL, the unconstrained training is skipped. As Figures 6.3 and 6.2 show, metric values behave very predictably during training. This indicates that it could be possible to either reuse the metric values from the previous round of FL, or use them to mathematically predict good target metric values for this round. This would reduce the computation and network overhead, since one less model needs to be trained and transferred over the network. Whether this concept still allows our system to be effective would have to be evaluated in future work.

8. Conclusion

Defense against poisoning attacks in FL remains an area of active research. In this thesis, we have proposed the approach of constraining benign training to improve the ability of defense mechanisms to detect adversarial models.

We have determined the best approach to constraining ML for our setup, comparing Bagdasaryan training with a new approach that optimizes for each objective individually, coming to the conclusion that the Bagdasaryan approach reaches a better adaption. We have also evaluated different methods of weighting the individual objectives in order to reach the best results, coming to the conclusion that, in this setup, weighting the losses once, at the beginning of training, to the smallest value, reaches the best results. Experiments on constraining more than two metrics suggest that, when constraining a larger number of metrics, with large differences in size, this approach does not lead to ideal results, instead suggesting that metrics should not be weighted at all in this case.

During our evaluation, we have seen that, in most cases, the adversary can constrain the adversarial models just as well as - or better than - the benign models. This indicates that it is not possible to detect adversarial models purely on the metric values when training benign and malicious models identically. We have shown, however, that there is a potential benefit if the adversary has to scale the model updates before aggregation since this limits the adaption capability of adversarial training, potentially allowing defense mechanisms to detect the adversarial models more easily when using our approach.

The main benefit of our approach has been shown to be when modifying the training setup further on the benign side. This combines some discoveries. First, *momentum* does not have a significant influence on benign training, while being critical for achieving a good backdoor accuracy. Second, the metric values reached by unconstrained training are typically larger for training with momentum than for training without momentum. Last, adaption to the constraints works best when constraining towards the average of the values the metrics would naturally take. It can also work well when constraining to larger values but performs worse for smaller values. This means that constrained benign training can be performed without momentum, forcing the adversary into a dilemma. Either, adversarial training is done with momentum, leading to a good backdoor accuracy, at the cost of having a bad adaption, which a clustering defense mechanism would easily find, or, training is done without momentum, reaching a good adaption that would pass the defense mechanism, but no strong backdoor is implanted. This means that the adversary has no way of embedding a backdoor in the aggregated model.

A strong point of our approach is that any improvement in adaption an attacker could reach would likely also be applicable to benign training, meaning that it is unlikely for the attacker to gain a permanent benefit in adaption quality. On the other hand, this also applies to benign training, where it is equally difficult to reach an improvement in adaption quality that would not also apply to the adversarial side.

Our system has many parameters specific to constrained training - weighting function and approach, choice of constrained metrics, alpha values, etc. - that can be modified. While our evaluation has found good values for the setups evaluated here, it is uncertain whether these choices are ideal for all situations. Further research should investigate whether the same rules apply (and whether the same results are reached) for other setups, for example in different domains, with different datasets, or under different distributions. For any real-world use of the system, it would have to be carefully adjusted for the concrete use case.

List of Figures

2.1 A possible defense setup using both DF and IR mechanisms. 8
2.2 A basic MTL setup. The first 3 layers of the NN are shared between the tasks, while each task also has an individual layer 10

4.1 An example of a two-dimensional constraint system. In the left figure, there are no constraints. In the right figure, the vectors are constrained to an angle between the blue lines. Green arrows correspond to benign model updates while the red arrow corresponds to adversarial model updates. The dotted arrows represent the aggregated model without (green) and with (red) the adversarial model update. One can see that in the left figure, the difference between the benign aggregated model and the adversarial aggregated model is larger than in the right figure. 18

5.1 Class Diagram for the core parts of the Machine Learning framework. . . . 23

6.1 Values of the cosine distance metric before (blue) and after (orange) scaling the local models. 30
6.2 The average value of the Cos metric after each round of training. The two uppermost lines show values for unconstrained models, where the values diverge, while the values for constrained models stay close to the targeted values after as little as one epoch of training. 32
6.3 The average value of the Euclidean metric after each round of training. At the top, the values for the unconstrained models can be seen diverging, while the values for the constrained models stay close to the targeted value after roughly 5 epochs of training. 32
6.4 The main-task accuracy after each round of training. Benign models always reach better accuracies than the respective malicious model, with the constrained models reaching better accuracies than the unconstrained models. Notably, for all models, the accuracy does not increase significantly after the first few rounds of training. 33
6.5 The absolute difference between measured and targeted cosine values for local models trained with different weighting functions. Lower values mean a better adaption to the constraint. 35
6.6 The absolute difference between targeted and measured Euclid values for local models trained with different weighting methods. Lower values correspond to a better adaption. 35
6.7 Backdoor accuracy of local models for different weighting methods. Higher values are better for the adversary. The fourth models have significantly worse backdoor accuracy due to their non-IID class being equal to the target class of the backdoor attack. 36

6.8 Main-task accuracy of local models trained with different weighting methods. Higher values are better. Max-weighting reaches significantly worse results than the other methods. 36
6.9 The absolute difference between targeted and measured Cosine values for local models trained with different weighting approaches. Lower values are better. Blue and green values are significantly smaller than red and orange ones. Note the logarithmic scale. 37
6.10 The absolute difference between targeted and measured Euclid values for local models trained with different weighting approaches. Lower values are better. Blue and green values are significantly smaller than red and orange ones. Note the logarithmic scale. 38
6.11 Main-task accuracy for local models trained with different weighting approaches. Higher values are better. Once-weighting (blue and green values) reaches a significantly higher accuracy than batch-weighting (orange and red values). 38
6.12 Backdoor accuracy for local models trained with different weighting approaches. Higher values are better for the adversary. 39
6.13 The absolute difference between targeted and measured Cos values for models trained with different α parameters. Lower values are better. 40
6.14 The absolute difference between targeted and measured Euclid values for local models trained with different α parameters. Lower values are better. . 40
6.15 Main-task accuracy of local models trained with different α parameters. Higher values are better. 41
6.16 Backdoor accuracy of local models trained with different α parameters. Higher values are better for the adversary. 42
6.17 The absolute difference between targeted and achieved Euclidean values for local models trained with different constraint learning rates. Lower values are better. 42
6.18 The absolute difference between targeted and achieved cos values for local models trained with different constraint learning rates. Lower values are better. 43
6.19 Main-task accuracy for local models trained with different constraint learning rates. Higher values are better. 43
6.20 Backdoor accuracy of local models trained with different constraint learning rates. Higher values are better for the adversary. 44
6.21 The absolute difference between targeted and measured cosine values for local models trained with both approaches. Bagdasaryan training constrains the metrics slightly better. Lower values are better. 45
6.22 The absolute difference between targeted and measured Euclidean values for local models trained with both approaches. Bagdasaryan training performs significantly better at constraining the values. Lower values are better. . . . 45
6.23 Main-task accuracy for local models trained with both approaches. No significant difference is visible between Bagdasaryan training and Individual Optimization. Higher values are better. 46
6.24 The backdoor accuracy of local models trained with both approaches. Individual Optimization reaches slightly better backdoor accuracy values. Higher values are better for the adversary. 46
6.25 Time required to train one epoch under both approaches. Lower values are better. Bagdasaryan Training trains roughly 0.5 seconds faster than Individual Optimization. There is no significant difference between benign and malicious training. 47

6.26 Cosine values of benign and malicious local models trained without constraints. 47
6.27 Euclid values of benign and malicious local models trained without constraints. While most adversarial values are similar to benign ones, the fourth value is significantly different. 48
6.28 Cosine values of unconstrained benign local models and constrained adversarial local models. The cosine values for the adversarial models are almost identical and similar to the benign ones. 48
6.29 Cosine values of benign and malicious local models trained with constraints. 49
6.30 Euclid values of benign and malicious local models trained with constraints. 49
6.31 Cosine values of benign and scaled malicious local models trained with constraints. The values for the malicious are consistently higher than those for the benign models. 50
6.32 Cosine values of benign and scaled malicious local models trained with constraints and adjusted target values. Even with correct constraint targets, scaling causes the metric values for malicious models to differ from those for benign models. 50
6.33 Training time required for one epoch of unconstrained and constrained training. Lower values are better. While unconstrained training takes roughly one second, constrained training requires 5 seconds. 51
6.34 Backdoor accuracy of aggregated models with backdoors trained with and without constraints during benign training. Higher values are better for the adversary. The backdoor accuracy falls to irrelevant levels immediately. . . 51
6.35 Backdoor accuracy of local models with backdoors trained with and without constraints. Higher values are better for the adversary. Constrained models reach moderately worse backdoor accuracies than unconstrained models. . . 52
6.36 The absolute difference between targeted and measured values for the Cosine metric of local models with different non-IID rates. Lower values are better. The data shows that higher non-IID rates do not lead to a better adaption, but do lower the range of values for each configuration. 53
6.37 The absolute difference between targeted and measured Euclid values of local models trained with different learning rates. Lower values are better. Benign models are marked `B`, while malicious models are marked `M`. Learning rate 0.001 leads to the best results. 54
6.38 The absolute difference between measured and targeted Cos values of local models trained with different learning rates. Benign models are marked `B`, while malicious models are marked `M`. The smallest learning rates lead to the best results (i.e., lowest values). 54
6.39 Main-task accuracy of local models trained with different learning rates. Higher values are better. Benign models are marked `B`, while malicious models are marked `M`. 55
6.40 Backdoor accuracy of local models trained with different learning rates. Higher values are better for the adversary. 55
6.41 The absolute difference between measured and targeted Euclid values of local models trained with different batch sizes. Lower values are better. The largest batch size (128) leads to the best results (i.e., the lowest values). 56
6.42 The absolute difference between measured and targeted Cos values of local models trained with different batch sizes. Lower values are better. The smallest batch size (32) leads to the best results (i.e., the lowest values). . . 56
6.43 Main-task accuracy of local models trained with different batch sizes. Higher values are better. 57

6.44 Backdoor accuracy of local models trained with different batch sizes. Higher values are better for the adversary. 57
6.45 The absolute difference between targeted and measured Cos metric values of local models trained with SGD and Adam optimizers. Lower values are better. The models trained by SGD reach significantly better values. 58
6.46 Main-task accuracy of local models trained with SGD and Adam optimizers. Higher values are better. The models trained using SGD reach better main-task accuracy values. 58
6.47 Backdoor accuracy of local models trained with SGD and Adam optimizers. Higher values are better for the adversary. The models trained using the Adam optimizer reach significantly higher backdoor accuracies. 59
6.48 Absolute difference of measured and targeted Euclid values of local models trained with different constraint target values. Lower values are better. . . 59
6.49 Absolute difference of measured and targeted Cos values of local models trained with different constraint target values. Lower values are better. . . 60
6.50 Main-task accuracy of local models trained with different constraint target values. Higher values are better. 60
6.51 Backdoor accuracy of local models trained with different constraint target values. Higher values are better for the adversary. 61
6.52 The absolute difference between targeted and measured Euclid values of local models trained with and without momentum. Lower values are better. 62
6.53 The absolute difference between targeted and measured Cos values of local models trained with and without momentum. Lower values are better. . . . 62
6.54 Main-task accuracy of local models trained with and without momentum. Higher values are better. 63
6.55 Backdoor accuracy of local models trained with and without momentum. Higher values are better for the adversary. 63
6.56 The absolute difference between targeted and measured cosine values of local models with and without momentum, for the target values of training without momentum. Lower values are better. 64
6.57 Backdoor accuracy of local models trained with and without momentum, for the target values of training without momentum. Higher values are better for the adversary. 64
6.58 Euclidean values for local models trained on the MNIST [28] dataset. Euclidean values can be constrained when training on the MNIST dataset, though they do not adapt as well as when using the Cifar10 [29] dataset. . . 65
6.59 Cosine values of local SqueezeNet models. 65
6.60 Main-task accuracy of local SqueezeNet models. Higher values are better. . 66
6.61 Backdoor accuracy of local SqueezeNet models. Higher values are better for the adversary. 66

List of Tables

3.1 The presented defense mechanisms and the metrics they use to detect backdoors. 13

6.1 Aggregated main-task and backdoor accuracies of FL setups with different momentum (M) and decay (D) values and six benign and five malicious models. 28
6.2 Average Main-Task and backdoor accuracies of the five malicious local models with different momentum (M) and decay (D) values. 28
6.3 Accuracies of per-layer constraint training with and without scaling. 29
6.4 Metric values of one local model before and after scaling. 30
6.5 Average of natural metric values for different configurations. 31
6.6 Range of natural metric values for different configurations. 31
6.7 Average and range of difference between targeted values and achieved values for local models. 41

Listings

5.1 General pattern for the implementation of loss functions. First, the absolute difference between the metric value and the targeted metric value is calculated. Subtracting half of the range and passing the value through a ReLU leads to values smaller than half of the range being set to 0. 24
5.2 Implementation of the loss function for the Cosine metric. At its core, the `cosine_distance_for_backpropagation` function iterates over the model's layers and sums up their cosine distances. 24
5.3 Implementation of the *Individual Optimization* approach. For each batch, the code iterates over all loss functions and performs the normal training steps for each. 25

Acronyms

ML Machine learning

FL Federated learning

NN Neural Network

MTL Multi-Task Learning

FedAVG Federated Averaging

DF Detection and Filtering

IR Influence Reduction

MOO Multi-Objective Optimization

IID identically and independently distributed

PDR poisoned data rate

SGD Stochastic Gradient Descent

Bibliography

[1] Q. Rao and J. Frtunikj, "Deep learning for self-driving cars: Chances and challenges," in *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*, SEFAIS '18, (New York, NY, USA), p. 35–38, Association for Computing Machinery, 2018.

[2] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, "Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1025–1032, 2017.

[3] S. Y. Alaba and J. E. Ball, "Deep learning-based image 3d object detection for autonomous driving: Review," *IEEE Sensors Journal*, pp. 1–1, 2023.

[4] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, "Medical image classification with convolutional neural network," in *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, pp. 844–848, 2014.

[5] D. Jha, M. A. Riegler, D. Johansen, P. Halvorsen, and H. D. Johansen, "Doubleunet: A deep convolutional neural network for medical image segmentation," in *2020 IEEE 33rd International Symposium on Computer-Based Medical Systems (CBMS)*, pp. 558–564, 2020.

[6] Q.-Q. Chen, Z.-H. Sun, C.-F. Wei, E. Q. Wu, and D. Ming, "Semi-supervised 3d medical image segmentation based on dual-task consistent joint learning and task-level regularization," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 1–1, 2022.

[7] S. K. Maher, S. G. Bhable, A. R. Lahase, and S. S. Nimbhore, "Ai and deep learning-driven chatbots: A comprehensive analysis and application trends," in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 994–998, 2022.

[8] R. S. Sai Dinesh, R. Surendran, D. Kathirvelan, and V. Logesh, "Artificial intelligence based vision and voice assistant," in *2022 International Conference on Electronics and Renewable Systems (ICEARS)*, pp. 1478–1483, 2022.

[9] J. Shang and J. Wu, "Voice liveness detection for voice assistants through ear canal pressure monitoring," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 3, pp. 1225–1234, 2022.

[10] E. García-Martín, C. F. Rodrigues, G. Riley, and H. Grahn, "Estimation of energy consumption in machine learning," *Journal of Parallel and Distributed Computing*, vol. 134, pp. 75–88, 2019.

[11] L. Wang, W. Wang, and B. Li, "Cmfl: Mitigating communication overhead for federated learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 954–964, 2019.

[12] 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, Sept. 2020.

[13] A. Goldsteen, G. Ezov, R. Shmelkin, M. Moffie, and A. Farkash, "Data minimization for GDPR compliance in machine learning models," *AI and Ethics*, vol. 2, pp. 477–491, Sept. 2021.

[14] A. Goldsteen, G. Ezov, R. Shmelkin, M. Moffie, and A. Farkash, "Anonymizing machine learning models," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology* (J. Garcia-Alfaro, J. L. Muñoz-Tapia, G. Navarro-Arribas, and M. Soriano, eds.), (Cham), pp. 121–136, Springer International Publishing, 2022.

[15] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," 2016.

[16] 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," 2022.

[17] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *Proceedings of 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.

[18] H. Zhu, J. Xu, S. Liu, and Y. Jin, "Federated learning on non-iid data: A survey," 2021.

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

[20] P. Rieger, T. D. Nguyen, M. Miettinen, and A.-R. Sadeghi, "DeepSight: Mitigating backdoor attacks in federated learning through deep model inspection," in *Proceedings 2022 Network and Distributed System Security Symposium*, Internet Society, 2022.

[21] X. Jiang, H. Hu, T. On, P. Lai, V. D. Mayyuri, A. Chen, D. M. Shila, A. Larmuseau, R. Jin, C. Borcea, and N. Phan, "Flsys: Toward an open ecosystem for federated learning mobile apps," *IEEE Transactions on Mobile Computing*, pp. 1–18, 2022.

[22] H. Zhang, J. Bosch, and H. H. Olsson, "End-to-end federated learning for autonomous driving vehicles," in *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2021.

[23] 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.

[24] K. Pillutla, S. M. Kakade, and Z. Harchaoui, "Robust aggregation for federated learning," *IEEE Transactions on Signal Processing*, vol. 70, pp. 1142–1154, 2022.

[25] D. Yin, Y. Chen, R. Kannan, and P. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *Proceedings of 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.

[26] 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," 2021.

[27] C. Huang, J. Huang, and X. Liu, "Cross-silo federated learning: Challenges and opportunities," 2022.

[28] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with mnist and emnist," *Applied Sciences*, vol. 9, no. 15, 2019.

[29] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[30] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing, and C. Igel, "Detection of traffic signs in real-world images: The german traffic sign detection benchmark," in *Proceedings of the International Joint Conference on Neural Networks*, 08 2013.

[31] 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.

[32] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *CoRR*, vol. abs/1708.06733, 2017.

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

[34] R. Caruana, "Multitask learning," *Machine Learning*, vol. 28, no. 1, pp. 41–75, 1997.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[36] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1139–1147, PMLR, 17–19 Jun 2013.

[37] A. Krogh and J. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems* (J. Moody, S. Hanson, and R. Lippmann, eds.), vol. 4, Morgan-Kaufmann, 1991.

[38] T. Krauß and A. Dmitrienko, "Avoid adversarial adaption in federated learning by multi-metric investigations," 2023.

[39] C. Fung, C. J. M. Yoon, and I. Beschastnikh, "The Limitations of Federated Learning in Sybil Settings," in *Symposium on Research in Attacks, Intrusion, and Defenses*, RAID, 2020.

[40] L. Muñoz-González, K. T. Co, and E. C. Lupu, "Byzantine-robust federated machine learning through adaptive model averaging," 2019.

[41] S. Andreina, G. A. Marson, H. Möllering, and G. Karame, "Baffle: Backdoor detection via feedback-based federated learning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 852–863, 2021.

[42] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 707–723, 2019.

[43] L. Zhao, S. Hu, Q. Wang, J. Jiang, C. Shen, and X. Luo, "Shielding collaborative learning: Mitigating poisoning attacks through client-side detection," *CoRR*, vol. abs/1910.13111, 2019.

[44] Z. Xiong, Z. Cai, D. Takabi, and W. Li, "Privacy threat and defense for federated learning with non-i.i.d. data in aiot," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 1310–1321, 2022.

[45] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private language models without losing accuracy," *CoRR*, vol. abs/1710.06963, 2017.

[46] T. Sun, D. Li, and B. Wang, "Decentralized federated averaging," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–12, 2022.

[47] S. Schäffler, R. Schultz, and K. Weinzierl, "Stochastic method for the solution of unconstrained vector optimization problems," *Journal of Optimization Theory and Applications*, vol. 114, pp. 209–222, July 2002.

[48] O. Sener and V. Koltun, "Multi-task learning as multi-objective optimization," in *Advances in Neural Information Processing Systems 31* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 525–536, Curran Associates, Inc., 2018.

[49] "Welcome to python.org." `https://www.python.org/`. Accessed: 2023-07-25.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[51] "Matplotlib - visualization with python." `https://matplotlib.org/`. Accessed: 2023-07-25.

[52] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[53] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[54] Z. Zhang, "Improved adam optimizer for deep neural networks," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pp. 1–2, 2018.

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, 4. September 2023

....

(Tobias Fella)