University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Black-box Robustness Analysis Module for an DFL Platform

Wenzhe Li
Zurich, Switzerland
Student ID: 21-739-156

Supervisor: Chao Feng, Dr. Alberto Huertas Celdran
Date of Submission: August 6, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Black-box adversarial attacks involve the generation of adversarial samples that can mislead a model by exploring and adjusting input data without requiring knowledge of the model's internal structure and parameters. Attackers typically observe the model's output to infer and optimise the input, identifying data that causes the model to misclassify or make incorrect predictions. This approach more closely reflects real-world scenarios, making it highly threatening. While researchers have made significant progress in applying black-box adversarial attack methods to machine learning, their performance in federated learning has not been thoroughly validated.

In this work, widely used black-box adversarial attack methods in machine learning were selected and thoroughly studied to gain a comprehensive understanding of their attack principles and implementation methods. These attack methods were then integrated into a federated learning platform, called Fedstellar in this work. By setting different federated learning parameters, the performance of the attack methods was evaluated in different environments and the robustness of the federated learning platform was assessed. The experimental results showed significant performance differences between the different attack methods. Moreover, the performance of these attack methods was highly related to the number of federated learning nodes, datasets and federation methods, while the topology had minimal impact on the attack performance. Based on the analysis of the experimental results, a new potential timing for black-box adversarial attacks is proposed, which could be further explored in future work to have a greater impact on federated learning.

Black-Box-Angriffe beinhalten die Generierung negativer Stichproben, die ein Modell in die Irre führen können, indem sie die Eingabedaten untersuchen und anpassen, ohne dass die interne Struktur und die Parameter des Modells bekannt sein müssen. Die Angreifer beobachten in der Regel die Ausgabe des Modells, um daraus Rückschlüsse auf die Eingabe zu ziehen und diese zu optimieren, indem sie Daten identifizieren, die das Modell zu Fehlklassifizierungen oder falschen Vorhersagen veranlassen. Dieser Ansatz entspricht eher realen Szenarien und ist daher äußerst bedrohlich. Während Forscher bei der Anwendung von Black-Box-Angriffsmethoden auf das maschinelle Lernen erhebliche Fortschritte gemacht haben, wurde ihre Leistung beim föderierten Lernen noch nicht gründlich validiert.

In dieser Arbeit wurden weit verbreitete Blackbox-Angriffsmethoden für das maschinelle Lernen ausgewählt und gründlich untersucht, um ein umfassendes Verständnis ihrer Angriffsprinzipien und Implementierungsmethoden zu gewinnen. Diese Angriffsmethoden wurden dann in eine föderierte Lernplattform integriert, die in dieser Arbeit Fedstellar genannt wird. Durch die Einstellung verschiedener föderierter Lernparameter wurde die Leistung der Angriffsmethoden in verschiedenen Umgebungen bewertet und die Robustheit der föderierten Lernplattform beurteilt. Die experimentellen Ergebnisse zeigten signifikante Leistungsunterschiede zwischen den verschiedenen Angriffsmethoden. Außerdem hing die Leistung dieser Angriffsmethoden stark von der Anzahl der föderierten Lernknoten, der Datensätze und der Föderationsmethoden ab, während die Topologie nur minimale Auswirkungen auf die Angriffsleistung hatte. Auf der Grundlage der Analyse der experimentellen Ergebnisse wird ein neues potenzielles Timing für Black-Box-Angriffe vorgeschlagen, das in zukünftigen Arbeiten weiter erforscht werden könnte, um einen größeren Einfluss auf föderiertes Lernen zu haben.

Acknowledgments

I would like to sincerely thank everyone who has supported and helped me throughout this process. First and foremost, I would like to thank Prof Dr Stiller, for his endless guidance and support throughout this research. I have benefited greatly from his expertise and patient guidance. I would like to thank all the members of the Communication Systems Group, especially my supervisors, Chao Feng and Dr Alberto Hueartas, for their invaluable advice and assistance during the experiments. Their cooperation and support have been crucial to my research. I would like to thank my family for their unconditional support and encouragement when I needed it most. Finally, I would like to thank everyone who has helped me, your support is deeply appreciated.

Contents

Chapter 1

Introduction

Federated Learning (FL) is an emerging Machine Learning (ML) approach that employs a distributed architecture. In this architecture, users' raw data is kept local and not transmitted over the network. Instead, users could collect and train their data locally, sharing the resulting local models across the network. This collaborative learning method ensures knowledge sharing while preserving user privacy. However, the decentralized nature of FL, especially in Decentralized FL (DFL), where nodes do not distinguish between trainers and aggregators, makes it vulnerable to malicious attacks [1]. Such attacks can compromise the robustness of the DFL system, leading to inaccurate results and diminished trustworthiness. Therefore, it is essential to develop and implement a scheme to analyze the robustness of the DFL system.

Previous research developed a module that uses a white-box approach to analyze the DFL platform's robustness [2]. This approach relies on the analyst having access to the DFL model's parameters or training data, which often is not the case in real-world scenarios. In contrast, black-box approach is preferred for its practical and realistic assessment. Model outputs are utilized in black-box analysis, avoiding assumptions about parameters or training data. This analysis often employs evasion attacks to create adversarial samples that mislead the model. As a result, the goal of this thesis is to design and implement a black-box robustness analysis module that specifically utilizes evasion attacks as a tool for a more practical evaluation of model robustness within the context of DFL.

1.1 Description of Work

The purpose of this work is to integrate existing black-box adversarial attack methods into a existing FL platform and to comprehensively evaluate the performance of different attack methods as well as the robustness of the FL platform by setting various parameters. The project is mainly divided into the following phases:

Literature Review. In this phase, the goal is to review and document the state-of-the-art in terms of the concepts of DFL, adversarial attack methods, technologies, and systems

relevant to the project. A thorough understanding of this prior knowledge will lay a solid foundation for the implementation of subsequent functionalities.

Design and Implementation of Different Attack Methods. During this phase, the first step is to present a proposal outlining the design and structure of the black-box robustness analysis module within the DFL framework. Following that, the selected attack methods will be integrated into the selected FL platform.

Evaluation and Conclusion. The primary task in this phase is to evaluate the attack methods that have been integrated into the FL platform. By setting different parameters within the platform, the performance of the attack methods will be measured comprehensively under various conditions.

1.2 Thesis Outline

The structure of this work is outlined as follows. First, Chapter 2 establishes the theoretical baseline and describes the fundamental concepts used in this work. Subsequently, currently existing ML platforms for adversarial attacks and several currently popular black-box adversarial attack methods are described in Chapter 3. Based on these findings, the specific implementation of the attack method on the FL platform and the methods required for the attack are thoroughly outlined in Chapter4. In the next Chapter 5, an extensive evaluation of the proposed attack methods with different parameters is given. Lastly, Chapter 7 summarizes the findings of this work and proposes future opportunities.

Chapter 2

Background

This chapter clarifies the theoretical foundations of FL and points out the advantages of FL as well as some of the security vulnerabilities that may currently exist. Based on these security vulnerabilities, it can greatly provide attackers with the opportunity to attack and negatively affect or even crash the platform. Therefore, this chapter will provide a general overview of FL as well as the black-box Evasion attack in the attack methodology.

2.1 Federated Learning

FL, a distributed ML approach where multiple clients collaborate to solve ML problems under the coordination of a central aggregator, has gained increasing attention and widespread application in recent years. In FL, a global model is initialized by a central server, which then sends the model parameters to all participating clients. Upon receiving these parameters, clients locally train the model using their own data and the global model parameters. After training, each client sends their updated local model parameters back to the central server. The central server then aggregates the local model parameters (e.g., via weighted averaging) to produce new global model parameters, which are redistributed to the clients. This process repeats until a predetermined number of training rounds is reached or the model converges, at which point training is terminated. The final global model is then ready for real-world application.

This method enhances user privacy by sharing only model parameters or gradient information, rather than raw data. Additionally, FL leverages decentralized local data to train models with improved performance. Its scalability and flexibility allow it to be extended to large-scale distributed systems, enabling numerous participants to collaboratively train models.

However, as a developing and emerging field, FL still faces several challenges, despite its growing impact in various areas. The first challenge is the communication cost [3]. In

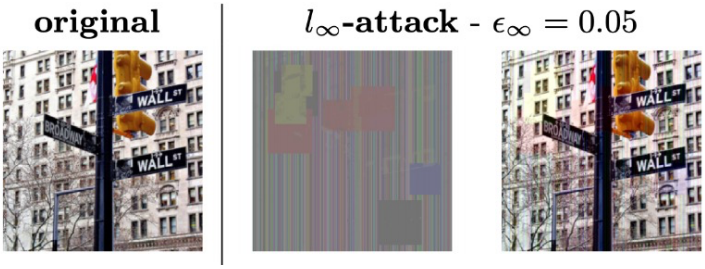


Figure 2.1: Visualization of the adversarial perturbations and examples of the Square Attack. [5]

fact, a federated network may consist of numerous devices, leading to significant communication overhead during the training process. Therefore, it is essential to develop efficient communication methods.

Secondly, there is the issue of model quality and robustness. Ensuring that the aggregated global model maintains good generalization ability and robustness across various data distributions and participant environments is crucial. Although FL enhances data privacy to some extent, there remains a risk of attack or theft during the transmission of model parameters or updates [4]. Malicious attackers can exploit vulnerabilities to manipulate the global model, posing threats to data integrity, algorithms, and the overall federation.

2.2 Evasion Attacks

In an adversarial attack, an attacker attempts to alter a data point x into an adversarial data point x' so that x' is misclassified by a high-confidence model, even though x' appears indistinguishable from the original data point x to a human observer. Figure 2.1 illustrates the visualization of an adversarial attack. The original sample is shown in the left image, where a square attack was applied. The right image depicts the adversarial sample obtained after the attack, while the middle image shows the hotspot map of the attack. It is evident that it is challenging for the naked eye to distinguish the difference between the sample before and after the attack. A evasion attack is a type of adversarial attack in which the attacker subtly modifies the input data to deceive the ML model, causing it to make incorrect predictions. This attack is particularly relevant to supervised learning models used for classification tasks.

2.2.1 Types of Evasion Attack

Evasion attacks can be categorized based on different classification methods. According to the level of knowledge possessed by the attacker, attacks can be divided into black-box and white-box attacks. In black-box attacks, the attacker has no knowledge of the model's internal structure, while in white-box attacks, the attacker has full access to the model. Based on different attack targets, it can be divided into targeted and untargeted attacks. Targeted attacks aim to misclassify inputs into specific categories, whereas non-targeted

attacks simply aim to cause any form of misclassification [6]. Additionally, different norms (l_0, l_2, and l_∞) are used in these attacks, each producing different effects. These norms represent various norms used to measure the perturbations applied to the original data points [7].

A *Black-box attack vs White-box attack*

- A black-box attack means that the attacker has no knowledge of the internal structure and parameters of the target model and can only generate adversarial samples by observing the relationship between inputs and outputs [8]. Due to the lack of information about the model's internals, black-box attacks require repeated trials and adjustments to generate effective adversarial samples. This attack method typically relies on evolutionary algorithms, optimization techniques, or generating adversarial samples by training models. Although the success rate and efficiency of black-box attacks may be lower than that of white-box attacks, they are more realistic in practical applications because attackers usually do not have access to detailed information about the target model.
- A white-box attack is one in which the attacker has complete knowledge of the internal structure and parameters of the target model. This attack leverages detailed information about the model architecture, weights, and training data, allowing for the precise computation of adversarial samples, thus making the attack more effective. White-box attacks typically employ gradient computation methods, such as the Fast Gradient Sign Method (FGSM) [9] and Projected Gradient Descent (PGD) [10], which generate adversarial samples by calculating the gradient of the input data with respect to the loss function [11]. Because the attacker has access to and can exploit all the information about the model, white-box attacks are generally more efficient and effective in generating adversarial samples.

B *Targeted attack vs Untargeted attack*

- The goal of a targeted attack is to cause the model to misclassify the input data into a specific category predetermined by the attacker. Such attacks require precise tuning of the adversarial samples so that the model outputs the wrong class specified by the attacker. As a result, targeted attacks are usually more complex and demand higher computational accuracy and resources [6].
- In contrast, untargeted attacks aim to induce misclassifications without focusing on any specific incorrect classification. The attacker only seeks to ensure that the model's output differs from the true category, and any misclassification is acceptable. Untargeted attacks are generally easier to implement and less computationally demanding, as they do not require precise control over the classification outcomes [12].

C *Attacks with Different Norms*

- The l_0 norm attack focuses on the number of modified features in the adversarial sample, aiming to minimize the number of changes. Attack methods using

the l_0 norm, such as JSMA (Jacobian-based Saliency Map Attack) [13], modify features by selecting those that have the greatest impact on the classification result. The advantage of this type of attack is that it can generate adversarial samples with minimal modifications, making the attack harder to detect. However, finding the optimal solution typically requires a high computational cost.

- The l_2 norm attack measures the Euclidean distance between the adversarial sample and the original sample, with the goal of minimizing this distance. Attack methods using l_2 norm, such as the Carlini & Wagner (C&W) attack [8], generate perturbations through an optimization algorithm to make the adversarial samples as close as possible to the original samples. The adversarial samples generated by this type of attack are usually more natural and less noticeable to the human eye, but the computational complexity is higher.
- The l_∞ norm attack focuses on countering the magnitude of the largest perturbation in the sample, with the goal of minimizing this maximum. This is achieved by using l_∞ norm attack methods, such as the Fast Gradient Sign Method (FGSM) [9] and Projected Gradient Descent (PGD) [10], which generate adversarial samples by adding small magnitude perturbations to each feature. These attacks are computationally efficient and suitable for large-scale applications, but the generated adversarial samples may sometimes be more easily detectable by the human eye [14].

Chapter 3

Related Work

This chapter introduces a series of existing platforms and frameworks for evaluating the robustness of ML, and provides new ideas for further integration of adversarial attack methods in FL platforms by comparing their strengths, weaknesses and unique features. It also provides a comprehensive overview of popular black-box adversarial attack methods and their implementation mechanisms.

3.1 Existing Adversarial Attack Platforms

As research on ML models progresses, attack and defence techniques have become an important research focus. To assess and improve the robustness of models, researchers have developed a series of tools and frameworks, including ART (Adversarial Robustness Toolbox) [15], Foolbox [16], Cleverhans [17], TextAttack [18] and so on. These tools not only provide researchers with convenient ways to implement attacks and defences, but also guarantee the repeatability and standardisation of experiments. Although there is no well-functional platform for evaluating the robustness of FL, based on the correlation between ML and FL, the study of the above platforms will also provide some inspiration for constructing a platform for FL. In the following, the features and applications of these tools will be described in detail.

3.1.1 ART (Adversarial Robustness Toolbox)

Adversarial Robustness Toolbox (ART) [15] is an open source toolbox developed by IBM for generating, detecting, and defending against adversarial attacks.ART has a wide range of application areas including image classification, speech recognition, text processing, and IoT device security. The toolkit supports several ML frameworks such as TensorFlow, Keras, PyTorch, and MXNet, making it easy to integrate with a wide variety of existing workflows.

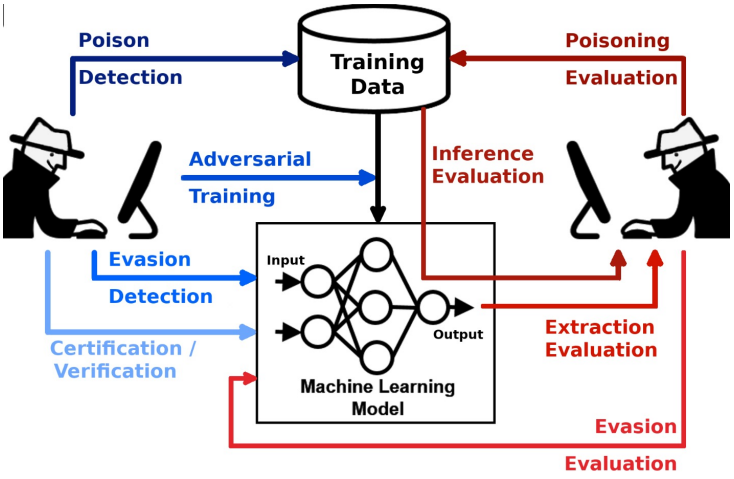


Figure 3.1: Illustration the Interaction Between Adversarial Attacks and Defenses in ART. [15]

ART's strength lies in its versatility, offering a rich set of attack and defense methods, including FGSM [9], PGD [10], and DeepFool [19], that address different types of adversarial threats, which means that we can test not only the effectiveness of the attack, but also the effectiveness of the platform's defences. The comparison between the two metrics helps us optimise and adjust our attack and defence methods. Figure 3.1 illustrates various of the attacks and defences available to ART. The ease use of ART is also a highlight, featuring a straightforward API that allows users to develop and test with ease. However, ART has its shortcomings, such as performance bottlenecks that can occur when dealing with large-scale datasets due to certain complex attacks and defense methods.

Additionally, ART's support for some frameworks may be less comprehensive than that of their native libraries. And in terms of user interface, ART is primarily used through a programming interface and does not have a standalone graphical user interface, requiring users to write Python code to call its functions. However, ART is well-documented, with detailed usage guidelines, API references, and various sample codes to help users get started quickly and fully utilize its features. ART's comprehensiveness and flexibility make it a unique and powerful platform for researchers and developers aiming to improve the robustness and security of ML models.

3.1.2 Cleverhans

Cleverhans [17] is an open source toolkit developed by the Google Brain team specifically for generating and defending against adversarial attacks. It is mainly used in the fields of image classification, speech recognition and text processing, and is designed to help researchers and developers better understand and respond to adversarial attacks.

The strength of Cleverhans lies in its flexibility and robust feature support. It supports a variety of adversarial attack methods, such as FGSM [9], BIM [20], and JSMA [14], allowing users to choose the appropriate attack method for their specific needs. Cleverhans also integrates very tightly with TensorFlow, making it suitable for projects using this

framework. However, a significant shortcoming of Cleverhans is its relatively weak support for other ML frameworks, which may be less convenient for users employing different frameworks. For novice users, some of Cleverhans' advanced features may require more in-depth background knowledge of adversarial attacks to fully understand and apply.

In terms of user interface, Cleverhans is primarily used through a programming interface and does not have a standalone GUI. Users need to write TensorFlow code to invoke its features. Cleverhans provides detailed documentation and sample code, including implementations of various attack methods and usage examples, to help users quickly understand and apply its features. Cleverhans is unique due to its tight integration with TensorFlow and its flexible and diverse adversarial attack methods, providing researchers and developers with a powerful tool to study and enhance the adversarial robustness of models.

3.1.3 TextAttack

TextAttack [18] is an adversarial attack and defence framework dedicated to Natural Language Processing (NLP) models. It is mainly used in NLP domains such as text categorisation, sentiment analysis, Q&A systems and machine translation etc. TextAttack was originally designed to provide a dedicated adversarial attack platform for NLP models, which fills the gap in this field. Its strength lies in its focus on textual data, supporting a wide range of adversarial attack methods for NLP tasks, such as TextFooler [21], DeepWordBug [22] and BAE [23]. TextAttack is commendably easy to use, offering straightforward command line tools and Python APIs for generating adversarial samples and conducting defense tests.

However, TextAttack has some shortcomings. Due to its focus on NLP tasks, support for other types of data is relatively limited. Additionally, when dealing with large text datasets, certain complex attack methods may lead to performance bottlenecks. TextAttack is primarily used through command-line tools and programming interfaces. TextAttack's detailed and easy-to-understand documentation includes usage guidelines, API references, and various sample codes to help users get started quickly and take full advantage of its functionality. TextAttack's uniqueness lies in its focus on NLP tasks and its diverse attack methodologies, providing researchers and developers with a powerful platform to study and improve the adversarial robustness of NLP models.

3.1.4 Foolbox

Foolbox [16] is an open-source toolkit developed by Bethge Lab that is designed for adversarial attacks and defences against ML models. It supports application domains such as image classification, object detection, and speech recognition.A significant advantage of Foolbox is its support for a variety of ML frameworks, including TensorFlow, PyTorch, and Keras, making it easy to integrate with a variety of existing workflows.Foolbox provides a

variety of attack methods such as the L-BFGS [7], DeepFool [19] and Carlini-Wagner attacks [8], allowing users to choose the right attack method for their specific needs.Foolbox's modular design also makes it easy to extend and customise attack methods.

Foolbox has its own drawbacks, certain complex functions that require time and background knowledge to understand and apply fully. Despite the detailed documentation, the use of certain advanced features may still need to be explored further. Foolbox primarily utilizes a programming interface and lacks a separate GUI. In order to take full advantage of its various functionalities, users are required to write Python code that allows them to access and utilize these features effectively. Foolbox's documentation offers comprehensive usage guidelines, API references, and sample code to assist users in understanding and utilizing its features. Foolbox stands out in the field of ML and adversarial robustness due to its impressive range of supported frameworks. This unique combination provides researchers and developers with a highly effective and powerful tool that is invaluable for studying adversarial attacks. Additionally, it facilitates efforts to enhance the robustness of models against such attacks, making it an essential resource for both theoretical exploration and practical application in the realm of adversarial ML.

3.2 Black-box Evasion Attack Methods

This section discusses the current state of research and contributions from other authors. Since black-box attacks do not require access to too much information about the model's internals and present a powerful capability in different attack scenarios, researchers have developed a variety of efficient black-box attack algorithms, such as Simple Black-box Attack(SimBA) [24], Square Attack [5], Boundary Attack [25], HopSkipJumpAttack [26] and etc. In this work, the above attacks are integrated in the Fedstellar [27] to measure its robustness. In the following, the design theory of these key algorithms and their implementations are described in detail.

3.2.1 Simple Black-box Attack(SimBA)

SimBA (Simple Black-box Attack) [24] is a black-box adversarial attack method for image classifiers that aims to find adversarial samples that can spoof the model relies on queries to the model output. SimBA's core idea is to use random sampling to modify the pixel values in the pixel space of an image step-by-step to minimise or maximise the output probability of the target classifier, so as to cause the classifier to misjudge.

SimBA firstly randomly selects a pixel or direction from the input image and tests perturbations in both positive and negative directions to find which one most reduces the classifier's predicted probability. Then, the direction of the perturbation with more significant effect is selected and this perturbation is accumulated to gradually construct the adversarial samples. This process is repeated until a predetermined attack effect or an upper limit on the number of queries is reached.

SimBA has the advantage of simplicity and ease of implementation, and can be applied on a wide range of complex models and tasks since it only requires access to the output of the model [28]. In addition, SimBA employs a random sampling strategy that avoids relying on gradient information, allowing it to exhibit high efficiency when dealing with high-dimensional inputs. SimBA achieves a better balance between query count and attack effectiveness, but its random nature can lead to significant performance variability across different images and models [29]. SimBA offers an innovative and efficient approach to black-box attacks, with a straightforward design and minimal computational overhead, making it highly practical for real-world applications.

Algorithm 1 Pseudo-code about SimBA

1: **Input:** x, y, Q, ϵ
2: $\delta \leftarrow 0$
3: $p \leftarrow p_h(y \mid x)$
4: **while** $p_y = \max_{y'} p_{y'}$ **do**
5: Pick randomly without replacement: $q \in Q$
6: **for** $\alpha \in \{\epsilon, -\epsilon\}$ **do**
7: $p' \leftarrow p_h(y \mid x + \delta + \alpha q)$
8: **if** $p'_y < p_y$ **then**
9: $\delta \leftarrow \delta + \alpha q$
10: $p \leftarrow p'$
11: **break**
12: **end if**
13: **end for**
14: **end while**
15: **return** δ

Algorithm 1 shows the pseudo-code about the SimBA pixel attack, which illustrates the SimBA pixel attack process. In addition to the attacked image x and the target classification label y (the desired misclassification label for a targeted attack and the correct label for a untargeted attack), it is also necessary to provide the set of perturbation directions Q and the perturbation step size ϵ. Before starting the attack (the second and third line of the code) there is no perturbation in the initial state, i.e., the initialisation against perturbation δ is a zero vector, and p is the probability distribution of image x under current model. The loop is entered when the probability p_y of the current label y is the highest among all categories. Firstly, a perturbation direction q is randomly selected from the set Q, to avoid repetition the same direction is selected and not put back. Then, try the two values of the perturbation step size, ϵ and $-\epsilon$, in succession. For the selected perturbation step size, calculate the predicted probability distribution of the new image under the model after adding the perturbation αq and assign it to p' (seventh line of code). If the new predicted probability p'_y is less than the probability p_y of the current label y, the direction of the perturbation is considered valid, and the valid perturbation αq will be accumulated to the current perturbation δ and the new predicted probability distribution will be assigned to p (the eighth to the tenth lines of code). Finally, when the probability of the current label y is no longer the maximum probability, the final adversarial perturbation δ is returned.

3.2.2 Square Attack

Square Attack [5] is an efficient black-box adversarial attack method that generates adversarial samples that can mislead DL models through random search. The method derives its name from its use of square-shaped perturbations to attack image classifiers.The core idea of Square Attack is to iteratively apply small square-shaped perturbations to an image, and gradually optimize these perturbations to maximize the target model's classification error probability. In particular, Square Attack first initialises an adversarial perturbation, then randomly selects a part of the image in each iteration, applies a square perturbation of a certain magnitude, calculates the model's prediction, and decides whether to accept the perturbation based on the result. If the new perturbation makes the model's classification probability significantly lower, the perturbation is retained; otherwise, the perturbation is discarded and then search for a new perturbation direction continues randomly. In this way, Square Attack is able to find effective adversarial samples with fewer queries.

Square Attack provides a wide array of advantages. Firstly, just like other black-box attack methods, it does not rely on the gradient information of the target model, and thus is suitable for any type of black-box model, including those without access to the internal structure and parameters [28]. Second, due to the strategy of random search and square perturbation, Square Attack exhibits high attack efficiency in high-dimensional input spaces. In addition, the use of square perturbation is able to generate visually imperceptible adversarial samples while keeping the number of queries low.Square Attack's approach is straightforward, yet it is highly effective, demonstrating superior attack performance on several standard datasets and models [30]. Although Square Attack has a certain degree of randomness in the random search process, stable and effective attack results can usually be obtained through reasonable hyper-parameter settings and multiple experiments. Overall, Square Attack provides a novel and effective idea for black-box counterattacks, and its high efficiency and versatility make it have a wide range of potential and value in practical applications.

Algorithm 2 Pseudo-code about Square Attack

Require: classifier f, point $x \in \mathbb{R}^d$, image size w, number of color channels c, l_p-radius ϵ, label $y \in \{1, \ldots, K\}$, number of iterations N
Ensure: approximate minimizer $\hat{x} \in \mathbb{R}^d$ of the problem stated in Eq. (1)
$\hat{x} \leftarrow \text{init}(x)$, $l^* \leftarrow L(f(x), y)$, $i \leftarrow 1$
while $i < N$ **and** $\hat{x}$ is not adversarial **do**
  $h^{(i)} \leftarrow$ side length of the square to modify (according to some schedule)
  $\delta \sim P(\epsilon, h^{(i)}, w, c, \hat{x}, x)$
  $\hat{x}_{\text{new}} \leftarrow \text{Project}(\hat{x} + \delta \text{ onto } \{z \in \mathbb{R}^d : \|z - x\|_p \leq \epsilon\} \cap [0, 1]^d)$
  $l_{\text{new}} \leftarrow L(f(\hat{x}_{\text{new}}), y)$
  if $l_{\text{new}} < l^*$ **then**
    $\hat{x} \leftarrow \hat{x}_{\text{new}}$, $l^* \leftarrow l_{\text{new}}$
  end if
  $i \leftarrow i + 1$
end while

Algorithm 2 is a concise description of the Square Attack flow using pseudo-code. Before the attack begins, initialize the adversarial sample as the input image $\hat{x}$, compute its loss value under the current model and assign it to l^*, then set the iteration count i to 1. The loop continues while the number of iterations i is less than the maximum N or the current sample $\hat{x}$ is not adversarial. Determine the side length $h^{(i)}$ of the square to be modified according to some strategy (e.g. stepwise reduction) (third line of code). Sample a perturbation δ from a perturbation distribution P that depends on the perturbation step size ϵ, the side length of the square $h^{(i)}$, the width of the image w, the number of colour channels c, the current adversarial sample $\hat{x}$ and the original input image x. Then, project the image $\hat{x} + \delta$ after adding the perturbation into the image space that satisfies the l_p paradigm constraint ϵ and has pixel values in the range [0, 1] to ensure that the perturbation does not exceed the given constraint. Compute the loss value of the new adversarial sample under the model and assign it to l_{new}. If the new loss value is less than the current minimum loss value, the adversarial sample is updated to the new sample and the minimum loss value is updated. The number of iterations is increased until either i reaches the maximum N or an adversarial sample is generated.

3.2.3 Boundary Attack

Boundary Attack [25] is an efficient and reliable black-box adversarial attack method that focuses on image classification models. The core idea of the method is to find the smallest perturbation that can deceive the model by progressively searching for adversarial samples near the decision boundary of the target model. Boundary Attack starts with an adversarial sample that has been classified incorrectly. This initial step is crucial, as it sets the foundation for the subsequent processes within the attack methodology. It then progressively reduces the adversarial perturbation through a series of iterations, while ensuring that the perturbed sample remains misclassified. Each iteration is divided into two steps: first, large jumps are made in the direction normal to the decision boundary to get closer to the decision boundary as quickly as possible; then, small fine-tuning steps are taken along the direction parallel to the decision boundary to reduce the magnitude of the perturbation. In this way, by alternating large and small steps, Boundary Attack can efficiently search for adversarial samples with the smallest perturbation near the decision boundary [31].

Boundary Attack has several significant advantages. The method is versatile, applicable to various models and tasks, including image classification and speech recognition. In addition, Boundary Attack is highly efficient in generating adversarial samples. Its iterative process can quickly converge to the optimal solution, reducing the number of queries and computational overhead.

In practical implementation, Boundary Attack employs a random search strategy, exploring the decision boundary through random perturbations in high-dimensional space. This strategy avoids reliance on gradient information, making it robust and stable when handling high-dimensional inputs. Additionally, the method incorporates optimization techniques such as adaptive step size adjustment and sample pruning to enhance search efficiency and the quality of adversarial samples. Algorithm 3 shows the pseudo-code for

Algorithm 3 Boundary Attack Pseudo-code

Require: original image o, adversarial criterion $c(.)$, decision of model $d(.)$
Ensure: adversarial example $\tilde{o}$ such that the distance $d(o, \tilde{o}) = \|o - \tilde{o}\|_2^2$ is minimized
1: initialization: $k = 0, \tilde{o}^0 \sim U(0, 1)$ such that $\tilde{o}^0$ is adversarial
2: **while** $k <$ maximum number of steps **do**
3: draw random perturbation from proposal distribution $\eta_k \sim \mathcal{P}(\tilde{o}^{k-1})$
4: **if** $\tilde{o}^{k-1} + \eta_k$ is adversarial **then**
5: set $\tilde{o}^k = \tilde{o}^{k-1} + \eta_k$
6: **else**
7: set $\tilde{o}^k = \tilde{o}^{k-1}$
8: **end if**
9: $k = k + 1$
10: **end while**

Boundary Attack. The inputs needed for Boundary Attack are original image $\tilde{o}$, adversarial criterion $\tilde{c}(.)$ and model $\tilde{d}(.)$. Firstly, the number of iterations k is set to 0 and an initial adversarial sample $\tilde{o}^0$ is randomly generated from the uniform distribution U(0, 1) so that it is an adversarial sample. When k is less than the predetermined maximum number of iterations, a random perturbation η_k is drawn from the proposed distribution $P(\tilde{o}^{k-1})$, which is based on the current adversarial sample $\tilde{o}^{k-1}$. If the new perturbation sample $\tilde{o}^{k-1} + \eta_k$ is an adversarial sample, set the new adversarial sample $\tilde{o}^k$ to $\tilde{o}^{k-1} + \eta_k$. Conversely, if the new perturbation sample is not an adversarial sample, leave the current adversarial sample unchanged.

3.2.4 HopSkipJumpAttack (HSJA)

HopSkipJumpAttack (HSJA) [26] is an efficient decision-based attack method designed to generate adversarial samples that deceive an image classification model while minimizing the number of queries. HSJA achieves this by accessing the decision results of the model instead of obtaining information about its internal structure or parameters. The core idea of the method is to generate adversarial samples by using a boundary gradient estimation algorithm and a binary search strategy to perform effective perturbations near the classification boundary in order to find the minimum amount of perturbations.

Initially, HSJA generates an adversarial sample on the decision boundary of the target model. To be specific, the attacker starts with a correctly categorised sample and finds the initial adversarial sample on the boundary by gradually adding perturbations such that the sample is just misclassified. This process can be achieved by a simple geometric method, i.e., searching along the connecting line between the input sample and the target category sample.

After obtaining the initial adversarial samples, HSJA uses a boundary gradient estimation method to optimize the perturbations. Boundary gradient estimation exploits the geometric properties near the decision boundary by estimating the normal vector of the decision boundary, which is completed by applying small perturbations to the input samples and observing the changes in the model's classification results. This normal vector

indicates how the minimum perturbation on the decision boundary can be made so that samples are misclassified across the boundary.

Subsequently, HSJA further optimises the adversarial samples using a binary search strategy. Concretely, the attacker performs a binary search between the input sample and the initial adversarial sample, gradually narrowing the search interval to find the adversarial sample with the smallest amount of perturbation. The binary search decides the next search direction by taking the midpoint between the input sample and the adversarial sample and checking result of this midpoint. This method not only improves the search efficiency [32], but also ensures that the adversarial samples found have the minimum amount of perturbation.

The basic intuition behind the HopSkipJumpAttack algorithm is depicted in Algorithm 4. The algorithm is initialised from initialising the variable θ, and is guaranteed to initialise $\tilde{x}0$ so that it is on the decision boundary of the target class x^*, and also need to calculate the ℓ_p distance between $\tilde{x}_0$ and the target class x^* , denoted as d_0. When the number of iterations is below the maximum limit T, a boundary search is first performed to find a new adversarial sample x_t on the boundary using a binary search, a method that will be mentioned next. The following step is gradient-direction estimation, calculate the batch size for the current iteration B_t by calculating $B_0\sqrt{t}$, and compute the direction vector v_t. Next for the step size search, where ξ_t (Step size) is initialised as the ℓ_p distance between the current adversarial sample x_t and the target class sample x^* divided by $\sqrt{t}$. As the number of iterations increases, the step size is gradually reduced to 1/2 of the original. The while loop ends when the model no longer classifies it as the target class x^*. Then, calculating the ℓ_p distance between the current $\tilde{x}$ and the target class x^*, denoted as d_t. Upon exiting the for loop, output the final adversarial sample x_t using binary search again.

The binary search in Algorithm 5 used in HopSkipJumpAttack is a method to finally obtain a sample that satisfies the constraints by gradually narrowing the upper and lower bounds and adjusting the position of the upper and lower bounds according to the classification results. Firstly, the upper and lower bounds are initialised as α_u and α_l respectively, and when the difference between the upper and lower bounds is greater than the threshold θ, iteration is performed. In the loop, the midpoint α_m of α_l and α_u is first calculated, and if $\phi(\Pi_{x,\alpha_m}(x'))$ is equal to 1 (i.e., the sample at the midpoint is classified as 1), the upper bound α_u is set to α_m. Otherwise, set the lower bound α_l to α_m. Finally output samples that close to the decision boundary x''.

3.3 Research Motivation

Current research reveals a significant knowledge gap in DFL regarding defenses against black-box attacks. Currently, platforms like Adversarial Robustness Toolbox (ART) and Foolbox provide comprehensive frameworks for assessing the adversarial robustness of ML models. However, these platforms are mainly oriented towards traditional ML models and lack support for FL environments, especially those containing DFL models, which are part of the FedStellar. DFL models operate under a fully decentralised architecture with no

Algorithm 4 Pseudo-code about HopSkipJumpAttack

Require: Classifier C, a sample x, constraint ℓ_p, initial batch size B_0, iterations T
Ensure: Perturbed image x_t

Set θ
Initialize at $\tilde{x}_0$ with $\phi_x^*(\tilde{x}_0) = 1$
Compute $d_0 = \|\tilde{x}_0 - x^*\|_p$
for t in $1, 2, \ldots, T-1$ **do**
  (Boundary search)
  $x_t = \text{BIN-SEARCH}(\tilde{x}_{t-1}, x, \theta, \phi_x^*, p)$
  (Gradient-direction estimation)
  Sample $B_t = B_0\sqrt{t}$ unit vectors $u_1, \ldots, u_{B_t}$
  Set δ_t
  Compute $v_t(x_t, \delta_t)$
  (Step size search)
  Initialize step size $\xi_t = \|x_t - x^*\|_p/\sqrt{t}$
  while $\phi_x^*(x_t + \xi_t v_t) = 0$ **do**
    $\xi_t \leftarrow \xi_t/2$
  end while
  Set $\tilde{x}_t = x_t + \xi_t v_t$
  Compute $d_t = \|\tilde{x}_t - x^*\|_p$
end for
Output $x_t = \text{BIN-SEARCH}(\tilde{x}_{t-1}, x, \theta, \phi_x^*, p)$

Algorithm 5 Pseudo-code about HopSkipJumpAttack's Binary Search Method

Require: Samples x', x, with a binary function ϕ, such that $\phi(x') = 1$, $\phi(x) = 0$, threshold θ, constraint ℓ_p
Ensure: A sample x'' near the boundary

Set $\alpha_l = 0$ and $\alpha_u = 1$
while $|\alpha_l - \alpha_u| > \theta$ **do**
  Set $\alpha_m \leftarrow \frac{\alpha_l + \alpha_u}{2}$
  if $\phi(\Pi_{x,\alpha_m}(x')) = 1$ **then**
    Set $\alpha_u \leftarrow \alpha_m$
  else
    Set $\alpha_l \leftarrow \alpha_m$
  end if
end while
Output $x'' = \Pi_{x,\alpha_u}(x')$

Table 3.1: Classification of evasion attacks.

Category	**Type**	**Method**	**Efficiency**
SimBA	Black-box	Random order	
		Diagonal order	
		SimBA-Pixel	Low
		SimBA-DCT	
		Targeted/Untargeted	
Pixel Attack	Black-box	Random Pixel Attack	
		One Pixel Attack	Low
		Targeted/Untargeted	
Square Attack	Black-box	l_2 norm	
		l_∞ norm	Moderate
		Targeted/Untargeted	
Threshold Attack	Black-box	l_0 norm	
		l_∞ norm	Moderate
		Targeted/Untargeted	
Boundary Attack	Black-box	Targeted/Untargeted	High
HSJA	Black-box	l_0 norm	
		l_∞ norm	High
		Targeted/Untargeted	
PGD	White-box	Iterative attack	High
FGSM	White-box	Single-step attack	High

central server to coordinate model updates. This decentralised nature requires robustness assessment tools to be able to handle different topologies and communication patterns in order to efficiently assess the defensive capabilities of the model.

Another issue is the lack of user-friendly interfaces for existing tools. These platforms usually require users to have a high level of technical background to use them effectively, which limits their application to a wider user community. Moreover, the straightforward user interface enables researchers to easily perform adversarial tests and analyses.

A summary of previous work with a focus on their advantages and things need to be improved is available in Table 3.1, which highlights the need for innovative research. Another Table summarises the current common adversarial attack methods and the sub-categories included in each attack, making it more intuitive to select the attack methods to be integrated on Fedstellar by comparison.

As a result, design and implementation of a black-box robustness analysis module for DFL models is of great significance. This module can not only make up for the shortcomings of existing platforms in DFL, but also improve the usability of the tool through a user-friendly interface, and promote the further development of adversarial robustness research.

Table 3.2: Classification of popular adversarial machine learning platforms nowadays.

Platform	**Application Area**	**ML/FL**	**UI**	**Documentation**	**Pros**	**Cons**
Foolbox	Adversarial machine learning	ML	CLI	Comprehensive	Comprehensive attack methods and framework	Not include the latest attack techniques
Cleverhans	TensorFlow-based model robustness testing	ML	CLI	Good	Seamless integration with TensorFlow models	Primarily focused on TensorFlow
ART	Comprehensive research in adversarial attacks and defences(Healthcare, finance)	ML	CLI	Extensive	Includes a wide range of attack methods, defences, and robustness evaluation metrics	May take a long time to handle large-scale datasets and models
TextAttack	NLP research	ML	CLI	Good	Supports various attack methods tailored for NLP	Limited to NLP applications

Chapter 4

Design and Implementation

With well-defined attack methods and identified research gaps. Firstly, this chapter will present the attack methods used in this research, the selection criteria for the parameters required during the attack and the metrics used to evaluate the performance of the attack. Secondly, the next sections will detail the implementation steps, experimental setups in Fedstellar for each of the attack methods.

4.1 Attack Specification & Evaluation Metrics

In this context of work, four types of black-box attacks are selected to evaluate the effectiveness of current methods and the model's capacity to resist attacks, which are Simple Black-box Attack (SimBA), Square Attack, Boundary Attack, and HopSkipJumpAttack(HSJA). The reasons for choosing these attacks are that they cover almost all types of black-box attacks types, such as decision-based attack and score-based attack, use different norms as well as different attack order for the samples. Moreover, these attacks are designed without assuming that the attacker understands the deployed aggregation algorithm. In addition, the following attack parameters are introduced:

- The **step size(ϵ)** sets the maximum perturbation applied to the original input, which means that the difference between the generated adversarial samples and the original samples will not exceed the range of epsilon. Smaller (ϵ) values can make the adversarial samples more stealthy, while larger (ϵ) values may generate more easily detectable adversarial samples.

- **Maximum number of iterations(max_iter)** Although the probability of a successful attack is extremely high, a suitable *max_iter* will be set in order to avoid some extreme cases that will lead to failure of the attack and a dead loop. Furthermore, choosing an appropriate *max_iter* can effectively reduce the time of the attack, which is very helpful to improve the efficiency of the attack.

	Parameters			Metrics
SimBA	$\epsilon = 0.4$	$max\_iter = 1000$		Accyracy
Square Attack	$\epsilon = 0.3$	$max\_iter = 500$	$p_{init} = 0.3$	Precision
Boundary Attack	$\epsilon = 0.01$	$max\_iter = 500$	$\delta = 0.01$	Recall
HSJA	$norm = l_2$	$max\_iter = 50$	δ and θ depend on l_2 distance	F1 Score
	$norm = l_\infty$		δ and θ depend on l_∞ distance	

Table 4.1: Configuration overview for each of the selected attacks. The calculations of HSJA δ using l_2 norm are given in Equations 4.1, and the calculations of HSJA δ using the l_∞ norm are given in Equations 4.2.

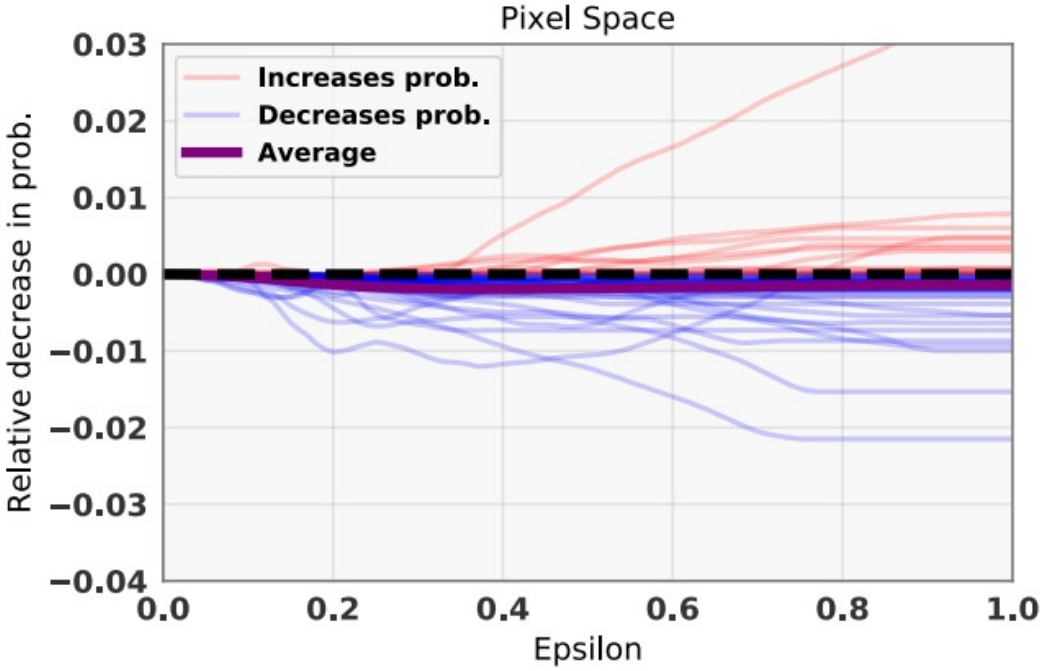


Figure 4.1: Relative Probability with Different ϵ. [24]

- The **size of the initial perturbation** is an initial probability parameter that controls the size and scope of the initial perturbation. Higher values of the initial probability parameter allow the algorithm to explore more in different regions of the image, thus increasing the chances of finding vulnerabilities. With the progress of the attack, the value of the initial probability parameter usually decreases to allow for finer tuning of the perturbation, thus increasing the accuracy and effectiveness of the attack.

- **Sign of the end of the attack** are similar to *max_iter* in that both of them aim to prematurely terminate attacks. It is typically used to indicate a change in the model's probability of misclassification or incorrect prediction.

Table 4.1 summarises all the attacks and the parameters required for their attack. This course of section explains the details of the parameter value selection and the metrics used to evaluate the effectiveness of the attack.

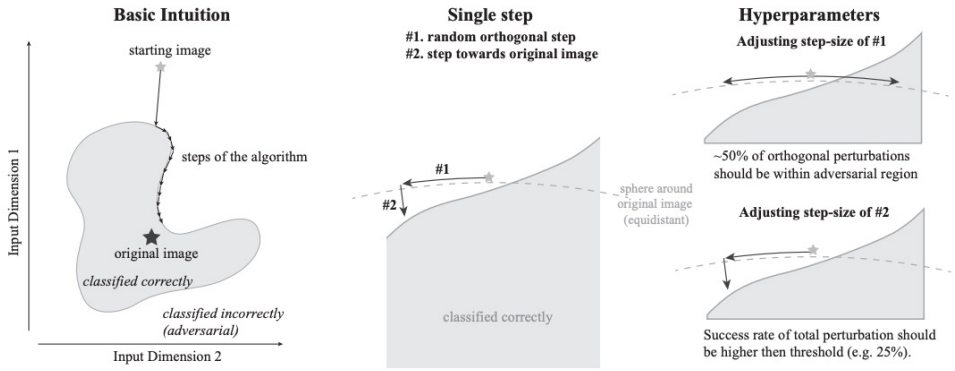


Figure 4.2: Demo of the Boundary Attack Process. [25]

SimBA. In this attack, the value of ϵ is chosen to be 0.3. As can be seen in Figure 4.1, the probability of both increasing and decreasing predicted class probability increases as ϵ increases, and the average change (purple line) exhibits a tendency to first decrease and then increase, with a peak at about the time when ϵ is 0.3. Most of the probability change curves remain within a relatively small range when ϵ is near 0.3, which neither induces a large perturbation nor makes the effect of the perturbation too small. When applying small ϵ (e.g., below 0.1), the perturbation effect has a limited impact on the probability, while applying larger ϵ (e.g., above 0.6), the perturbation may significantly alter the probability, potentially resulting in unstable model performance or failure of the attack. Experiments show that when the number of queries is large enough (>7500), the probability of success of the SimBA attack tends to be close to 1. Due to differences in the dataset and the significant effect of increasing max_iter on attack time, we select a maximum of 1000 iterations to attain about 90% success rate (showed in [24]) and the reduction of attack time.

Square Attack. In this attack, the value of ϵ is sample dependent and needs to be taken into account whether the pixels are in $[0, 1]$ or in $[0, 255]$ for a particular dataset dataset and model. For example, for the standard ImageNet models, the correct l_2 ϵ to specify is 1275 since after division by 255 it will become 5.0. In this experiment, all pixels are in $[0, 1]$. Therefore, the final value of ϵ selected is 0.3. Experiments show that $\epsilon = 0.3$ can produce effective adversarial samples on multiple datasets and models with good generality. Based on experimental verification and theoretical analysis, the optimal choice for p_{init} is 0.3, as it significantly enhances the probalility of generating effective perturbations in the initial stage, facilitating the exploration of the input space and identification of the optimal attack path.

Boundary Attack. Two relevant parameters in Boundary Attack: the length of the total perturbation δ and the step size towards the original input are dynamically varied. In other words, these two parameters are dynamically adjusted according to the local geometry of the boundary. As it is shown in Figure 4.2, Boundary Attack begins with rejection sampling along the boundary separating the adversarial and non-adversarial images (Left). At each step, generate a new random direction by (#1) sampling from an iid Gaussian and projecting it onto a sphere, and (#2) making a slight adjustment toward the target image (Center). Typically, the closer we get to the original image, the flatter the decision

boundary becomes, and the smaller the step size has to be to continue making progress. The attack is considered converged when the step size approaches zero. The *max_iter* depends on the angle of the decision boundary in the local neighbourhood . If the success rate is too small we decrease it, if it is too large we increase it. Comprehensive experiments have shown that *max_iter* of 500 is able to guarantee a high success rate of the attack while at the same time taking into account the efficiency.

HSJA. In HopSkipJumpAttack, Monte Carlo estimation is used to approximate the gradient direction. Specifically, θ is a small positive parameter that controls the size of the perturbation at the sampled points on the unit sphere. This perturbation parameter needs to be chosen in such a way that the estimated gradient direction is unbiased, i.e., the estimation is unbiased as θ tends to zero. δ is used in the boundary search step to ensure that the adversarial samples are located near the decision boundary. The size of δ affects the proximity to the boundary, with smaller values of δ bringing the adversarial sample closer to the decision boundary, but also potentially increasing the number of queries. For different norms, δ is calculated differently, when using the l_2 norm:

$$\delta = \sqrt{\prod_{i=1}^{n} \text{shape}_i} \times \theta \times \|\text{original\_sample} - \text{current\_sample}\|_2 \tag{4.1}$$

when using l_∞ norm:

$$\delta = \left(\prod_{i=1}^{n} \text{shape}_i\right) \times \theta \times \|\text{original\_sample} - \text{current\_sample}\|_\infty \tag{4.2}$$

Metrics. Measuring the performance of an attack method using Accuracy (Equation 4.3), Precision (Equation 4.4), Recall (Equation 4.5), and F1 Score (Equation 4.6) has several significant advantages. These metrics provide a comprehensive view of the model's performance on positive and negative samples; Accuracy measures the model's overall predictive correctness, but may not be sensitive enough in class-imbalanced datasets, Precision and Recall measure the model's performance in predicting positive cases, providing a more fine-grained analysis, which is especially important when the data is class-imbalanced, and F1 Score, which is the reconciled average of Precision and Recall, combines the strengths of the two metrics for scenarios where they need to be balanced. The above four metrics are calculated as follows.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.3}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.4}$$

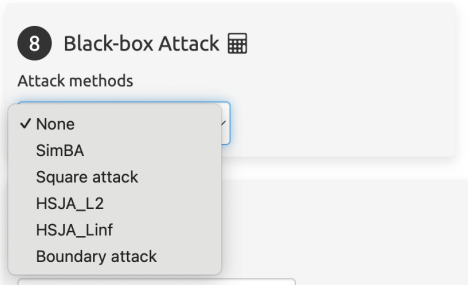
$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.5}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.6}$$

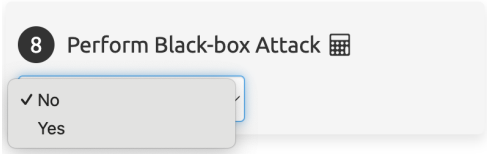
Meanwhile, the metrics used to measure the performance of DFL in Fedstellar are also Accuracy, Precision, Recall and F1 Score, which intuitively reflect the performance of the attack method by comparing the difference between the metrics before and after the attack. After the attack is complete, a significant decrease in model Accuracy indicates an effective attack and a decline in Precision suggests an increase in false positives, while a reduction in Recall implies that the attack successfully misclassifies many positive examples as negative. If the F1 Score also decreases, it reflects a more detrimental overall effect of the attack.

4.2 Adaption to Fedstellar

For the Fedstellar with black-box evasion attack, a number of adaptions have been implemented. In the next sections, the structure of the added modules in Fedstellar will be presented through three different modules.



(a) Normal version of the Fedstellar front-end.



(b) Experimental version of the Fedstellar front-end.

Figure 4.3: Different versions of the Fedstellar front-end.

4.2.1 Front-end

Fedstellar is a DFL platform with a user-friendly interface, based on which a module for black-box attack is added (Figure 4.3a), and the attack methods that the user can choose from are SimBA with Random Order (`SimBA_Random`), Square Attack(`Square attack`), Boundary Attack(`Boundary attack`), HopSkipJumpAttack with l_2 norm(`HSJA_L2`), and HopSkipJumpAttack with l_∞ norm(`HSJA_Linf`). Users can select different datasets, models, topologies, and number of nodes, among other parameters for FL, and perform attacks on the trained models. An experimental version (Figure 4.3b) is included to ensure accuracy and minimize errors in the experiments. Users can conduct all attacks on a model after it has been trained, rather than performing FL before each attack, ensuring consistency across all attacks on the same model.

Listing 4.1: Code for the AdversarialSampleGenerator class

```
class AdversarialSampleGenerator:
    def __init__(self, model, dataloader):
        self.model = model
        self.dataloader = dataloader
        self.attack_methods = {
            "SimBA": (SimBA, {}),
            "Square attack": (SquareAttack, {}),
            "Boundary attack": (BoundaryAttack, {}),
            "HSJA_L2": (HSJA, {"norm": 2}),
            "HSJA_Linf": (HSJA, {"norm": "inf"})
        }

    def generate_adversarial_samples(self, attack_method):
        if attack_method not in self.attack_methods:
            raise ValueError(f"Unknown attack method: {
                attack_method}")

        attack_class, attack_params = self.attack_methods[
            attack_method]
        attacker = attack_class(self.model, self.dataloader, **
            attack_params)

        return attacker.attack()
```

4.2.2 Attack Performing Module

The attack module is the most important one, which is mainly responsible for generating adversarial samples by attacking the trained model and test dataloader using the attack method selected by the user in the front-end after the FL process is completed. The `AdversarialSampleGenerator`(Listing 4.1) is first instantiated by passing in the model and training dataloader, `generate_adversarial_samples` method in `AdversarialSampleGenerator` is then called to conduct the attack. The advantage of this structure is to abstract as much as possible the common parameters of all attack methods, such as `model` and `dataloader`, to avoid redundant code. In addition to this, any other parameters, if any, can be passed to the corresponding attack method via `attack_params`. Using `AdversarialSampleGenerator` enhances the code's extensibility and maintainability, allowing new attack methods to be directly added to the `self.attack_methods` dictionary which allows new attack methods to be implemented without changing other code.

4.2.3 Logging Module

The logging module is mainly concerned with displaying the current process and showing the final results. In the same way as the results of FL are displayed on Fedstellar, the results of the black-box attack are also displayed in the Black-boxAttack fold-out page

Figure 4.4: Black-box Attack Performance Logging in Fedstellar Tensorboard

(Figure 4.4) of Fedstellar's TensorBoard. In addition to this, the results of the black-box attack are also recorded in the logs of each node (e.g. *participant_0.log*, *participant_1.log*, etc.), which is aimed at the purpose that we are not convenient to record the metrics of each attack method directly from Fedstellar's TensorBoard but can easily record the metrics of each attack method from the node's logs by writing scripts. The time needed for black-box attacks differs across datasets and models. To better monitor the attack process, records will be logged in the node's file after every 10 attack batches. Listing 4.2 briefly depicts the node logs before the black box attack, while the black box attack is in progress and after the black box attack is completed.

Listing 4.2: Example of a node log *participant_x.log*, x represents the different nodes involved in the federated learning process.

```
2024-07-28 16:22:46,965 - Prepare black-box attack with HSJA_L2.
2024-07-28 16:22:46,967 - start hsja.....
2024-07-28 16:22:46,970 - norm = 2
2024-07-28 16:22:47,056 - Processing batch 0 / 312
......
......
2024-07-28 16:24:25,707 - Processing batch 10 / 312
......
......
2024-07-28 16:27:25,707 - Processing batch 310 / 312
2024-07-28 16:28:46,578 - Complete black-box attack with HSJA_L2.
2024-07-28 16:28:46,961 - Accuracy = 0.014829405583441257,
   Precision = 0.03971002623438835, Recall = 0.014839405583441257,
    F1 Score = 0.0.01766916923224926
```

4.3 Implementation of the Attacks

4.3.1 SimBA

Simple Black-box Attack (SimBA) is a simple and effective attack with only 20 lines of Pytorch code. It's pseudo-code has already mentioned in Figure **??**. In order to improve the efficiency and effectiveness of adversarial sample generation, the SimBA method was rewritten and successfully integrated into the Fedstellar platform. Listing 4.3 demonstrates the integration of SimBA with random order into Fedstellar to generate adversarial samples. This reformulation not only optimises the performance of the SimBA method, but also ensures its compatibility with the Fedstellar platform, which enables the nodes in the distributed environment to work efficiently and collaboratively to generate high-quality adversarial samples, thus further enhancing the robustness and security of the FL model. In summary, the SimBA attack process can be divided into the following three steps:

Step1: Preparation phase before each SimBA batch.(2 to 16 lines of code in Listing 4.3.) Firstly, initialise the empty lists `x_adv_list` and `y_list` for storing the adversarial samples and labels respectively, obtain the input data `x` and labels `y` by iterating through the batches in the data loader and clone the input data to initialise the adversarial samples `x_adv`. Since SimBA is a scored-based attack method, it is necessary to compute the predicted probability by using the `softmax` method and assign it to `y_prob_pred`. it is also essential to obtain the predicted labels `y_i` by using the `argmax` method, which is mainly applied to make the model end the current attack after making an incorrect judgement and to avoid wasting resources.

Step2: Preparation phase before each SimBA sample.(18 to 35 lines of code in Listing 4.3.) In this step, it is firstly required to set the `desired_label` as the label initially predicted by the model, in untargeted attack, when the `current_label` is not equal to the `desired_label`, it can be regarded as the success of this attack. Also, the predicted probability (`last_prob`) of the current sample on the initial label is obtained, and the direction of the attack is optimised and adjusted for the change of `last_prob` in the attack. It should be noted that since SimBA requires a large number of queries to ensure the success of the attack, it is very likely that the total number of dimensions `n_dims` of the current sample is less than `max_iter`. Therefore, the index list indices`indices` need to be randomly arranged and spliced by repeated generation to reach the required length. Also to prevent the values after adding the perturbation from going over the range of the original data, it is possible to ensure that the adversarial samples remain within a reasonable range after adding the perturbation by determining the minimum(`clip_min`) and maximum(`clip_max`) values of the input tensor.

Step3: Attack phase in SimBA.(39 to 82 lines of code in Listing 4.3.) First, initialize a zero perturbation `diff`, then update it based on the index list `indices`. Calculate the model outputs `left_logits` and `right_logits`, along with the corresponding probabilities `left_prob` and `right_prob` using the `softmax` method after adding and subtracting the perturbation. Compare the probabilities before and after the perturbation, selecting the direction that leads to a greater reduction in probability to update the adversarial

sample `x_adv`. The iteration stops if the current label is no longer the desired label or if the maximum number of iterations is reached.

Listing 4.3: Algorithm for SimBA with Random Order.

```
1
2  def generate(self):
3          x_adv_list = []
4          y_list = []
5
6          for batch in self.dataloader:
7
8              x, y = batch
9              x = x.to(self.device)
10             y = y.to(self.device)
11
12             x_adv = x.clone()
13             y_list.append(y)
14
15             y_logits = self.model.forward(x)
16             y_prob_pred = softmax(y_logits, dim=1)
17             y_i = torch.argmax(y_prob_pred, dim=1)
18
19             for i_sample in range(x.size(0)):
20                 desired_label = y_i[i_sample]
21
22                 current_label = y_i[i_sample]
23                 last_prob = y_prob_pred[i_sample][desired_label]
24                 init_prob = last_prob
25
26                 n_dims = torch.tensor(x[i_sample].shape).prod().
                       item()
27                 indices = torch.randperm(n_dims)[: self.max_iter]
28                 indices_size = len(indices)
29
30                 while indices_size < self.max_iter:
31                     tmp_indices = torch.randperm(n_dims)
32                     indices = torch.cat((indices, tmp_indices))[:
                           self.max_iter]
33                     indices_size = len(indices)
34
35                 clip_min = torch.min(x)
36                 clip_max = torch.max(x)
37
38                 term_flag = 0
39                 nb_iter = 0
40                 while term_flag == 0 and nb_iter < self.max_iter:
41                     diff = torch.zeros(n_dims)
42
43                     diff[indices[nb_iter]] = self.epsilon *
                           clip_max
```

```
44
45                      left_logits = self.model.forward(torch.clamp(x
                            [i_sample] - diff.view_as(x[i_sample]), min
                            =clip_min, max=clip_max).unsqueeze(0))
46                      left_prob = softmax(left_logits, dim=1).view
                            (-1)[desired_label]
47
48                      right_logits = self.model.forward(torch.clamp(
                            x[i_sample] + diff.view_as(x[i_sample]),
                            min=clip_min, max=clip_max).unsqueeze(0))
49                      right_prob = softmax(right_logits, dim=1).view
                            (-1)[desired_label]
50
51                      if left_prob < last_prob:
52                          if left_prob < right_prob:
53                              x_adv[i_sample] = torch.clamp(x[
                                    i_sample] - diff.view_as(x[i_sample
                                    ]), min=clip_min, max=clip_max)
54                              last_prob = left_prob
55                              current_label = torch.argmax(
                                    left_logits, dim=1)
56                          else:
57                              x_adv[i_sample] = torch.clamp(x[
                                    i_sample] + diff.view_as(x[i_sample
                                    ]), min=clip_min, max=clip_max)
58                              last_prob = right_prob
59                              current_label = torch.argmax(
                                    right_logits, dim=1)
60                      else:
61                          if right_prob < last_prob:
62                              x_adv[i_sample] = torch.clamp(x[
                                    i_sample] + diff.view_as(x[i_sample
                                    ]), min=clip_min, max=clip_max)
63                              last_prob = right_prob
64                              current_label = torch.argmax(
                                    right_logits, dim=1)
65
66                      if desired_label != current_label:
67                          term_flag = 1
68
69              x_adv_list.append(x_adv)
70
71          x_adv_tensor = torch.cat(x_adv_list)
72          y_tensor = torch.cat(y_list)
73
74          return x_adv_tensor, y_tensor
```

Listing 4.4: Preparation phase before each Square Attack batch.

```
1 def square_attack(self):
```

```
2
3           x_adv_list = []
4           y_list = []
5
6           # Step1: Preparation phase before each Square Attack batch
                .
7           for batch in self.dataloader:
8
9               x, y = batch
10              x = x.to(self.device)
11              y = y.to(self.device)
12
13              x_adv = x.clone()
14
15              def adv_criterion(y_pred, y):
16                  return y_pred != y
17
18              if self.channels_first:
19                  channels = x.shape[1]
20                  height = x.shape[2]
21                  width = x.shape[3]
22              else:
23                  height = x.shape[1]
24                  width = x.shape[2]
25                  channels = x.shape[3]
26              y_adv = y[torch.randperm(y.size(0))]
27
28              clip_min = torch.min(x)
29              clip_max = torch.max(x)
30              ......
31              ......
```

4.3.2 Square Attack

Square Attack is an image region-based attack method that randomly selects a square region on the image to be perturbed each time. Square Attack has a larger and more localized perturbation compared to SimBA, quickly identifying effective attack points by gradually increasing the perturbed area. In the next few paragraphs, the attack process of Square Attack will be described in detail.

Step1: Preparation phase before each Square Attack batch.(Listing 4.4) Rather similarly to SimBA, Square Attack also needs to initialise the list of adversarial samples (`x_adv_list`) and labels (`y_list`) as it iterates through each batch in the `data_loader`, as well as determining the minimum (`clip_min`) and maximum (`clip_max`) values that will prevent values from falling beyond the range of the original data after the addition of the perturbation, based on the original samples `x`. In addition to this, the values of `height` and `width` need to be determined by determining whether is `channels_first`,

as the shape to be perturbed is a rectangle. Different DL frameworks or models may use different tensor layout methods, commonly channels first and channels last. In the channels first format, the shape of the image data is usually `[batch_size, channels, height, width]`, which is commonly used in the PyTorch framework, while in channels last format, the image data is usually in the shape of `[batch_size, height, width, channels]`, which is commonly used in the TensorFlow or Keras frameworks.

Listing 4.5: The pre-attack phase of Square Attack.

```
def square_attack(self):

        # Step1: Preparation phase before each Square Attack batch
            .
        for batch in self.dataloader:
        ......
        ......
        # Step2: The pre-attack phase of Square Attack.
        for _ in range(self.nb_restarts):
            y_pred = torch.argmax(self.model.forward(x_adv), dim
                =1)
            sample_is_robust = adv_criterion(y_pred, y)
            if torch.sum(sample_is_robust) == 0:
                break

            x_robust = x[sample_is_robust]
            y_robust = y[sample_is_robust]
            sample_loss_init = self._get_logits_diff(x_robust,
                y_robust)

            if self.channels_first:
                size = (x_robust.shape[0], channels, 1, width)
            else:
                size = (x_robust.shape[0], 1, width, channels)

            x_robust_new = torch.clamp(
                x_robust + self.eps * (2 * torch.randint(low=0,
                    high=2, size=size, dtype=x_robust.dtype, device
                    =self.device) - 1),
                min=clip_min,
                max=clip_max
            )

            sample_loss_new = self._get_logits_diff(x_robust_new,
                y_robust)
            loss_improved = (sample_loss_new - sample_loss_init) <
                0.0

            x_robust[loss_improved] = x_robust_new[loss_improved]
            x_adv[sample_is_robust] = x_robust
```

Step2: The pre-attack phase of Square Attack.(Listing 4.5) The reason why this phase is called pre-attack phase is that the `nb_restarts` parameter is set in Square Attack. When `max_iter` is reached in the attack but no adversarial samples are generated, the attack is restarted based on the `nb_restarts` parameter set before to increase the chances of successfully generating adversarial samples. Before the formal attack, firstly, use the model to predict the adversarial samples `x_adv` to get the prediction labels `y_pred`, and then call the `adv_criterion` method is used to determine which samples are still robust (i.e., not successfully attacked). If all samples are successfully attacked (`sample_is_robust` is `False`), exit the loop immediately. Otherwise, the robust samples `x_robust` and the corresponding labels `y_robust` are extracted from the original data, and the initial loss of the robust samples is computed by the `sample_loss_init` method, which is used to compare the effect before and after the perturbation. Before generating a new perturbation `x_robust_new`, the perturbation shape `size` is set according to `channels_first` to ensure it matches the original sample perturbation. After the new perturbation `x_robust_new` is determined, the `_get_logits_diff` method is called to determine if the new loss is lower than the initial loss, i.e., if the perturbation succeeded in reducing the loss. Finally, the portion of samples for which the perturbation was successful is updated in `x_adv`.

Listing 4.6: Attack phase in Square Attack.

```
def square_attack(self):

    # Step1: Preparation phase before each Square Attack batch.
    for batch in self.dataloader:
    ......
    ......
        # Step2: The pre-attack phase of Square Attack.
        for _ in range(self.nb_restarts):
            ......
            ......
            # Step3: Attack phase in Square Attack
            for i_iter in range(self.max_iter):
                percentage_of_elements = self.
                    _get_percentage_of_elements(i_iter)

                sample_is_robust = adv_criterion(torch.argmax(self
                    .model.forward(x_adv), dim=1), y)

                if torch.sum(sample_is_robust) == 0:
                    break

                x_robust = x_adv[sample_is_robust]
                x_init = x[sample_is_robust]
                y_robust = y[sample_is_robust]
                sample_loss_init = self._get_logits_diff(x_robust,
                    y_robust)

                height_tile = max(int(round(math.sqrt(
                    percentage_of_elements * height * width))), 1)
                height_mid = torch.randint(0, height - height_tile
```

```
                    , (1,)).item()
27                  width_start = torch.randint(0, width - height_tile
                    , (1,)).item()
28
29                  delta_new = torch.zeros_like(x_robust)
30                  if self.channels_first:
31                      delta_new[:, :, height_mid:height_mid +
                        height_tile, width_start:width_start +
                        height_tile] = (
32                          torch.randint(0, 2, size=(channels, 1, 1),
                                device=self.device, dtype=torch.
                            float32) * 4 * self.eps - 2 * self.eps
33                      )
34                  else:
35                      delta_new[:, height_mid:height_mid +
                        height_tile, width_start:width_start +
                        height_tile, :] = (
36                          torch.randint(0, 2, size=(1, 1, channels),
                                device=self.device, dtype=torch.
                            float32) * 4 * self.eps - 2 * self.eps
37                      )
38
39                  x_robust_new = x_robust + delta_new
40
41                  x_robust_new = torch.min(torch.max(x_robust_new,
                    x_init - self.eps), x_init + self.eps)
42
43                  x_robust_new = torch.clamp(x_robust_new, min=
                    clip_min, max=clip_max)
44
45                  sample_loss_new = self._get_logits_diff(
                    x_robust_new, y_robust)
46                  loss_improved = (sample_loss_new <
                    sample_loss_init)
47
48                  x_robust[loss_improved] = x_robust_new[
                    loss_improved]
49                  x_adv[sample_is_robust] = x_robust
50
51          x_adv_list.append(x_adv)
52          y_list.append(y_adv)
53      return torch.cat(x_adv_list), torch.cat(y_list)
```

Step3: Attack phase in Square Attack. (Listing 4.6) This phase implements the core part of Square Attack, where adversarial samples are generated by adding perturbations to different regions of the image. This process gradually adjusts the size and position of the perturbed regions until the maximum number of iterations is reached or all samples are successfully attacked. The steps are as follows: first, the percentage of perturbed regions for the current iteration is derived using the `_get_percentage_of_elements` method,

which returns the percentage of perturbed regions based on the number of iterations. Subsequently, similar to the pre-attack phase, the model is used to predict the current adversarial samples `x_adv` to check if all samples have been successfully attacked. When there are samples that have not yet been successfully attacked, further attacks are performed and the initial loss `sample_loss_init` of the robust samples is recorded. The size of the perturbed region for the current iteration `height_tile` is determined by the previously computed `percentage_of_elements`, and the currently perturbed region gets smaller and smaller as the attack progresses. The perturbed region's starting positions, `height_mid` and `width_start`, are randomly chosen from `height_tile`. This is followed by the generation of the perturbation by the Square Attack, which adds a square perturbation within the range $[-2\epsilon, 2\epsilon]$ at a random position in the image based on the channel order (channels first or channels last) on an initialised all-zero perturbation tensor `delta_new`. Add the new perturbation to the robust sample, generate a new adversarial sample `x_robust_new` and use `torch.clamp` to crop the result to the range `[clip_min, clip_max]` to ensure that the perturbed value does not fall outside the range of the original data. When complete the attack, calculate whether the new loss `sample_loss_new` after adding the perturbation is lower than the initial loss `sample_loss_init`, i.e., whether the perturbation succeeded in reducing the loss. If the perturbation succeeds in reducing the loss, update the robust sample `x_robust` to the new perturbed sample `x_robust_new`.

Listing 4.7: `_get_logits_diff` method in Square Attack.

```
1
2  def _get_logits_diff(self, x, y):
3
4      y_pred = softmax(self.model.forward(x))
5      logit_correct = torch.gather(y_pred, 1, y.unsqueeze(1))
6
7      sorted_logits, _ = y_pred.sort(dim=1, descending=True)
8      logit_highest_incorrect = sorted_logits[:, 1:2]
9
10     return (logit_correct - logit_highest_incorrect).squeeze()
```

Listing 4.8: `_get_percentage_of_elements` method in Square Attack.

```
1
2  def _get_percentage_of_elements(self, i_iter):
3
4      i_p = i_iter / self.max_iter
5      intervals = [0.001, 0.005, 0.02, 0.05, 0.1, 0.2, 0.4, 0.6,
           0.8]
6      p_ratio = [1, 1 / 2, 1 / 4, 1 / 8, 1 / 16, 1 / 32, 1 / 64, 1 /
           128, 1 / 256, 1 / 512]
7      i_ratio = bisect.bisect_left(intervals, i_p)
8
9      return self.p_init * p_ratio[i_ratio]
```

Other methods used in Square Attack In addition to the aforementioned attack methods, Square Attack includes other methods to help perform the attack. `_get_logits_diff`

(Listing 4.7) implements a method for calculating differences in logits, which is primarily used to evaluate the effectiveness of counter samples. Specifically, it calculates the difference between the logit of the correct category and the highest logit of the wrong category for each sample. The goal attack is to reduce this difference so that the model has less confidence in the correct category, thus increasing the likelihood of misclassification. Another method is called `_get_percentage_of_elements` (Listing 4.8), which is used to dynamically adjust the percentage of perturbation based on the current number of iterations. During the adversarial attack, the proportion of perturbation decreases as the number of iterations increases, thus refining the impact of the perturbation. This approach allows the use of larger perturbations in the early stages of the attack to quickly reduce model confidence, and smaller perturbations in the later stages of the attack to further fine-tune and optimise the effect of the attack.

4.3.3 Boundary Attack

The main idea of Boundary Attack is to find an adversarial sample by gradually adjusting the initial sample so that it is close to the decision boundary. The method first generates a random initial sample with a different class from the original sample, then adds orthogonal perturbations in each iteration and gradually reduces the magnitude of the perturbations. The next overview of the whole attack process is presented through the methods used in the implementation of Boundary Attack.

Listing 4.9: `boundary_attack` method in Boundary Attack.

```
1
2  def boundary_attack(self):
3
4      x_adv_list = []
5      y_list = []
6
7      for batch in self.dataloader:
8
9          x, y = batch
10         x, y = x.to(self.device), y.to(self.device)
11         x_adv = x.clone()
12         y_list.append(y)
13
14         preds = torch.argmax(self.model.forward(x), dim=1)
15
16         x_adv = x_adv.clone()
17         clip_min, clip_max = torch.min(x), torch.max(x)
18
19         for ind, val in enumerate(x_adv):
20             x_adv[ind] = self._perturb(
21                 x=val,
22                 y=-1,  # For untargeted attack
23                 y_p=preds[ind],
24                 clip_min=clip_min,
```

```
                clip_max=clip_max
            )

        x_adv_list.append(x_adv)
    return torch.cat(x_adv_list), torch.cat(y_list)
```

`boundary_attack` **method (Listing 4.9)** This part of code implements the main loop of the Boundary Attack, which calls other methods in the `boundary_attack` method to accomplish the whole attack, and prepares the parameters used in the Boundary Attack, including the storage of adversarial samples `x_adv_list` and labels `y_list`, as well as the minimum value `clip_min` and maximum value `clip_max` for later perturbation trimming, and so on. Subsequently, the `_perturb` method is invoked to generate the adversarial samples by iterating over each sample `x_adv` in the current batch.

Listing 4.10: `_perturb` method in Boundary Attack.

```

def _perturb(self, x, y, y_p, clip_min, clip_max):
    initial_sample = self._init_sample(x, y, y_p, clip_min,
       clip_max)

    if initial_sample is None:
        return x

    x_adv = self._attack(initial_sample, x, y_p, self.delta, self.
       epsilon, clip_min, clip_max)

    return x_adv
```

`_perturb` **method (Listing 4.10)** This method is used to generate a single adversarial sample. The `_init_sample` method is called first to generate an initial perturbation sample `initial_sample`. If the initial perturbation sample fails to be generated, the original sample `x` is returned, otherwise `_attack` method is called to optimise the initial perturbation sample and create the final adversarial sample `x_adv`.

Listing 4.11: `_init_sample` method in Boundary Attack.

```

def _init_sample(self, x, y, y_p, clip_min, clip_max):
    nprd = torch.Generator().manual_seed(0)
    initial_sample = None

    # The initial image unsatisfied
    for _ in range(self.init_size):
        random_img = torch.empty_like(x).uniform_(clip_min,
           clip_max, generator=nprd)
        random_class = torch.argmax(self.model.forward(random_img.
           unsqueeze(0)), dim=1).item()

        if random_class != y_p:
            initial_sample = random_img
```

```
13             break
14
15     return initial_sample
```

`_init_sample` method (Listing 4.11) `_init_sample` is a method that finds a valid initial sample of perturbations by generating random images and ensuring that the predicted categories of these images are different from the predicted classes of the original sample. At first, a random number generator `nprd` is created and the seed is set to 0 to ensure that the random numbers generated are repeatable. Loop `init_size` times, each time generating a random image `random_img` with the same shape as the input sample `x`, whose values are uniformly distributed in the range `[clip_min, clip_max]`. Then, use the model to predict the random image to get the prediction class `random_class`, and if it is different from the prediction class `y_p` of the original sample, use that random image as the initial perturbation sample `initial_sample` and exit the loop.

Listing 4.12: `_attack` method in Boundary Attack.

```
1
2  def _attack(self, initial_sample, original_sample, y_p,
       initial_delta, initial_epsilon, clip_min, clip_max):
3      x_adv = initial_sample.clone()
4      self.curr_delta = initial_delta
5      self.curr_epsilon = initial_epsilon
6
7      self.curr_adv = x_adv
8
9      for _ in range(self.max_iter):
10         for _ in range(self.num_trial):
11             potential_advs_list = []
12             for _ in range(self.sample_size):
13                 potential_adv = x_adv + self._orthogonal_perturb(
                       self.curr_delta, x_adv, original_sample)
14                 potential_adv = torch.clamp(potential_adv,
                       clip_min, clip_max)
15                 potential_advs_list.append(potential_adv)
16
17             preds = torch.argmax(self.model.forward(torch.stack(
                   potential_advs_list)), dim=1)
18             satisfied = preds != y_p
19
20             delta_ratio = torch.mean(satisfied.float()).item()
21
22             if delta_ratio < 0.2:
23                 self.curr_delta *= self.step_adapt
24             elif delta_ratio > 0.5:
25                 self.curr_delta /= self.step_adapt
26
27             if delta_ratio > 0:
28                 x_advs = torch.stack(potential_advs_list)[
                       satisfied]
```

```
29                 break
30             return x_adv
31
32         for _ in range(self.num_trial):
33             perturb = (original_sample - x_advs) * self.
                   curr_epsilon
34             potential_advs = x_advs + perturb
35             potential_advs = torch.clamp(potential_advs, clip_min,
                    clip_max)
36             preds = torch.argmax(self.model.forward(potential_advs
                   ), dim=1)
37             satisfied = preds != y_p
38
39             epsilon_ratio = torch.mean(satisfied.float()).item()
40
41             if epsilon_ratio < 0.2:
42                 self.curr_epsilon *= self.step_adapt
43             elif epsilon_ratio > 0.5:
44                 self.curr_epsilon /= self.step_adapt
45
46             if epsilon_ratio > 0:
47                 x_adv = self._best_adv(original_sample,
                       potential_advs[satisfied])
48                 self.curr_adv = x_adv
49                 break
50             return self._best_adv(original_sample, x_advs)
51
52         if self.curr_epsilon < self.min_epsilon:
53             return x_adv
54
55     return x_adv
```

`_attack` method (Listing 4.12) This piece of code implements the `_attack` method of Boundary Attack, which is used to generate the final adversarial samples by adjusting the perturbations to the initial perturbation samples through several iterations, gradually approaching the decision boundary. After initialising the adversarial samples and perturbation parameters, iterative optimisation begins. The first step is to adjust `delta` at the first inner loop, in which a set of potential adversarial samples is generated for each trial, and orthogonal perturbations are generated using the `_orthogonal_perturb` method and clipped to the `[clip_min, clip_max]` range. Subsequently, the generated potential adversarial samples are predicted to obtain the prediction labels `preds`, and `satisfied` is obtained by comparing the original labels `y_p`. Finally, `satisfied` is used to calculate `delta_ratio`, the proportion of the prediction changed by the perturbation. And the current `delta` is adjusted according to the value of `delta_ratio`. This serves to try to fine-tune the input samples in all directions that are closer to the decision boundary, if the `delta_ratio` is too small it means that very few perturbation samples succeeded in changing the model's prediction, indicating that `delta` is too large and the perturbation is too large, thus it is necessary to reduce the `delta`. Conversely, `delta` needs to be increased to explore a wider range. The second internal loop is used to adjust `epsilon`

in a similar way as adjusting delta, and eventually adjusts the current `epsilon` based on the value of `epsilon_ratio`. If `epsilon_ratio` is too small, it indicates that few perturbation samples succeed in changing the model's prediction, which suggests that `epsilon` and the size of the perturbation is too large. So that `epsilon` needs to be reduced for finer tuning, and conversely, that `epsilon` needs to be increased for more extensive exploration. It is worth noting that if `epsilon_ratio` larger than zero, it means that there is at least one perturbation sample that has successfully changed the model's prediction. In this case, `_best_adv` method is called to select the adversarial sample `x_adv` from the successfully perturbed samples that is closest to the original sample. Otherwise, there are no successfully perturbed samples, the best adversarial sample `x_adv` can only be selected from all potential adversarial samples.

Listing 4.13: `_orthogonal_perturb` method in Boundary Attack.

```
1
2 def _orthogonal_perturb(self, delta, current_sample,
     original_sample):
3     perturb = torch.randn_like(current_sample)
4     perturb /= torch.norm(perturb)
5     perturb *= delta * torch.norm(original_sample - current_sample
         )
6
7     direction = original_sample - current_sample
8     direction_flat = direction.view(-1)
9     perturb_flat = perturb.view(-1)
10
11    direction_flat /= torch.norm(direction_flat)
12    perturb_flat -= torch.dot(perturb_flat, direction_flat) *
         direction_flat
13    perturb = perturb_flat.view_as(current_sample)
14
15    hypotenuse = torch.sqrt(torch.tensor(1.0 + delta ** 2))
16    perturb = ((1 - hypotenuse) * (current_sample -
         original_sample) + perturb) / hypotenuse
17    return perturb
```

`_orthogonal_perturb` method(Listing 4.13) The `_orthogonal_perturb` method is used to generate orthogonal perturbations. An orthogonal perturbation is a change between the original and current samples that is orthogonal to the direction from the original sample to the current sample. In this way, the effectiveness of the perturbation can be increased while maintaining the similarity between the current sample and the original sample. This is achieved by first generating a random tensor `perturb` with the same shape as `current_sample` and normalising it to have a unit norm. Then calculate the direction vector `direction` from `current_sample` to *original_sample*. The `direction` and `perturb` are converted to one-dimensional tensors `direction_flat` and `perturb_flat` for further calculations. The so-called orthogonalised perturbation, in this piece of code, is to standardise the direction vector `direction_flat` by subtracting its projection on `direction_flat` from `perturb_flat` so that it is orthogonal to `direction_flat`, and then reconvert the orthogonalised `perturb_flat` back to its original shape. The final perturbation `perturb` returned is a magnitude-adjusted and orthogonalised tensor.

Listing 4.14: `_best_adv` method in Boundary Attack.

```
1
2 def _best_adv(self, original_sample, potential_advs):
3     shape = potential_advs.shape
4     min_idx = torch.norm(original_sample.flatten() -
         potential_advs.view(shape[0], -1), dim=1).argmin()
5     return potential_advs[min_idx]
```

`_best_adv` method (Figure 4.14) This method is used to select the closest adversarial sample to the original sample from a set of potential adversarial samples. Selecting the

nearest adversarial sample ensures that the perturbation is as small as possible while still achieving the goal of countering the attack. The basic theorem in this implementation is to find the index `min_idx` of the sample with the smallest distance by calculating the Euclidean distance between `original_sample` and each flattened potential adversary sample. In the end, the final output(`potential_advs[min_idx]`) is based on the index `min_idx`.

4.3.4 HopSkipJumpAttack(HSJA)

Boundary Attack and HopSkipJumpAttack (HSJA) are both decision-based black-box attack methods. The difference is that Boundary Attack approximates the decision boundary by gradually adding orthogonal perturbations, while HSJA optimises the samples by computing perturbation updates and binary search. Compared to Boundary Attack, HSJA adopts a more systematic approach to select the optimal perturbations in each iteration, further improving the efficiency and effectiveness of the attack. In the following, a detailed description of the attack process of HSJA will be presented with each method used in HSJA.

Listing 4.15: `hsja` method in HopSkipJumpAttack(HSJA).

```
1
2  def hsja(self):
3      x_adv_list = []
4      y_list = []
5
6      for batch in self.dataloader:
7
8          x, y = batch
9          x, y = x.to(self.device), y.to(self.device)
10         y_list.append(y)
11
12         if self.theta is None:
13             input_shape = x.shape[1:]
14             if self.norm == 2:
15                 self.theta = 0.01 / torch.sqrt(torch.prod(torch.
                       tensor(input_shape, device=self.device)))
16             else:
17                 self.theta = 0.01 / torch.prod(torch.tensor(
                       input_shape, device=self.device))
18
19         preds = torch.argmax(self.model.forward(x), dim=1)
20
21         init_preds = [None] * len(x)
22         x_adv_init = [None] * len(x)
23         x_adv = x.clone()
24         clip_min, clip_max = torch.min(x), torch.max(x)
25
26         for ind, val in enumerate(x):
27             self.curr_iter = 0
```

```
28
29              x_adv[ind] = self._perturb(x=val,
30                                         y_p=preds[ind],
31                                         init_pred=init_preds[ind],
32                                         adv_init=x_adv_init[ind],
33                                         clip_min=clip_min,
34                                         clip_max=clip_max)
35
36          x_adv_list.append(x_adv)
37
38      return torch.cat(x_adv_list), torch.cat(y_list)
```

`hsja` method (Listing 4.15) The role of this code is similar to the `boundary_attack` method in Boundary Attack, which is mainly responsible for iterating through each batch of data in the data loader and generating the corresponding adversarial samples for each sample by calling `_perturb`. In addition, an initial perturbation parameter `theta` is computed and set according to the shape and the type of paradigm (l_2 norm or l_∞ norm) of the input samples. In HopSkipJumpAttack (HSJA), `theta` is computed differently for different norms. For l_2 norm, the size of the perturbation is related to the dimension of the input samples. Therefore, to ensure that the perturbation is within a reasonable range, `theta` needs to be divided by the square root of the dimension.The l_∞ paradigm measures the maximum absolute value of the vector elements, i.e., the maximum perturbation among all elements is considered. The size of its perturbation is independent of the dimension of the input sample, as it only considers the maximum perturbation of a single element. Therefore, `theta` only need be divided by the product of the dimensions.

Listing 4.16: `_perturb` method in HopSkipJumpAttack(HSJA).

```
1
2 def _perturb(self, x, y_p, init_pred, adv_init, clip_min, clip_max
     ):
3     initial_sample = self._init_sample(x, y_p, init_pred, adv_init
         , clip_min, clip_max)
4
5     if initial_sample is None:
6         return x
7
8     x_adv = self._attack(initial_sample[0], x, initial_sample[1],
         clip_min, clip_max)
9
10    return x_adv
```

`_perturb` method (Listing 4.16) The major purpose of the `_perturb` method is to generate an adversarial sample. It first generates an initial perturbation sample through the `_init_sample` method. If the generation fails, the original sample is returned. Otherwise, the final adversarial sample is further optimised by the `_attack` method. The `_init_sample` method and the `_attack` method are explained in detail in the next paragraphs.

Listing 4.17: `_init_sample` method in HopSkipJumpAttack(HSJA).

```
1
2 def _init_sample(self, x, y_p, init_pred, adv_init, clip_min,
     clip_max):
3     nprd = torch.Generator().manual_seed(0)
4     initial_sample = None
5
6     if adv_init is not None and init_pred != y_p:
7         return adv_init, y_p
8
9     for _ in range(self.init_size):
10        random_img = torch.empty_like(x).uniform_(clip_min,
              clip_max, generator=nprd)
11        random_class = torch.argmax(self.model.forward(random_img.
              unsqueeze(0)), dim=1).item()
12
13        if random_class != y_p:
14            random_img = self._binary_search(random_img, x, y_p,
                  self.norm, clip_min, clip_max, threshold=0.001)
15            initial_sample = random_img, y_p
16            break
17
18    return initial_sample
```

`_init_sample` method (Listing 4.17) Similarly to the `_init_sample` method in Boundary Attack, the `_init_sample` method in HSJA aims to finds a valid initial perturbation sample by generating random images and ensuring that the predicted classes of these images are different from the predicted classes of the original sample. The difference is that the initial perturbation sample `random_img` is further optimised in HSJA by calling the `_binary_search` method if the `random_class` is different from the predicted class `y_p` of the original sample.

Listing 4.18: `_attack` method in HopSkipJumpAttack(HSJA).

```
1
2 def _attack(self, initial_sample, original_sample, target,
     clip_min, clip_max):
3     current_sample = initial_sample.clone()
4
5     for _ in range(self.max_iter):
6         delta = self._compute_delta(current_sample,
              original_sample, clip_min, clip_max)
7         current_sample = self._binary_search(current_sample,
              original_sample, target, self.norm, clip_min, clip_max)
8         num_eval = min(int(self.init_eval * (self.curr_iter + 1)
              ** 0.5), self.max_eval)
9
10        update = self._compute_update(current_sample, num_eval,
              delta, target, clip_min, clip_max)
11
```

```
12          if self.norm == 2:
13              dist = torch.norm(original_sample - current_sample)
14          else:
15              dist = torch.max(torch.abs(original_sample -
                    current_sample))
16
17          epsilon = 2.0 * dist / ((self.curr_iter + 1) ** 0.5)
18          success = False
19
20          while not success:
21              epsilon /= 2.0
22              potential_sample = current_sample + epsilon * update
23              success = self._adversarial_satisfactory(
                    potential_sample.unsqueeze(0), target, clip_min,
                    clip_max)
24
25          current_sample = torch.clamp(potential_sample, clip_min,
                clip_max)
26          self.curr_iter += 1
27
28          if torch.isnan(current_sample).any():
29              return original_sample
30
31      return current_sample
```

`_attack` method (Listing 4.18) The method is a continuous process of adding perturbations to `current_sample` and optimising them. Therefore, the first thing that should be determined is the size of the perturbation. The range `delta` of the current perturbation can be calculated by calling the `_compute_delta` method. The `num_eval` determines how many randomly orientated samples of the perturbation will be generated and evaluated when generating the potential adversarial samples. The calculated `num_eval` is passed into the `_compute_update` method to calculate the perturbation `update` for the current sample. Subsequently, the initial step size `epsilon` is computed depending on the norm applied (l_2 norm or l_∞). With respect to the `current_sample` optimisation process, `epsilon` is halved at each step, update `potential_sample` and check whether it satisfies the condition. Finally, return the valid cropped adversarial sample.

Listing 4.19: `_compute_delta` method in HopSkipJumpAttack(HSJA).

```
1
2 def _compute_delta(self, current_sample, original_sample, clip_min
      , clip_max):
3       if self.curr_iter == 0:
4           return 0.1 * (clip_max - clip_min)
5
6       if self.norm == 2:
7           dist = torch.norm(original_sample - current_sample).item()
8           delta = (torch.prod(torch.tensor(current_sample.shape[1:],
                device=self.device)).sqrt() * self.theta * dist).item
               ()
```

```
9       else:
10          dist = torch.max(torch.abs(original_sample -
                current_sample)).item()
11          delta = (torch.prod(torch.tensor(current_sample.shape[1:],
                device=self.device)) * self.theta * dist).item()
12
13      return delta
```

`_compute_delta` method (Listing 4.19) `_compute_delta` method for calculating the perturbation size `delta` in the current iteration. This method dynamically adjusts the perturbation size based on the distance between the current sample and the original sample and the type of paradigm (l_2 norm or l_∞ norm) to ensure that the perturbation is within a reasonable range. If this is the first iteration, the initial perturbation size $0.1 \times (clip_max - clip_min)$ is returned directly. The point of this is to provide a baseline value for the initial perturbation to ensure that there is a sufficient amount of perturbation in the first iteration. The calculation of `delta` is shown in Equation 4.1 when the norm is l_2 and Equation 4.2 when the norm is l_∞.

Listing 4.20: `_binary_search` method in HopSkipJumpAttack(HSJA).

```
1
2   def _binary_search(self, current_sample, original_sample, target,
        norm, clip_min, clip_max, threshold=None):
3       if norm == 2:
4           upper_bound, lower_bound = torch.tensor(1.0, device=self.
                device), torch.tensor(0.0, device=self.device)
5           if threshold is None:
6               threshold = self.theta
7       else:
8           upper_bound = torch.max(torch.abs(original_sample -
                current_sample))
9           lower_bound = torch.tensor(0.0, device=self.device)
10          if threshold is None:
11              threshold = min(upper_bound * self.theta, self.theta)
12
13      while (upper_bound - lower_bound) > threshold:
14          alpha = (upper_bound + lower_bound) / 2.0
15          interpolated_sample = self._interpolate(current_sample,
                original_sample, alpha, norm)
16
17          satisfied = self._adversarial_satisfactory(
                interpolated_sample.unsqueeze(0), target, clip_min,
                clip_max)
18          if satisfied:
19              upper_bound = alpha
20          else:
21              lower_bound = alpha
22
23      result = self._interpolate(current_sample, original_sample,
            upper_bound, norm)
```

```
24      return result
```

`_binary_search` method (Listing 4.20) `_binary_search` is used to find an optimal adversarial sample between the current sample and the original sample. The method finds a valid adversarial sample by adjusting the interpolation coefficient `alpha` and gradually approximating the decision boundary. First of all, an upper and lower bound needs to be determined. For l_2 norm, the upper bound is set to 1.0, the lower bound is set to 0.0, and `threshold` is set to `self.theta`, and for l_∞ norm, the upper bound is set to the maximum difference between the original sample and the current sample, the lower bound is set to 0.0, and the threshold is set to the smaller one between $upper\_bound \times theta$ and `theta`. This is followed by the core code of `_binary_search`, it will enter the loops when the difference between the upper and lower bounds is greater than `threshold`, and calculates the interpolation coefficient `alpha`, which acts as the midpoint between the upper and lower bounds. Then call the `_interpolate` method, which interpolates between the current sample and the original sample based on `alpha`, generating an interpolated sample `interpolated_sample`. Subsequently, the `_adversarial_satisfactory` method is used to check whether the interpolated samples satisfy the adversarial condition, and if so, the upper bound is updated to `alpha`, otherwise, the lower bound is updated to `alpha`. Finally, an optimal adversarial sample result is found between the current sample and the original sample.

`_interpolate` method. The `_interpolate` method is used to perform an interpolation operation between the current sample and the original sample. For the l_2 norm, the new sample is computed using linear interpolation, as described in Equation 4.7. For the l_∞ norm, the new sample is computed using a trimming operation that restricts `current_sample` to between $original\_sample - alpha$ and $original\_sample + alpha$.

$$\text{result} = (1 - alpha) \times original\_sample + alpha \times current\_sample \tag{4.7}$$

Listing 4.21: `_compute_update` method in HopSkipJumpAttack(HSJA).

```

def _compute_update(self, current_sample, num_eval, delta, target,
    clip_min, clip_max):
    rnd_noise_shape = [num_eval] + list(current_sample.shape)
    if self.norm == 2:
        rnd_noise = torch.randn(rnd_noise_shape, device=self.
           device)
    else:
        rnd_noise = torch.empty(rnd_noise_shape, device=self.
           device).uniform_(-1, 1)

    rnd_noise = rnd_noise / torch.sqrt(torch.sum(rnd_noise ** 2,
       dim=tuple(range(1, len(rnd_noise_shape))), keepdim=True))
    eval_samples = torch.clamp(current_sample + delta * rnd_noise,
        clip_min, clip_max)
    rnd_noise = (eval_samples - current_sample) / delta

```

```
13      satisfied = self._adversarial_satisfactory(eval_samples,
            target, clip_min, clip_max)
14      f_val = 2 * satisfied.float().view(-1, 1, 1, 1) - 1
15
16      if torch.mean(f_val) == 1.0:
17          grad = torch.mean(rnd_noise, dim=0)
18      elif torch.mean(f_val) == -1.0:
19          grad = -torch.mean(rnd_noise, dim=0)
20      else:
21          f_val -= torch.mean(f_val)
22          grad = torch.mean(f_val * rnd_noise, dim=0)
23
24      if self.norm == 2:
25          result = grad / torch.norm(grad)
26      else:
27          result = torch.sign(grad)
28
29      return result
```

`_compute_update` **method (Listing 4.21**) The method enables the adversarial samples to approximate the decision boundary by generating random noise samples, evaluating their perturbation effects, and calculating the update direction. Firstly, the shape of the random noise `rnd_noise_shape` is generated based on the shapes of `num_eval` and `current_sample`. For the l_2 norm, the random noise `rnd_noise` is generated with a standard normal distribution, while for the l_∞ norm, it is generated with a uniform distribution between $[-1, 1]$. The normalised `rnd_noise` is used to compute the actual perturbation direction, which is later updated into `rnd_noise`. Subsequently, the evaluation samples are checked to ensure that the adversarial condition is fulfilled, and to conveniently perform subsequent calculations, `satisfied` is converted into a tensor (`f_val`) containing -1 and 1, where a value of 1 indicates a successful attack, and a value of -1 indicates a failed attack. If all evaluation samples are successful (the mean value of `f_val` is 1), calculate the mean value of the random noise as the gradient direction. If all the evaluation samples failed against it (the mean of `f_val` is -1), calculate the negative mean of the random noise as the direction of the gradient. Otherwise, adjust the weighted mean of the random noise according to the deviation of ‘`f_val` as the gradient direction. At last, for the l_2 norm, the normalised update direction `grad` is taken, whereas for the l_∞ norm, only take the sign of `grad` after the update direction as the result.

Chapter 5

Evaluation

5.1 Experimental Setup

The next sections will discuss the datasets and models used to evaluate black-box adversarial attacks and the black-box adversarial attack methods. Furthermore, parameters used in setting up Fedstellar [27] will also be explained.

5.1.1 Datasets and Models

In this work, three datasets are selected as attack objects to evaluate the performance of the attack, they are:

- **MNIST** is a widely used handwritten digit recognition dataset, released in 1998 by Yann LeCun *et al.* [33]. It contains 60,000 training images and 10,000 test images, each of which is a 28×28 pixel greyscale map representing handwritten digits from 0 to 9. This dataset is widely used in the fields of ML and computer vision, especially in training and testing various image classification algorithms. Due to its simplicity and ease of use, MNIST is considered an introductory dataset in the field of ML and image recognition. Researchers use MNIST to test the effectiveness of new algorithms and techniques because its preprocessing and labelling are very standardised, allowing researchers to focus more on the development and optimisation of the algorithms themselves .
- **Fashion-MNIST** [34] is an alternative image dataset to MNIST introduced in 2017 by the Zalando research team to provide a more challenging and realistic benchmark test dataset. It contains 60,000 training images and 10,000 test images, each of which is also a 28×28 pixel greyscale image, but the image content has been swapped from handwritten numbers to 10 different categories of fashion products, such as t-shirts, trousers, shoes, etc. Fashion-MNIST maintains the same image sizes and data formats as MNIST, and can therefore be used on the same model and code base. The main goal of this dataset is to advance the field of image classification beyond simple handwritten numbers to more complex real-world applications.

- **CIFAR-10** [35] is a benchmark dataset for image classification released by the Canadian Institute for Advanced Research (CIFAR) in 2009. It contains 60,000 colour images of 32x32 pixels, of which 50,000 are used for training and 10,000 for testing. The images are classified into 10 categories including planes, cars, birds, cats, deer, dogs, frogs, horses, boats, and trucks.CIFAR-10 has a much larger amount of data and complexity than MNIST and Fashion-MNIST because each image contains rich colour information and a more complex background, which makes CIFAR-10 one of the most important datasets for evaluating the performance of DL algorithms, especially convolutional neural networks (CNNs) .

In the area of ML and DL, the selection of the dataset plays a crucial role in the performance and effectiveness of the model. Therefore, after determining the dataset, the models used in this work are then introduced.

- **Multi-Layer Perceptron (MLP)** is a feed-forward artificial neural network consisting of at least three layers of nodes (an input layer, one or more hidden layers, and an output layer). Each node (also known as a neuron or perceptron) uses a nonlinear activation function to process inputs and passes information through weighted connections.The core concept of the MLP stems from the workings of the perceptron, the basic building block of a neural network, which was proposed by Rosenblatt [36] in 1957. The MLP is a model of supervised learning and is commonly used for classification and regression tasks. During training, the MLP uses a backpropagation algorithm to adjust the weights so as to minimise the prediction error. Backpropagation was proposed by Rumelhart, Hinton and Williams [37] in 1986, which optimises the network by calculating the gradient of the loss function with respect to each of the weights using gradient descent methods. The training of the MLP can be achieved by optimisation algorithms such as Stochastic Gradient Descent (SGD), Momentum, and Adaptive Learning Rate Methods (e.g. Adam).

- **Convolutional Neural Network (CNN)** is a DL model specifically designed to process data with a grid structure. The key to CNN is its convolutional and pooling layers, which are capable of efficiently extracting and processing the spatial and hierarchical features of an image. The concept of CNN was first proposed in 1980 by Fukushima [38] , who developed the Neocognitron for handwritten digit recognition, and then LeCun *et al.* [39] further developed this model for handwritten digit recognition with "LeNet-5" in 1998. Convolutional layer is the core component of CNN, which uses a set of convolutional kernels to slide over the input data and extract features through local connections and shared weights. Each convolutional kernel applies the same weights at different locations in the image, thus significantly reducing the number of parameters and computational complexity. Pooling layers (e.g. maximum pooling or average pooling) further reduce the size of the feature map by downsampling the local neighbourhood, reducing computation and controlling overfitting.

In the end, MLP is selected to process the MNIST dataset in this work because of its simple structure and suitability for low-complexity image tasks. For Fashion-MNIST

and CIFAR-10 datasets, CNN is selected.Fashion-MNIST data structure though similar to MNIST, but the image content is more complex and contains details and textures of various fashion items. The CIFAR-10 dataset is more complex compared to both of them.The convolution operation of CNN is able to extract multi-level features such as edges, textures and shapes, which is of great importance especially for dealing with high dimensional colour images.

5.1.2 Attack Methods

In this work, four attacks were selected and the following six attacks were determined through the use of different norms and the order of attacks. In the following, the selected attack methods are briefly described.

- **SimBA** (Section 4.3.1) is a black-box attack algorithm that optimises the performance of the attack by gradually adding small amounts of random noise to the image and evaluating the changes in the model output. In this work random order and diagonal order are used for the attacks.
- **Square Attack** (Section 4.3.2) is based on random search that maximises the misclassification probability of the target model by adding square perturbations to the image.
- **Boundary Attack** (Section 4.3.3) is a black-box method that uses iterative optimization to minimize perturbations for misclassification by gradually adjusting the input image along the decision boundary.
- **HSJA** (Section 4.3.4) is an efficient black-box boundary attack algorithm that utilises zero-order optimisation techniques to achieve an effective attack on the target model by jump searching and progressively adjusting the perturbations. In this attack method, different norms (l_2 and l_∞) are used.

Meanwhile, the parameters used in the attack have been elaborated in Table 4.1. It is worth noting that due to the difference in the complexity of the datasets and the models, the parameters required to achieve the optimal attack state when attacking against different datasets and trained models are varying. In this work, the parameters identified in Table 4.1 are used consistently to enable a better comparison of the attack effects of different attack methods on different models at a macro level.

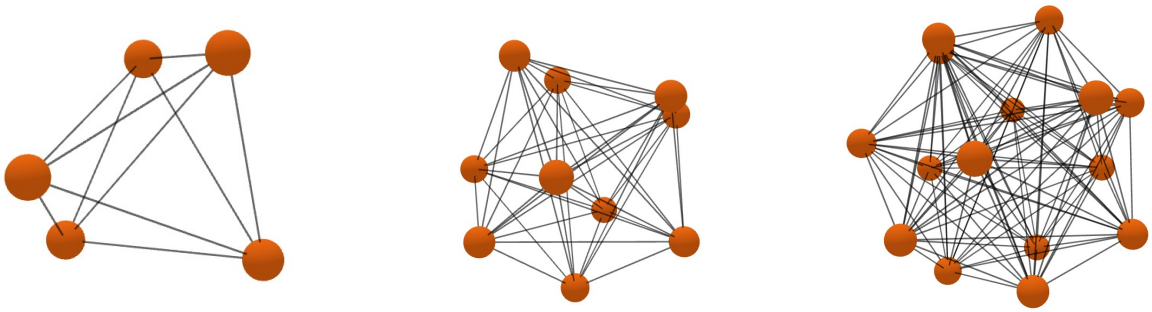
5.1.3 Fedstellar Configuration

In the next paragraphs, the main focus will be on the selection of parameters other than the dataset and the model.

In this work, nodes of 5, 10, and 15 are selected to simulate distributed system environments of different sizes. Fewer nodes (e.g., 5) can reflect the outcome of an attack on a

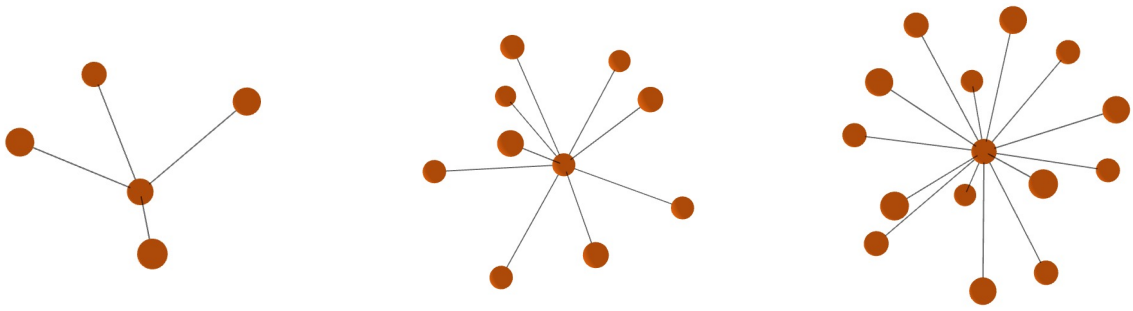
small FL System, while more nodes (e.g., 15) can demonstrate the attack behaviour in a larger system. Also, to test the performance of the attack method in different topologies, the following four topologies are selected:

1. **Full topology:** each node is directly connected to all other nodes, reflecting the optimal communication situation. Figure 5.1 shows the shape of full topology for different number of nodes.
2. **Star topology:** the central node is directly connected to all other nodes and other nodes are not directly connected to each other. Figure 5.2 shows the shape of full topology for different number of nodes.
3. **Random topology:** random connections between nodes reflecting a dynamic network structure closer to practical applications. Figure 5.3 shows the shape of full topology for different number of nodes.
4. **Ring topology:** each node is connected to only two of its neighbouring nodes, forming a ring structure. Figure 5.4 shows the shape of full topology for different number of nodes.



(a) Node = 5 (b) Node = 10 (c) Node = 15

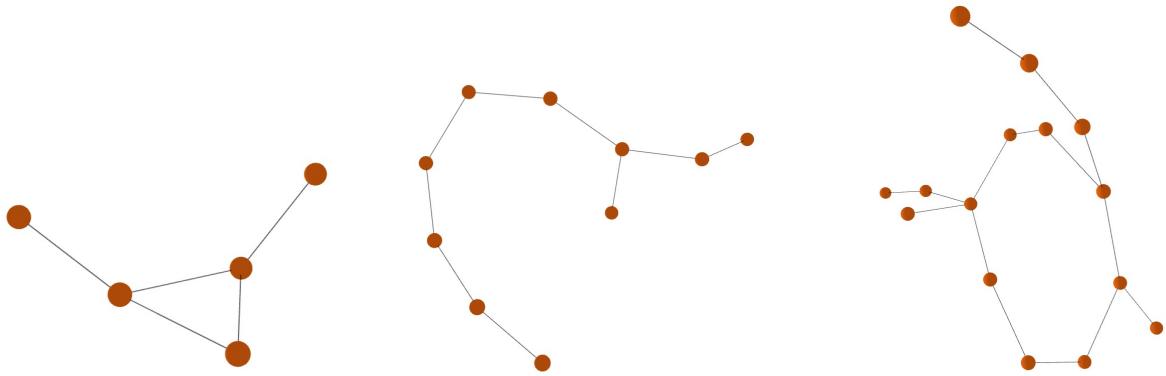
Figure 5.1: Full topologies with different number of nodes.



(a) Node = 5 (b) Node = 10 (c) Node = 15

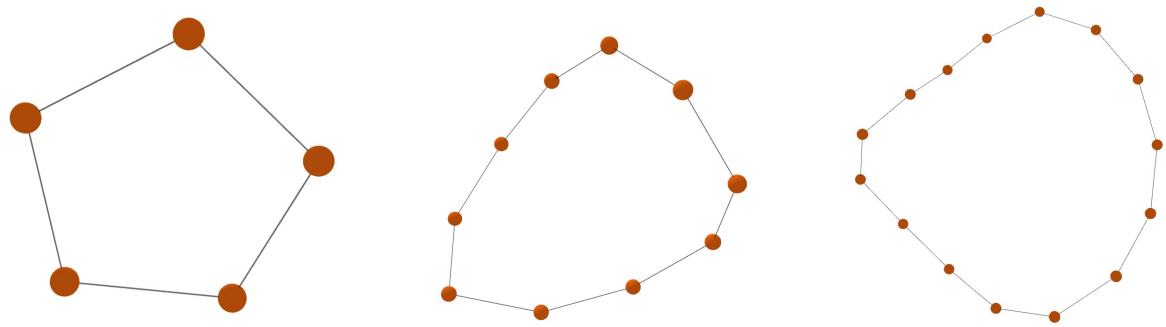
Figure 5.2: Star topologies with different number of nodes.

In this work, two FL architecture, CFL (Centralised Federated Learning) and DFL (Decentralized Federated Learning), are chosen for comparing the effectiveness of attacks



(a) Node = 5 (b) Node = 10 (c) Node = 15

Figure 5.3: Random topologies with different number of nodes. Note that the random topology generates a topology randomly based on the number of nodes, which has uncertainty. Therefore only one of its possibilities is shown in this figure.



(a) Node = 5 (b) Node = 10 (c) Node = 15

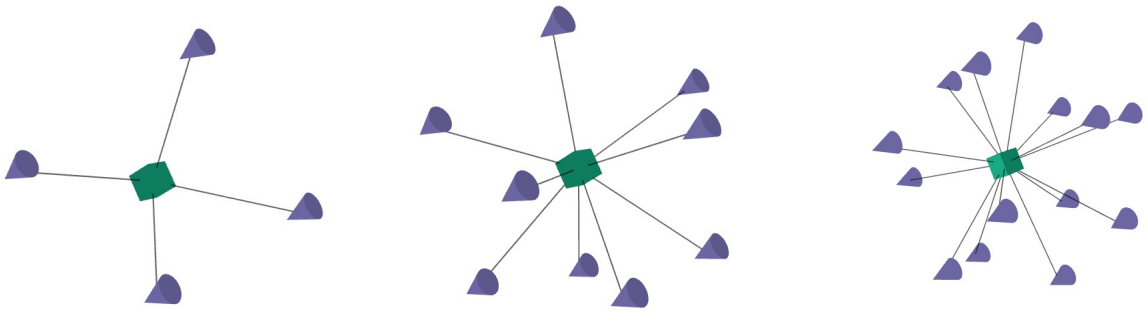
Figure 5.4: Ring topologies with different number of nodes.

under CFL and DFL architectures. In DFL, nodes exchange parameters directly with each other and there is no central server. Whereas in CFL, all nodes perform parameter aggregation through a central server. Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4 show the case of DFL and in Figure 5.5 will show the topology under CFL with different number of nodes.

Finally, to ensure the consistency of the experimental results, the duration of the FL process in all experiments was set to 10 rounds of 3 epochs each.

5.2 Results

In the following, the evaluation of the performance of SimBA, Square attack, Boundary Attack and HSJA is reported. First, a baseline performance reference under benign settings is established. Subsequently, each attack is discussed individually for all datasets. Where applicable, mean and the Standard Error Mean (SEM) is reported.



(a) Node = 5 (b) Node = 10 (c) Node = 15

Figure 5.5: Topologies with different number of nodes in CFL, where the node in green is Server and the nodes in blue are Trainer.

5.2.1 Baseline Performance

Baseline performance refers to the recorded Accuracy, Precision, Recall and F1 Score of the current model when the trained model has not yet been attacked.This serves the purpose of being able to provide a point of reference. By comparing the baseline performance with the performance after the attack, the impact of the attack on the model performance can be clearly assessed. And it provides a consistent reference point, which also ensures that the results are comparable between different experiments.

Table 5.1 shows the baseline Accuracy, Precision, Recall, F1-score performance for MNIST. As can be seen from the tabular data, the SEM values are relatively small, usually between 0.0007 and 0.017, which suggests that the distribution of data points for these averages is more concentrated, which is negligible. The data in the table shows that the performance of the model in CFL and DFL is significantly affected by the topology and the number of nodes. For CFL, the model performs best in all metrics when configured with 5 nodes. Whereas, in DFL, with Full,Ring and Star topologies and a configuration of 5 nodes, the model achieves the highest values in Accuracy, Precision, Recall and F1 Score for this topology, showing its superior performance. And when the number of nodes is the same, the metrics under different topologies do not differ much. In addition, the Random topology presents different results, with the best performance at 15 nodes.

Table 5.2 shows the baseline Accuracy, Precision, Recall, F1-score performance for F-MNIST. Similarly, the SEM values in the tables are small, which is negligible. For CFL, as with the results under the MINIST dataset, the model performs best in all metrics when in the 5-node configuration. Whereas for DFL, Full and Ring topologies, with a configuration of 5 nodes, the model achieves the highest performance in terms of Accuracy, Precision, Recall and F1 Score under this topology. While for Star topology, the model performs better when the number of nodes is 10. When the number of nodes is the same, there is not much difference in the metrics between Full,Ring and Star topologies, Random topology outperforms the other topologies and achieves its best performance at 10 nodes.

Table 5.3 shows the baseline Accuracy, Precision, Recall, F1-score performance for CIFAR-10. The results are similar to those under the MINIST dataset in general. In CFL, the

Table 5.1: Baseline Accuracy, Precision, Recall, F1-score performance for MNIST after 10 rounds in terms of mean and SEM on the local test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.850±0.003**	**0.848±0.002**	**0.85±0.003**	**0.824±0.002**
		10	0.751±0.008	0.710±0.009	0.751±0.008	0.697±0.008
		15	0.779±0.016	0.798±0.017	0.779±0.018	0.748±0.016
DFL	Full	5	**0.972±0.001**	**0.973±0.001**	**0.972±0.001**	**0.968±0.001**
		10	0.969±0.002	0.969±0.001	0.962±0.001	0.964±0.001
		15	0.91±0.016	0.918±0.019	0.855±0.016	0.897±0.019
	Ring	5	**0.971±0.002**	**0.972±0.002**	**0.966±0.001**	**0.967±0.001**
		10	0.964±0.005	0.971±0.007	0.964±0.005	0.966±0.013
		15	0.953±0.011	0.954±0.011	0.953±0.011	0.947±0.013
	Star	5	**0.966±0.002**	**0.961±0.001**	**0.962±0.001**	0.953±0.001
		10	0.96±0.006	0.96±0.005	0.96±0.006	**0.95±0.005**
		15	0.943±0.013	0.945±0.018	0.943±0.012	0.936±0.014
	Random	5	**0.961±0.001**	0.96±0.003	0.95±0.001	0.954±0.002
		10	0.955±0.005	**0.963±0.007**	0.955±0.005	0.948±0.005
		15	0.962±0.01	0.963±0.016	**0.962±0.011**	**0.956±0.015**

best performance is achieved when the number of nodes is 5. And in DFL, despite the different topologies, the model performs better when the number of nodes is 5 than when the number of nodes is more. In addition, when the number of nodes is the same, the metrics do not differ much between the topologies.

Overall, the model performs considerably better under DFL than CFL.In addition, when the number of nodes is the same, there is a slight difference in the performance of the model under different topologies. And when the topologies are the same, the performance of the model does not become better with the increase in the number of nodes.

5.2.2 Attacks Performance

In the following paragraphs, the main focus is on illustrating the changes in terms of their Accuracy, Precision, Recall and F1 Score under different datasets by comparing them with the baseline performance when different attack methods are used, thus providing a better measure of the attack efficiency of different attack methods.

SimBA

When SimBA is used for the MINIST and F-MINIST dataset With MLP model, it is evident that the attack is not effective, i.e., there is not much difference between the metrics after being attacked and the baseline performance. Table 5.4 presents the results under the MINIST dataset after the SimBA attack and Table 5.5 presents the results under the F-MINIST dataset after the SimBA attack. The reason for this is elaborated in detail next in Section 5.3.1.

Table 5.2: Baseline Accuracy, Precision, Recall, F1-score performance for FMNIST after 10 rounds in terms of mean and SEM on the local test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.763±0.006**	**0.724±0.006**	**0.763±0.006**	**0.824±0.008**
		10	0.724±0.011	0.694±0.013	0.724±0.017	0.697±0.014
		15	0.699±0.016	0.672±0.015	0.699±0.015	0.748±0.016
DFL	Full	5	**0.891±0.001**	**0.89±0.002**	**0.891±0.002**	**0.968±0.002**
		10	0.864±0.008	0.854±0.01	0.864±0.007	0.964±0.005
		15	0.849±0.016	0.822±0.014	0.849±0.016	0.897±0.015
	Ring	5	**0.886±0.001**	0.872±0.001	**0.886±0.001**	**0.967±0.001**
		10	0.867±0.003	**0.882±0.005**	0.867±0.003	0.966±0.003
		15	0.836±0.014	0.837±0.012	0.833±0.014	0.947±0.013
	Star	5	0.861±0.001	**0.858±0.001**	0.861±0.001	0.953±0.001
		10	**0.867±0.002**	0.853±0.001	**0.862±0.002**	**0.954±0.002**
		15	0.832±0.017	0.827±0.019	0.831±0.013	0.936±0.012
	Random	5	**0.91±0.002**	0.894±0.001	**0.907±0.001**	0.954±0.001
		10	0.902±0.006	**0.895±0.009**	0.897±0.005	0.948±0.005
		15	0.873±0.019	0.868±0.017	0.875±0.018	**0.956±0.019**

Table 5.3: Baseline Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after 10 rounds in terms of mean and SEM on the local test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.6±0.002**	**0.594±0.002**	**0.6±0.01**	**0.597±0.001**
		10	0.573±0.004	0.576±0.003	0.575±0.005	0.575±0.004
		15	0.569±0.01	0.554±0.02	0.565±0.017	0.559±0.013
DFL	Full	5	**0.768±0.01**	**0.767±0.002**	**0.768±0.001**	0.733±0.001
		10	0.739±0.002	0.727±0.002	0.737±0.001	**0.764±0.001**
		15	0.687±0.01	0.677±0.016	0.685±0.019	0.681±0.013
	Ring	5	**0.782±0.005**	**0.791±0.006**	**0.784±0.005**	**0.787±0.006**
		10	0.773±0.007	0.782±0.005	0.774±0.017	0.778±0.07
		15	0.756±0.014	0.761±0.018	0.757±0.016	0.759±0.015
	Star	5	**0.755±0.001**	**0.758±0.003**	**0.755±0.003**	**0.756±0.002**
		10	0.732±0.006	0.741±0.007	0.738±0.006	0.739±0.005
		15	0.671±0.018	0.687±0.012	0.672±0.017	0.679±0.016
	Random	5	0.771±0.003	0.769±0.007	**0.77±0.002**	0.769±0.002
		10	**0.773±0.008**	**0.778±0.007**	0.771±0.006	**0.774±0.008**
		15	0.762±0.018	0.759±0.019	0.762±0.017	0.76±0.018

Table 5.4: Accuracy, Precision, Recall, F1-score performance for MINIST after SimBA in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	0.846±0.002	0.835±0.002	0.836±0.001	0.811±0.001
		10	**0.759±0.008**	**0.714±0.02**	**0.759±0.005**	**0.721±0.006**
		15	0.786±0.12	0.834±0.1	0.786±0.08	0.785±0.112
DFL	Full	5	0.92±0.001	0.928±0.002	0.92±0.001	0.921±0.001
		10	0.948±0.002	0.949±0.004	0.948±0.002	0.948±0.009
		15	**0.873±0.011**	**0.891±0.013**	**0.873±0.01**	**0.869±0.015**
	Ring	5	**0.919±0.003**	**0.925±0.001**	**0.919±0.002**	**0.919±0.002**
		10	0.944±0.002	0.947±0.002	0.944±0.002	0.945±0.002
		15	0.93±0.014	0.935±0.021	0.93±0.017	0.931±0.014
	Star	5	**0.916±0.001**	**0.917±0.001**	**0.916±0.02**	**0.915±0.002**
		10	0.933±0.005	0.934±0.002	0.933±0.003	0.933±0.003
		15	0.922±0.009	0.923±0.01	0.922±0.013	0.926± 0.012
	Random	5	**0.944±0.001**	0.956±0.002	**0.934±0.003**	**0.944±0.002**
		10	0.948±0.003	0.958±0.001	0.938±0.002	0.948±0.003
		15	0.944±0.009	**0.954±0.013**	0.934±0.009	0.945±0.011

The Accuracy, Precision, Recall and F1 Score of the CIFAR-10 dataset after the attack with different nodes as well as topologies are recorded in Table 5.6. By comparing with baseline, the decrease of each metrics is significantly higher with CFL federation architecture than with DFL. This indicates that SimBA is better attacked in the CFL scenario. In DFL, although it is shown in Table 5.3 that its model performance is better when the number of nodes is 5 for different topologies. However, in DFL, when the Star and Random topology apply, their attack is better than others when the number of nodes is 10. In addition, combining the different topologies, the Star topology has a slightly worse attack performance than the other three, and the Random topology shows its better attack efficiency.

Square Attack

The analysis of Square Attack's attack on MINIST (Table 5.7) shows that the attack on CFL is slightly better than the attack on DFL, but the difference is not significant in general. In addition, the attack performance under the same topology shows a tendency of decreasing with the increase of the number of nodes, except for the Random topology, where the attack performance basically remains in a stable state. In addition, the overall attack performance under the same nodes does not differ much despite the use of different topologies.

As for Square Attack under the F-MINIST dataset (Table 5.8), the attack performance under CFL and DFL is basically the same, and its attack effect is accompanied by slight

Table 5.5: Accuracy, Precision, Recall, F1-score performance for F-MINIST after SimBA in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	0.752±0.002	0.725±0.001	0.752±0.001	**0.741±0.002**
		10	**0.732±0.002**	**0.664±0.005**	**0.712±0.003**	0.758±0.004
		15	0.743±0.012	0.689±0.013	0.733±0.012	0.776±0.009
DFL	Full	5	0.875±0.002	0.8654±0.003	0.895±0.002	0.904±0.002
		10	0.891±0.005	0.882±0.008	0.881±0.009	0.922±0.007
		15	**0.864±0.013**	**0.811±0.019**	**0.874±0.011**	**0.878±0.015**
	Ring	5	**0.878±0.001**	**0.886±0.002**	**0.877±0.001**	**0.907±0.002**
		10	0.887±0.002	0.888±0.005	0.877±0.001	0.928±0.002
		15	0.89±0.016	0.889±0.015	0.9096±0.018	0.949±0.016
	Star	5	**0.855±0.002**	**0.805±0.003**	**0.835±0.002**	**0.905±0.002**
		10	0.831±0.009	0.832±0.004	0.831±0.009	0.932±0.013
		15	0.846±0.014	0.865±0.013	0.866±0.016	0.966±0.018
	Random	5	0.905±0.001	0.905±0.002	**0.905±0.001**	0.905±0.003
		10	0.911±0.007	0.908±0.004	0.911±0.009	0.909±0.008
		15	**0.904±0.016**	**0.905±0.017**	0.905±0.016	**0.904±0.018**

Table 5.6: Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after SimBA in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	0.309±0.002	0.433±0.002	0.309±0.001	**0.219±0.001**
		10	**0.288±0.002**	**0.401±0.004**	**0.289±0.002**	0.336±0.003
		15	0.33±0.017	0.452±0.014	0.33±0.019	0.382±0.017
DFL	Full	5	**0.679±0.001**	0.696±0.004	**0.679±0.002**	**0.678± 0.001**
		10	0.71±0.002	0.734±0.001	0.714±0.002	0.724±0.006
		15	0.686±0.015	**0.684±0.012**	0.686±0.017	0.685±0.02
	Ring	5	**0.67±0.006**	**0.681±0.004**	0.679±0.007	**0.68±0.007**
		10	0.679±0.012	0.684±0.009	**0.679±0.005**	0.682±0.006
		15	0.708±0.015	0.713±0.014	0.708±0.003	0.711±0.02
	Star	5	0.702±0.001	0.713±0.002	0.702±0.001	0.708±0.001
		10	**0.691±0.002**	**0.7± 0.002**	**0.691±0.001**	**0.696±0.002**
		15	0.731±0.008	0.751±0.014	0.731±0.008	0.741±0.011
	Random	5	0.679±0.002	0.696±0.001	0.679±0.005	0.687±0.003
		10	**0.64±0.002**	**0.657±0.002**	**0.64±0.002**	**0.649±0.001**
		15	0.667±0.017	0.679±0.021	0.668±0.013	0.673±0.019

Table 5.7: Accuracy, Precision, Recall, F1-score performance for MINIST after Square Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.096±0.003**	**0.096±0.002**	**0.096±0.003**	**0.096±0.001**
		10	0.1± 0.001	0.108±0.004	0.1±0.001	0.104±0.001
		15	0.112±0.016	0.12±0.011	0.116±0.012	0.118±0.008
DFL	Full	5	**0.093±0.001**	0.12±0.002	**0.094±0.001**	0.105±0.002
		10	0.112±0.003	**0.093±0.006**	0.112±0.005	**0.102±0.006**
		15	0.101±0.011	0.11±0.014	0.101±0.018	0.105±0.012
	Ring	5	**0.1±0.002**	0.122±0.001	**0.1±0.002**	0.11±0.005
		10	0.105±0.004	**0.097±0.001**	0.105±0.003	**0.101±0.003**
		15	0.125±0.014	0.125±0.001	0.125±0.001	0.125±0.001
	Star	5	**0.12±0.001**	0.12±0.011	0.12±0.007	0.119±0.006
		10	0.114±0.005	**0.114±0.003**	**0.114±0.005**	**0.114±0.005**
		15	0.123±0.012	0.125±0.011	0.123±0.013	0.124±0.009
	Random	5	0.123±0.001	0.124±0.002	0.123±0.001	0.123±0.002
		10	0.123±0.003	0.118±0.002	0.123±0.003	0.12±0.003
		15	**0.114±0.009**	**0.104±0.003**	**0.114±0.011**	**0.109±0.012**

ups and downs as the number of nodes increases. Similarly, there is no massive difference in the attack performance for the same number of nodes under different topologies.

Table 5.9 shows the Accuracy, Precision, Recall, and F1 Score of the CIFAR-10 dataset after Square Attack.Similar to the Square Attack attack on MINIST, the CFL presents a better attack, and the success rate of its attack is significantly lower than that on the remaining two datasets. In the same topology, the attack performance decreases slightly with the increase of the number of nodes. Apart from the Random topology, the attack performance remains almost the same when the same nodes are used in different topologies.

Boundary Attack

When using Boundary Attack to attack the MINIST dataset (Table 5.10), it can be concluded that the performance of the attack under DFL is significantly better than that of CFL, and this difference grows as the number of nodes increases, and the difference between the number of nodes is 15 and the number of nodes is 10 is much larger than the difference between the number of nodes and the number of nodes is 10 and the number of nodes is 5. In DFL, however, in general there is little difference between the topologies, and they also all show a tendency of decreasing their Accuracy, Precision, Recall and F1 Score as the number of nodes increases. It is worth mentioning that the results of Boundary Attack on F-MINIST (Table 5.11) and CIFAR (5.12) are the same as the above discussed conclusion. This implies that Boundary Attack is a stable attack method, i.e., the attack

Table 5.8: Accuracy, Precision, Recall, F1-score performance for F-MINIST after Square Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.092±0.002**	0.121±0.004	**0.092±0.005**	**0.105±0.004**
		10	0.112±0.005	**0.1±0.006**	0.112±0.009	0.106±0.004
		15	0.123±0.008	0.121±0.013	0.123±0.013	0.122±0.013
DFL	Full	5	0.099±0.006	**0.097±0.013**	0.099±0.001	0.098±0.013
		10	**0.091±0.006**	0.091±0.003	**0.091±0.013**	**0.091±0.01**
		15	0.093±0.011	0.103±0.012	0.093±0.003	0.098±0.008
	Ring	5	0.109±0.001	**0.1±0.003**	**0.109±0.001**	**0.104±0.001**
		10	0.11±0.003	0.109±0.004	0.11±0.003	0.11±0.005
		15	**0.108±0.004**	0.115±0.003	0.108±0.003	0.111±0.003
	Star	5	**0.097±0.007**	0.111±0.001	**0.097±0.005**	0.104±0.009
		10	0.104±0.005	0.109±0.003	0.104±0.006	0.106±0.004
		15	0.101±0.004	**0.103±0.007**	0.101±0.009	**0.102±0.006**
	Random	5	**0.092±0.004**	**0.107±0.004**	**0.092±0.004**	0.114±0.013
		10	0.123±0.006	0.109±0.005	0.123±0.005	**0.104±0.006**
		15	0.114±0.007	0.116±0.007	0.114±0.009	0.107±0.008

Table 5.9: Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after Square Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.1912±0.012**	0.232±0.01	**0.192±0.002**	0.21±0.008
		10	0.195±0.02	0.234±0.008	0.195±0.012	0.213±0.002
		15	0.198±0.007	**0.197±0.008**	0.198±0.013	**0.198±0.011**
DFL	Full	5	**0.192±0.01**	**0.194±0.007**	**0.192±0.009**	**0.193±0.001**
		10	0.214±0.012	0.219±0.007	0.214±0.001	0.217±0.008
		15	0.222±0.002	0.236±0.005	0.222±0.008	0.231±0.013
	Ring	5	**0.192±0.002**	0.222±0.008	**0.192±0.006**	**0.206±0.002**
		10	0.214±0.006	0.222±0.003	0.214±0.011	0.218±0.008
		15	0.205±0.001	**0.212±0.002**	0.205±0.013	0.209±0.014
	Star	5	0.195±0.002	0.205±0.005	0.195±0.002	**0.2± 0.003**
		10	**0.191±0.008**	0.214±0.011	**0.191±0.011**	0.202±0.006
		15	0.215±0.012	**0.198±0.017**	0.215±0.011	0.206±0.013
	Random	5	**0.21±0.001**	**0.22±0.001**	**0.21±0.002**	**0.215±0.001**
		10	0.233±0.001	0.229±0.003	0.233±0.002	0.231±0.003
		15	0.211±0.008	0.227±0.014	0.211±0.012	0.219±0.011

Table 5.10: Accuracy, Precision, Recall, F1-score performance for Minist after Boundary Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.047±0.001**	0.184±0.001	**0.047±0.001**	0.066±0.006
		10	0.047±0.001	0.17±0.003	0.047±0.004	**0.049±0.007**
		15	0.076±0.003	**0.03±0.006**	0.076±0.003	0.086±0.005
DFL	Full	5	**0.014±0.002**	0.054±0.001	**0.014±0.001**	**0.019±0.001**
		10	0.015±0.002	**0.035±0.003**	0.016±0.004	0.016±0.006
		15	0.038±0.008	0.119±0.008	0.038±0.008	0.041±0.006
	Ring	5	**0.013±0.002**	**0.046±0.003**	**0.013±0.002**	**0.016±0.003**
		10	0.014±0.001	0.059±0.002	0.014±0.001	0.01±0.002
		15	0.021±0.008	0.065±0.009	0.021±0.009	0.021±0.007
	Star	5	0.019±0.004	**0.013±0.002**	0.019±0.002	0.025±0.003
		10	**0.019±0.003**	0.057±0.002	**0.019± 0.005**	**0.019±0.005**
		15	0.022±0.004	0.113±0.007	0.022±0.006	0.026±0.007
	Random	5	0.019±0.003	0.097±0.005	0.019±0.007	0.025±0.002
		10	0.018±0.005	**0.056±0.004**	0.018±0.005	**0.019±0.006**
		15	**0.017±0.006**	0.062±0.006	**0.017±0.008**	0.02±0.006

performance and success rate do not fluctuate much when the attack is performed on different datasets.

HSJA

In this work, two norms, l_2 and l_∞, are used to examine the effectiveness of HSJA with different datasets. When the l_2 norm is used, the recording of the attack on MINIST, F-MINIST and CIFAR-10 is shown in Table 5.13, Table 5.14 and Table 5.15. When the l_∞ norm is used, the record for the attack on MINIST, F-MINIST and CIFAR-10 is in Table 5.16, Table 5.17 and Table 5.18. By comparison, it can be seen that no matter which norm is used, the attack effect is better under DFL than CFL. In CFL, the performance of the attack decreases with the number of nodes increase, but the decrease is much lower than that of the Boundary Attack. While when using the DFL federation structure, it can be found that when attacking with the l_2 norm, the Ring topology is more effective better than the other attacks, while the Full topology performs better when attacking using l_∞. However, in general, there is little difference between each topology. Meanwhile, similar to other attack methods, in HSJA, when the topology is certain, the performance of the attack is not positively correlated with the number of nodes. In addition, by comparing HSJA using both l_2 and l_∞ norms, it can be seen that there is a slight difference in the effectiveness of their attacks, for example, in different datasets, HSJA using l_∞ norm performs better in Full topology, while HSJA using l_2 norm shows better performance in the remaining three topologies.

Table 5.11: Accuracy, Precision, Recall, F1-score performance for F-Minist after Boundary Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.072±0.001**	**0.631±0.011**	**0.072±0.001**	**0.124±0.002**
		10	0.098±0.008	0.707±0.012	0.098±0.002	0.172±0.005
		15	0.111±0.011	0.776±0.011	0.115±0.018	0.21±0.008
DFL	Full	5	**0.051±0.001**	**0.114±0.017**	**0.051±0.004**	**0.045±0.001**
		10	0.052±0.019	0.139±0.016	0.052±0.007	0.076±0.005
		15	0.073±0.005	0.152±0.01	0.073±0.006	0.099±0.007
	Ring	5	**0.047±0.007**	0.102±0.008	**0.047±0.003**	**0.065±0.003**
		10	0.049±0.001	**0.099±0.018**	0.049±0.002	0.066±0.002
		15	0.061±0.007	0.104±0.013	0.061±0.005	0.077± 0.007
	Star	5	**0.057±0.014**	**0.133±0.017**	**0.048±0.008**	**0.07±0.001**
		10	0.061±0.002	0.142±0.001	0.06±0.009	0.085±0.006
		15	0.073±0.005	0.173±0.007	0.073±0.008	0.103±0.012
	Random	5	0.059±0.004	0.142±0.009	0.059±0.006	0.083±0.001
		10	**0.057±0.007**	**0.139±0.013**	**0.057±0.006**	**0.081±0.003**
		15	0.061±0.006	0.145±0.01	0.061±0.005	0.086±0.006

Table 5.12: Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after Boundary Attack in terms of mean and SEM on the adversarial test dataset.

Architecture	Topology	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.184±0.002**	0.261±0.032	**0.184±0.012**	**0.074±0.004**
		10	0.235±0.016	**0.258±0.002**	0.235±0.013	0.076±0.003
		15	0.256±0.004	0.372±0.009	0.256±0.009	0.167±0.004
DFL	Full	5	**0.173±0.003**	0.21±0.001	**0.173±0.014**	**0.12±0.002**
		10	0.202±0.006	**0.21±0.002**	0.202±0.013	0.206±0.009
		15	0.359±0.003	0.384±0.007	0.359±0.015	0.371±0.013
	Ring	5	**0.167±0.01**	**0.228±0.001**	**0.167±0.003**	**0.193±0.004**
		10	0.203±0.018	0.275±0.003	0.184±0.011	0.221±0.007
		15	0.251±0.008	0.388±0.007	0.252±0.01	0.305±0.008
	Star	5	**0.186±0.012**	**0.242±0.007**	**0.186±0.011**	**0.21±0.004**
		10	0.212±0.003	0.251±0.001	0.212±0.011	0.23±0.008
		15	0.384±0.008	0.361±0.013	0.384±0.016	0.372±0.007
	Random	5	0.201±0.017	0.252±0.012	0.201±0.001	0.2235±0.002
		10	**0.197±0.007**	**0.244±0.008**	**0.197±0.002**	**0.218±0.01**
		15	0.331±0.013	0.322±0.012	0.331±0.005	0.326±0.016

Table 5.13: Accuracy, Precision, Recall, F1-score performance for MINIST after HSJA with l_2 norm in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.049±0.002**	0.182±0.005	**0.049±0.005**	**0.067±0.006**
		10	0.065±0.002	**0.1673±0.009**	0.065±0.002	0.07±0.005
		15	0.107±0.008	0.293±0.01	0.107±0.006	0.133±0.007
DFL	Full	5	**0.015±0.001**	0.04±0.007	**0.015±0.001**	0.018±0.002
		10	0.017±0.003	**0.03±0.007**	0.017±0.003	**0.016±0.006**
		15	0.037±0.007	0.103±0.011	0.0373±0.008	0.04±0.009
	Ring	5	0.015±0.002	**0.039± 0.007**	0.015±0.004	0.017±0.004
		10	**0.014±0.001**	0.044±0.009	**0.014±0.006**	**0.015±0.007**
		15	0.022±0.006	0.06±0.011	0.022±0.008	0.022±0.006
	Star	5	**0.011±0.003**	**0.027±0.006**	**0.011±0.003**	0.029±0.003
		10	0.023±0.006	0.048±0.006	0.022±0.006	**0.022±0.008**
		15	0.028±0.008	0.094±0.008	0.028±0.004	0.0292±0.009
	Random	5	0.022±0.004	0.085±0.003	0.022±0.002	0.029±0.004
		10	0.021±0.004	**0.04±0.001**	0.021±0.004	0.021±0.003
		15	**0.019±0.007**	0.049±0.006	**0.019±0.006**	**0.021±0.005**

Table 5.14: Accuracy, Precision, Recall, F1-score performance for F-MINIST after HSJA with l_2 norm in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.067±0.001**	0.2755±0.005	**0.067±0.003**	**0.081±0.002**
		10	0.071±0.003	**0.269±0.004**	0.071±0.004	0.113±0.007
		15	0.143±0.005	0.367±0.005	0.143±0.008	0.206±0.007
DFL	Full	5	**0.061±0.002**	0.113±0.001	**0.061±0.002**	**0.053±0.003**
		10	0.069±0.004	**0.109±0.002**	0.069±0.003	0.085±0.004
		15	0.091±0.007	0.234±0.003	0.091±0.008	0.131±0.006
	Ring	5	**0.063±0.001**	**0.099±0.004**	**0.063±0.006**	**0.077±0.005**
		10	0.073±0.003	0.113±0.003	0.073±0.009	0.089±0.007
		15	0.137±0.005	0.263±0.008	0.137±0.006	0.18±0.009
	Star	5	**0.084±0.003**	**0.125±0.003**	**0.084±0.004**	**0.101±0.005**
		10	0.093±0.003	0.154±0.004	0.093±0.003	0.116±0.005
		15	0.143±0.006	0.231±0.008	0.143± 0.006	0.177±0.007
	Random	5	**0.06±0.004**	**0.108±0.006**	0.06±0.004	**0.077±0.004**
		10	0.065±0.003	0.129±0.005	**0.065±0.003**	0.086±0.006
		15	0.134±0.006	0.369±0.007	0.134±0.005	0.196±0.008

Table 5.15: Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after HSJA with l_2 norm in terms of mean and SEM on the adversarial test dataset.

Architecture	Topology	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.174±0.003**	**0.278±0.005**	**0.174±0.002**	**0.117±0.001**
		10	0.201±0.005	0.389±0.004	0.2012±0.006	0.265±0.005
		15	0.358±0.006	0.398±0.008	0.3582±0.006	0.377±0.003
DFL	Full	5	**0.2± 0.01**	0.206±0.01	**0.1997±0.0091**	**0.155±0.007**
		10	0.238±0.006	**0.199±0.007**	0.238±0.006	0.217±0.005
		15	0.342±0.008	0.31±0.009	0.3417±0.004	0.325±0.011
	Ring	5	0.195±0.01	**0.204±0.006**	0.195±0.01	**0.199±0.009**
		10	**0.194±0.005**	0.232±0.008	**0.194±0.011**	0.211±0.004
		15	0.247±0.004	0.308±0.011	0.2467±0.006	0.274±0.009
	Star	5	**0.232±0.003**	**0.263±0.006**	**0.232±0.009**	**0.247±0.004**
		10	0.259±0.011	0.284±0.007	0.259±0.006	0.271±0.003
		15	0.375±0.006	0.389±0.004	0.375±0.009	0.382±0.008
	Random	5	**0.255±0.005**	0.315±0.006	**0.255±0.008**	0.282±0.005
		10	0.258±0.006	**0.309±0.006**	0.258±0.01	**0.281±0.008**
		15	0.371±0.011	0.382±0.01	0.367±0.009	0.377±0.009

Table 5.16: Accuracy, Precision, Recall, F1-score performance for MINIST after HSJA with l_∞ norm in terms of mean and SEM on the adversarial test dataset.

Architecture	Topol.	Nodes	Accuracy	Precision	Recall	F1 Score
CFL	Star	5	**0.057±0.001**	0.178±0.003	**0.057±0.001**	0.076±0.002
		10	0.068±0.002	**0.174±0.005**	0.068±0.003	**0.075±0.004**
		15	0.116±0.006	0.292±0.009	0.116±0.008	0.147±0.007
DFL	Full	5	**0.015±0.003**	0.034±0.002	**0.015±0.002**	0.017±0.004
		10	0.016±0.002	**0.028±0.005**	0.016±0.005	**0.015±0.006**
		15	0.038±0.007	0.091±0.008	0.038±0.007	0.042±0.005
	Ring	5	0.015±0.001	**0.037±0.001**	0.015±0.002	0.018±0.001
		10	**0.014±0.003**	0.04±0.006	**0.014±0.005**	**0.015±0.007**
		15	0.023±0.008	0.048±0.009	0.023±0.006	0.023±0.005
	Star	5	0.024±0.002	0.094±0.003	**0.024±0.002**	0.032±0.002
		10	**0.023±0.003**	**0.041±0.005**	0.025±0.003	**0.022±0.003**
		15	0.028±0.004	0.09±0.004	0.029±0.004	0.03±0.004
	Random	5	0.022±0.003	0.073±0.004	0.022±0.002	0.029±0.003
		10	**0.02±0.005**	**0.033±0.003**	**0.02±0.004**	**0.021±0.004**
		15	0.021±0.006	0.044±0.004	0.021±0.004	0.023±0.006

Table 5.17: Accuracy, Precision, Recall, F1-score performance for F-MINIST after HSJA with l_∞ norm in terms of mean and SEM on the adversarial test dataset.

Architecture	**Topol.**	**Nodes**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
CFL	Star	5	**0.064±0.001**	**0.264±0.002**	**0.064±0.001**	**0.072±0.004**
		10	0.071±0.005	0.298±0.004	0.071±0.005	0.103±0.007
		15	0.138±0.007	0.37±0.008	0.138±0.006	0.176±0.008
DFL	Full	5	**0.06±0.001**	0.111±0.003	**0.06±0.001**	**0.05±0.001**
		10	0.062±0.002	**0.11±0.004**	0.123±0.003	0.162±0.003
		15	0.131±0.007	0.311±0.004	0.131±0.008	0.165±0.009
	Ring	5	**0.06±0.001**	**0.111±0.002**	**0.06±0.003**	**0.07±0.002**
		10	0.061±0.003	0.112±0.007	0.061±0.003	0.071±0.004
		15	0.162±0.006	0.137±0.009	0.162±0.005	0.156±0.005
	Star	5	0.06±0.001	0.111±0.002	0.06± 0.001	0.07±0.001
		10	**0.059±0.005**	**0.097±0.008**	**0.059±0.005**	**0.067±0.005**
		15	0.07±0.008	0.173±0.011	0.07±0.008	0.089±0.003
	Random	5	0.06±0.002	0.111±0.003	0.06±0.004	0.07±0.003
		10	**0.059±0.007**	**0.082±0.007**	**0.059±0.005**	**0.064±0.006**
		15	0.073±0.011	0.128±0.009	0.073±0.01	0.085±0.008

Table 5.18: Accuracy, Precision, Recall, F1-score performance for CIFAR-10 after HSJA with l_∞ norm in terms of mean and SEM on the adversarial test dataset.

Federation Architecture	**Topology**	**Nodes**	**Accuracy**	**Precision**	**Recall**	**F1 Sc**
CFL	Star	5	**0.169±0.002**	**0.26±0.003**	**0.169±0.002**	**0.112**
		10	0.219±0.004	0.299±0.002	0.219±0.004	0.253
		15	0.322±0.009	0.3512 ± 0.008	0.322±0.01	0.336
DFL	Full	5	**0.192±0.001**	**0.203±0.001**	**0.192±0.002**	**0.146**
		10	0.251±0.001	0.279± 0.002	0.251±0.001	0.264
		15	0.343±0.011	0.367±0.006	0.343±0.009	0.354
	Ring	5	**0.201±0.001**	**0.204±0.002**	**0.201±0.001**	**0.203**
		10	0.274±0.002	0.284±0.003	0.274±0.008	0.279
		15	0.371±0.007	0.369±0.008	0.371±0.007	0.37±
	Star	5	**0.252±0.002**	**0.283±0.003**	**0.252±0.002**	**0.267**
		10	0.298±0.002	0.31±0.004	0.299±0.002	0.304
		15	0.369±0.016	0.378±0.011	0.369±0.017	0.374
	Random	5	0.192±0.003	0.203±0.004	**0.182±0.003**	0.197
		10	**0.19±0.004**	**0.189±0.006**	0.189±0.004	**0.189**
		15	0.299±0.011	0.34±0.006	0.299±0.009	0.319

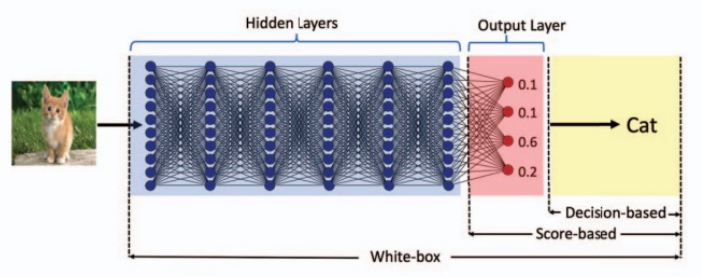


Figure 5.6: An illustration of accessible components for different attack methods. [26]

5.3 Discussion

5.3.1 SimBA witn MINIST and F-MINIST

When attacking the MINIST and F-MINIST datasets with SimBA, the Accuracy, Precision, Recall, and F1 Score metrics decrease by a minimal amount, in other words, it is difficult to trick the model into making incorrect predictions with the perturbations generated by SimBA. Possible reasons for this situation are:

1. In this work, the attack against pixels in SimBA is used, which also dictates that the range of each perturbation is particularly small, which in turn has a limited ability to affect the model's predictions, and therefore it is difficult to have a large impact on the model's predictions in a limited number of iterations.

2. Also when selecting the pixels to be perturbed, there are many ways to pick them. In this work, a random order is used, i.e. the order of the pixels in the original sample is disturbed and the attack is performed according to this random order, which makes the uncertainty of the attack higher. For example, an attack on a pixel in an iteration of SimBA decreases the probability of the model's prediction being successful, and in the next iteration, instead of attacking along the vertical, horizontal, or diagonal direction of the pixel, the attack is relocated to a new and random pixel to perturb. In this way, the latest attack position may not reduce the probability of correct model prediction, which on one hand makes the attack less efficient and on the other hand affects the final attack.

3. SimBA is a score-based attack method. Score-based attacks use the model's output scores (e.g., probability distributions or logits) to guide the generation of adversarial samples, whereas decision-based attacks use only the model's final decisions (i.e., predicted abels) to generate adversarial samples. Figure 5.6 illustrates of accessible components of the target model for each, from which it can be seen that score-based attack assumes access to the output layer and a decision-based attack assumes access to the predicted label alone. As can be seen from Table 5.1 and Table 5.2, when trained for 10 rounds, the accuracy is as high as 97% for the MINIST dataset using MLP, and almost 90% for F-MINIST using CNN, which indicate that the probability of successful model prediction is extremely high. Meanwhile, to further figure out the probability of the model predicting different labels, the `y_pred` after using

the `softmax` method for `logits` is recorded, and the following is an example of one of them: `y_pred = [4.9019e-03, 9.5949e-05, 6.0425e-03, 5.5782e-03, 1.2769e-04, 1.8458e-03, 8.0444e-06, 9.6489e-01, 1.3356e-03, 1.5170e-02]`. In this case, the probability of success of the model in predicting correct labels is 96.489%, while the maximum of the probability of predicting incorrect labels is only 1.517%. Therefore, trying to make the probability of predicting wrong labels larger than the probability of correct labels through pixel perturbation is not a simple task. In addition, in order to record whether SimBA is really successful in attacking, i.e., whether it has made the model reduce the probability of predicting correct labels, by recording the probability `y_init` at the beginning of the attack and the probability `last_prob` after the completion of the attack, and by calculating the difference between the them, the average probability of the decrease is 0.00581, which also means that SimBA is successful in attacking, but this level of attack can hardly make the model make wrong classification.

5.3.2 Attack Methods

In a comprehensive adversarial attack experiment on MNIST, F-MNIST and CIFAR-10 datasets, several different methods, namely SimBA, Square Attack, Boundary Attack and HSJA (including l_2 and l_∞ norms), are used in this work and different FL parameters are set for testing. The following are the findings by analysing the experimental data:

SimBA

SimBA method performs poorly on the MNIST and F-MNIST datasets. This suggests that the method may lack effectiveness when dealing with these simpler image datasets with high contrast. On the contrary, SimBA performs moderately well on the CIFAR-10 dataset, which may be due to the higher complexity of the CIFAR-10 images, and SimBA is able to find more adversarial samples on the more complex images.

Square Attack

Square Attack's performance on all three datasets is relatively stable, showing its strong adaptability and consistency. It implies that Square Attack is able to execute its attacks with roughly the same success rate on both the relatively simple MNIST and F-MNIST datasets, as well as the more complex CIFAR-10 dataset. This stability indicates that Square Attack is able to keep its attack effectiveness against datasets of different types and complexity without significant fluctuations due to changes in the dataset [40]. Therefore, Square Attack is considered to be a very reliable adversarial attack method, especially suitable for scenarios that require adversarial testing on multiple datasets. Whether in academic research or in real-world applications, Square Attack provides consistent and credible attack effects, enabling researchers and practitioners to better evaluate and improve the robustness of ML models.

Boundary Attack and HSJA

Among all the attack methods tested, Boundary Attack and HSJA perform outstandingly well, especially HSJA, which has a higher success rate. This suggests that HSJA is more effective in generating adversarial samples, benefiting from its more efficient search strategy (`_binary_search` method) and optimisation mechanism (`_compute_update` method). Specifically, HSJA is able to find better adversarial samples in a shorter period of time, which also can help improve the success rate of the attack. Boundary Attack, despite its excellent performance, is slightly inferior to HSJA.

A noteworthy phenomenon is that there is only little difference in the metrics of HSJA when attacking using different norms (l_2 and l_∞). This indicates that HSJA maintains a consistently high level of attack effectiveness whether the l_2 norm or the l_∞ norm is used. The insensitivity of the norms shows the robustness and versatility of HSJA, which means that it is able to adapt to different constraints and still maintain excellent performance. Regardless of the paradigm constraints, HSJA is able to generate adversarial samples efficiently, thus demonstrating powerful attack capabilities in different application scenarios. This feature is essential for adversarial testing in real-world applications, as it ensures HSJA operates stably under various conditions, thus serving as a valuable tool for assessing and enhancing the security of ML models.

5.3.3 Fedstellar Configuration

In the Fedstellar Configuration, aside from the fixed training rounds and epochs, this work also sets different datasets and models, number of nodes, and topologies. The following paragraph details how each parameter impacts black-box adversarial attacks.

Datasets

A comparative analysis of the accuracy of different attack methods across various datasets reveals that the effectiveness of black-box attacks varies significantly across datasets. With the exception of SimBA (see Section 1), Square Attack, Boundary Attack and HSJA demonstrate the most effective attack performance on MNIST, followed by F-MNIST, and the least effective on CIFAR-10.

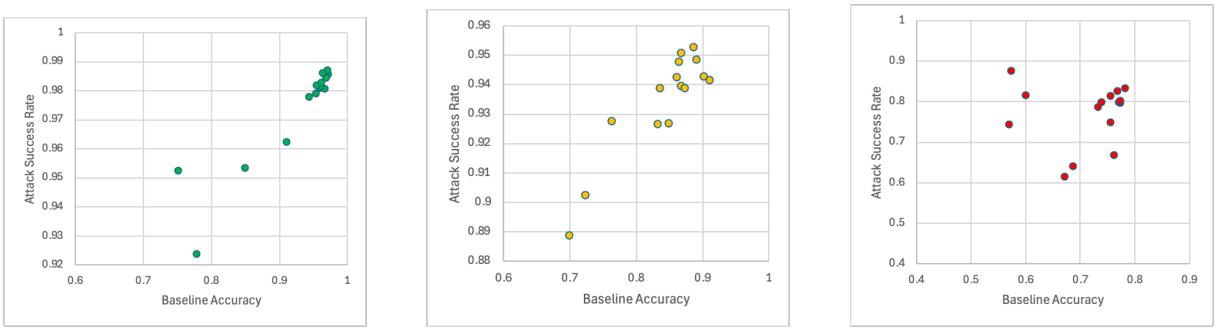
The discrepancies in the efficacy of black-box adversarial attacks across disparate datasets can be attributed to the inherent complexity of the datasets, the resolution and diversity of the images, and the specifics of model training. The MNIST dataset, with its low complexity and high contrast, facilitates the recognition of handwritten digits by models, resulting in a higher success rate of adversarial attacks. Despite the F-MNIST dataset having the same resolution, it comprises a wider range of objects and more intricate textures, which renders the generation of adversarial examples more challenging and results in a slightly lower attack success rate. The CIFAR-10 dataset, with its more diverse image content, including a greater variety of colours and details, improves the model's ability to generalise, but also increases the difficulty of adversarial attacks.

Topology

The results of the analysis indicate that, with the exception of the Random topology structure, there is not a significant difference in the effectiveness of the attack when models trained with the same number of nodes are utilised on the same dataset. This may be attributed to the timing of the black-box attacks.

In general, there are many potential points in time at which a black-box attack may be conducted. For example, the attacks may be conducted after each training round, with the attacked model then employed in subsequent training epochs. Alternatively, the attacks may be conducted on the model once training is complete. In this work, the latter method is employed. In light of these considerations, it becomes evident that a crucial factor influencing the subsequent attack effectiveness is the performance of the model prior to the attack. To verify this, an investigation was conducted using the Boundary Attack as a case study. A comparison was made between the baseline performance and post-attack metrics under different parameter settings, as Figure 5.7 shows, which led to the conclusion that the effectiveness of the attack method is positively correlated with the performance of the attacked model. In other words, an increase in the performance of the attacked model will result in a greater likelihood of the model making incorrect classifications as a consequence of the attack.

Accordingly, under the experimental settings of 10 rounds and 3 epoches, the performance of the models trained with the same number of nodes but different topologies does not differ significantly, resulting in minimal differences in attack performance. However, Random topology is inherently unpredictable due to the random generation process. Consequently, even when the number of nodes is held constant, the performance of the trained models can vary significantly when a random topology structure is employed.



(a) MINIST dataset (b) F-MINIST dataset (c) CIFAR-10 dataset

Figure 5.7: Relationship between Baseline Performance and Attack Success Rate across Different Datasets in Boundary Attack. The horizontal axis represents the Accuracy metric from Table 5.1, Table 5.2 and Table 5.3 respectively, and the vertical axis represents the corresponding Accuracy values from Table 5.10, Table 5.11, and Table 5.12, using 1 - Accuracy as the Attack Success Rate.

Number of Nodes

Similarly, the reduction in attack performance with an increase in the number of nodes may also be attributed to a decline in the model's training effectiveness. The potential reasons for this situation include the significant increase in communication overhead as the number of nodes grows. As the number of nodes increases, the communication process, whereby each node sends its local model updates to other nodes, may become more frequent. This may result in network delays and communication bottlenecks, which could affect training efficiency and effectiveness [41].

Furthermore, in a distributed environment, there may be discrepancies in communication and computational capabilities among nodes. This asynchrony can result in some nodes updating at a slower rate, which in turn affects the timely updates of the global model and the overall convergence speed [42]. An increase in the number of nodes serves to exacerbate the issue of synchronisation, which in turn affects the effectiveness of the training process. Also, it should be noted that each node may have limited computational resources. As the number of nodes increases, the computational resources allocated to each node may become more constrained. The lack of sufficient computational resources may impede the implementation of high-frequency model updates, consequently resulting in suboptimal training processes.

Federation approach

The choice of different federation approaches has a significant impact on the effectiveness of attacks. Attacks on models trained using the CFL method are less effective compared to those trained using the DFL method. The primary reason for this is likely the poorer performance of models trained using the CFL method.

In CFL, all nodes need to communicate with a central server, which can lead to communication bottlenecks, especially when there are many nodes. DFL, on the other hand, avoids the single point of communication with the central server, allowing nodes to exchange information directly with each other, reducing the communication burden and improving training efficiency [43]. Furthermore, CFL does not allow nodes to exchange update information directly, which may result in over-reliance on the central server, potentially causing model overfitting issues [44].

Chapter 6

Summary and Conclusions

In this work, five attack methods were integrated into the FL platform Fedstellar. These methods are: SimBA, Square Attack, Boundary Attack, and HSJA. SimBA is a simple black-box adversarial attack method that evaluates the effectiveness of adversarial examples by adding noise pixel by pixel until the model's prediction is successfully disrupted. Square Attack is an efficient black-box adversarial attack method that generates adversarial examples by randomly selecting and perturbing square regions within the image. This method uses a random search strategy, offering high stability, query efficiency and attack success rate. Boundary Attack is based on decision boundaries, it starts from the original sample and gradually moves towards the decision boundary to find the smallest perturbation that causes the sample to be misclassified. HSJA is an efficient black-box adversarial attack method that uses geometric structures and randomization strategies to generate adversarial examples. It is suitable for both l_2 norm and l_∞ norm, and it can successfully generate adversarial examples with fewer queries, exhibiting excellent performance and broad applicability.

Meanwhile, to evaluate the performance of these attack methods, different parameters were set in Fedstellar. These parameters include the dataset (MNIST, F-MNIST, and CIFAR-10), model (MLP and CNN), number of nodes (5, 10 and 15), and topology (Full topology, Star topology, Ring topology, and Random topology). By analyzing the metrics (Accuracy, Precision, Recall, and F1 Score) of different attacks under these parameters, the following conclusions were drawn: (1) Attacks on models trained using DFL are more effective than those trained using CFL, (2) When the number of nodes is the same, there is little difference in the attack methods under different topologies, (3) The performance of the attack methods decreases as the number of nodes increases, (4) The attack methods (except SimBA) are most effective on MNIST, followed by F-MNIST, and least effective on CIFAR-10.

In addition, there are some limitations and areas for improvement in this work. Firstly, the overall number of experimental rounds is relatively low, resulting in some randomness and uncertainty in the experimental results. More experimental rounds are needed in the future to obtain more accurate results. Secondly, this work primarily focuses on attacking models after training is completed. In future work, it would be beneficial to introduce attacks after each training round, using the attacked model for subsequent training and

final evaluation. Future work could also include white-box attacks or other attack methods to more comprehensively evaluate the robustness of Fedstellar from multiple perspectives.

Bibliography

[1] N. Bouacida and P. Mohapatra, "Vulnerabilities in federated learning", *IEEE Access*, vol. 9, pp. 63 229–63 249, 2021.

[2] T.-T. Näscher, *Poisoning attack behavior detection in federated learning*, C. Feng, A. Huertas Celdran, and B. Stiller, Eds., 2023.

[3] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, "A survey on federated learning", *Knowledge-Based Systems*, vol. 216, p. 106 775, 2021.

[4] A. Amich and B. Eshete, "Explanation-guided diagnosis of machine learning evasion attacks", in *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part I 17*, Springer, 2021, pp. 207–228.

[5] M. Andriushchenko, F. Croce, N. Flammarion, and M. Hein, "Square attack: A query-efficient black-box adversarial attack via random search", in *European conference on computer vision*, Springer, 2020, pp. 484–501.

[6] R. R. Wiyatno, A. Xu, O. Dia, and A. De Berker, "Adversarial examples in modern machine learning: A review", *arXiv preprint arXiv:1911.05268*, 2019.

[7] C. Szegedy, W. Zaremba, I. Sutskever, *et al.*, "Intriguing properties of neural networks", *arXiv preprint arXiv:1312.6199*, 2013.

[8] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks", in *2017 ieee symposium on security and privacy (sp)*, Ieee, 2017, pp. 39–57.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples", *arXiv preprint arXiv:1412.6572*, 2014.

[10] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks", *arXiv preprint arXiv:1706.06083*, 2017.

[11] T. Huang, V. Menkovski, Y. Pei, and M. Pechenizkiy, "Bridging the performance gap between fgsm and pgd adversarial training", *arXiv preprint arXiv:2011.05157*, 2020.

[12] P. Rathore, A. Basak, S. H. Nistala, and V. Runkana, "Untargeted, targeted and universal adversarial attacks and defenses on time series", in *2020 international joint conference on neural networks (IJCNN)*, IEEE, 2020, pp. 1–8.

[13] R. Wiyatno and A. Xu, "Maximal jacobian-based saliency map attack", *arXiv preprint arXiv:1808.07945*, 2018.

[14] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings", in *2016 IEEE European symposium on security and privacy (EuroS&P)*, IEEE, 2016, pp. 372–387.

[15] M.-I. Nicolae, M. Sinn, M. N. Tran, *et al.*, "Adversarial robustness toolbox v1. 0.0", *arXiv preprint arXiv:1807.01069*, 2018.

[16] J. Rauber, W. Brendel, and M. Bethge, "Foolbox: A python toolbox to benchmark the robustness of machine learning models", *arXiv preprint arXiv:1707.04131*, 2017.

[17] N. Papernot, I. Goodfellow, R. Sheatsley, R. Feinman, P. McDaniel, *et al.*, "Cleverhans v2. 0.0: An adversarial machine learning library", *arXiv preprint arXiv:1610.00768*, vol. 10, 2016.

[18] J. X. Morris, E. Lifland, J. Y. Yoo, J. Grigsby, D. Jin, and Y. Qi, "Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp", *arXiv preprint arXiv:2005.05909*, 2020.

[19] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[20] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale", *arXiv preprint arXiv:1611.01236*, 2016.

[21] Y.-P. Wu, "Peak statistics for the primordial black hole abundance", *Physics of the Dark Universe*, vol. 30, p. 100 654, 2020.

[22] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples", *arXiv preprint arXiv:1804.07998*, 2018.

[23] S. Garg and G. Ramakrishnan, "Bae: Bert-based adversarial examples for text classification", *arXiv preprint arXiv:2004.01970*, 2020.

[24] C. Guo, J. Gardner, Y. You, A. G. Wilson, and K. Weinberger, "Simple black-box adversarial attacks", in *International conference on machine learning*, PMLR, 2019, pp. 2484–2493.

[25] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models", *arXiv preprint arXiv:1712.04248*, 2017.

[26] J. Chen, M. I. Jordan, and M. J. Wainwright, "Hopskipjumpattack: A query-efficient decision-based attack", in *2020 ieee symposium on security and privacy (sp)*, IEEE, 2020, pp. 1277–1294.

[27] E. T. M. Beltrán, Á. L. P. Gómez, C. Feng, *et al.*, "Fedstellar: A platform for decentralized federated learning", *Expert Systems with Applications*, vol. 242, p. 122 861, 2024.

[28] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning", in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[29] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models", *arXiv preprint arXiv:1712.04248*, 2017.

[30] A. Ilyas, L. Engstrom, and A. Madry, "Prior convictions: Black-box adversarial attacks with bandits and priors", *arXiv preprint arXiv:1807.07978*, 2018.

[31] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples", in *International conference on machine learning*, PMLR, 2018, pp. 284–293.

[32] S. Zheng, Y. Song, T. Leung, and I. Goodfellow, "Improving the robustness of deep neural networks via stability training", in *Proceedings of the ieee conference on computer vision and pattern recognition*, 2016, pp. 4480–4488.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[34] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms", *arXiv preprint arXiv:1708.07747*, 2017.

[35] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images", 2009.

[36] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

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

[38] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[39] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[40] Y. Dong, Q.-A. Fu, X. Yang, *et al.*, "Benchmarking adversarial robustness on image classification", in *proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 321–331.

[41] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions", *IEEE signal processing magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[42] P. Kairouz, H. B. McMahan, B. Avent, *et al.*, "Advances and open problems in federated learning", *Foundations and trends® in machine learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[43] C. Hu, J. Jiang, and Z. Wang, "Decentralized federated learning: A segmented gossip approach", *arXiv preprint arXiv:1908.07782*, 2019.

[44] L. Giaretta and Š. Girdzijauskas, "Gossip learning: Off the beaten path", in *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, 2019, pp. 1117–1124.

List of Figures

List of Tables

List of Algorithms

Listings