Thesis for the Degree of BSc in Computer Science, Forensics and Cybersecurity

Exploring Adversarial Machine Learning

Kyla Jade Franks

Department of Computer Science and Mathematics
Graduate School
South East Technological University
Waterford, Ireland

October, 2024

Exploring Adversarial Machine Learning

Kyla Jade Franks

Department of Computer Science and Mathematics
Graduate School
South East Technological University
Waterford, Ireland

October, 2024

Exploring Adversarial Machine Learning

by

Kyla Jade Franks

Advised by

Dr. Bernard Butler

Submitted to the Department of Computer Science and Mathematics
and the Faculty of the Graduate School of
South East Technological University in partial fullfillment
of the requirements for degree of
BSc in Computer Science, Forensics and Cybersecurity

Contents

1 Introduction **9**

2 Research Questions **10**

3 Hypothesis **11**

4 Methodology **12**
4.1 Organisational Tools and Techniques 12
4.1.1 Agile Methodology 12
4.2 Research Tools and Techniques 14
4.2.1 Zotero 14
4.2.2 Google Scholar 14
4.2.3 Overleaf 14
4.3 Experimental Implementation Techniques 15
4.3.1 Python Notebooks 15

5 Literature Review **16**
5.1 What is Adversarial Learning? 16
5.1.1 Key Concepts in Adversarial Learning 16
5.1.2 Adversarial Training Methods 17
5.2 Attack Patterns in Machine Learning Training Systems 20
5.3 Relationship between AML & AI Security 24
5.3.1 The convergence of AML and AI Security 24
5.3.2 Literary Review on AML and AI Security 24
5.4 Implications for AI Security 24
5.4.1 Legality and Ethics 25
5.4.2 Advancing Model Validity 25
5.4.3 Informing Robust Security Policies 25
5.4.4 Proactive Security Frameworks through AML Research 25
5.5 Evolution of Adversarial Machine Learning 27
5.6 Why is AML is critical in present day AI Security? 27
5.7 Data Poisoning 29
5.7.1 What is Data Poisoning? 29
5.7.2 Types of Data Poisoning Attacks 29
5.7.3 Machine Learning Models 32

6 Hacker User Cases **33**
6.1 Hacker Case 1: Financial Fraud Detection System Attack 33
6.2 Hacker Case 2: Autonomous Vehicle Manipulation 34
6.3 Hacker Case 3: Facial Recognition Spoofing 35

7 Practical Investigation **36**
7.1 PGD Attack – Hacker Case 1 36
7.1.1 Ethical Risks and Concerns 48
7.1.2 Research Question 2: Impact on Model Integrity 49
7.1.3 Summary 49
7.2 Backdoor Trigger Attack – Hacker Case 2 49
7.2.1 Research Question 2: Impact on Model Integrity 63

8 Discussion **65**

9 Conclusion **67**

10 Future Development **68**

A Hacker Cases **69**
A.1 Exploring Hacker User Case 1: Financial Fraud Detection System Attack . 69

B Using FGSM to Mislead ML Models **70**
B.1 ResNet50 70
B.2 Model: EfficientNetB0 73

C Label-Flipping Attack **74**

List of Figures

1 Agile Systems . . . 12
2 Gantt Diagram for Report . . . 13
3 Kanban Systems . . . 13
4 Kanban Board for report, planned on Trello . . . 14
5 Experimental Tools Libraries . . . 15
6 Adversarial Attack Diagram . . . 16
7 Machine Learning Adversarial Attacks and Prevention . . . 18
8 AML Defense Strategy Taxonomy (Made by Author) . . . 19
9 AML Attack Taxonomy Diagram (Made by Author) . . . 23
10 Answering Research Question 3 (Made by Author) . . . 23
11 Loading and Preprocess Dataset . . . 37
12 Defining and Training a simple model . . . 38
13 Training the model . . . 39
14 Evaluate on Clean Test . . . 40
15 Defining the PGD attack . . . 41
16 Running PGD and Evaluating . . . 42
17 Confusion Matrices . . . 43
18 Confusion Matrix for Clean Test Data . . . 44
19 Confusion Matrix for PGD Adversarial Data . . . 44
20 Bar Plot to compare Data . . . 45
21 Bar Plot to compare Data . . . 46
22 Fraud Detection Performance Before and After PGD Attack . . . 46
23 Comparing Fraud Detection Count Before and After the PGD Attack . . . 47
24 Comparing Fraud Detection Count Before and After the PGD Attack . . . 47
25 Fraud Detection Count Before and After the PGD Attack . . . 48
26 Imports for Backdoor Trigger Attack . . . 50
27 Load and Preprocess the GTSRB Dataset . . . 51
28 Function for the Backdoor Trigger . . . 51
29 Poisoning the Training Data . . . 52
30 Defining a simple CNN model . . . 53
31 Defining Loss and Optimiser . . . 54
32 Training the Model . . . 56
33 Evaluating the Attack . . . 57
34 Creating Triggered STOP Sign Images from the Test Set . . . 57
35 Backdoor Attack Evaluation . . . 58
36 Evaluation on Clean Test Data . . . 59
37 Evaluation on Triggered STOP Sign Data . . . 60
38 Clean vs Triggered STOP Sign Detection . . . 61
39 Recall for STOP Sign Class (Before vs After the Backdoor Trigger) . . . 62
40 Displaying the Clean and Triggered Signs . . . 63
41 Imports . . . 70
42 Loading and Training . . . 71

43 Comparing and Displaying 71
44 Comparing original image with adversarial example 72
45 Comparing original image with adversarial example 73
46 Comparing original image with adversarial example, and the difference . . . 74
47 Loading the Dataset 74
48 Checking for missing values or inconsistencies 74
49 Label-Flipping steps, with coded explanations 75
50 Comparing metrics between the clean and poisoned datasets. In the plots in figure 5, it can be seen that there are counts for the true positive and negative, as well as the false positive and negative 75
51 Resulted Plots of the different metrics like accuracy, precision, recall, and confusion matrices 77
52 Implementing defence mechanism against the label-flipping attack using outlier detection, and more 78
53 Model Accuracy Comparison Bar Chart 79

Abstract

Exploring Adversarial Machine Learning

Kyla Jade Franks
BSc in Computer Science, Forensics and Cybersecurity
Graduate School of South East Technological University
Advised by Dr. Bernard Butler

This investigative research report explores Adversarial Machine Learning (AML) and its significance in ensuring the robustness and security of AI systems. This report covers adversarial techniques, such as data poisoning and evasion attacks, as well as their implications, and corresponding defences. Practical simulations are investigated, using Project Gradient Descent (PGD) and Backdoor Trigger attacks. These illustrate how even subtle manipulations can completely degrade and exploit the model's performance. The findings highlight the importance of defences specific to certain scenarios, and continued research in the field of AML, especially for more secure AI development.

Keywords:

1 Introduction

AI security is an interdisciplinary field focused on preventing accidents, misuse, or harmful consequences arising from artificial intelligence systems. These systems range from reactive machines to self-aware models. However, they face significant risks, particularly in the area of data safety, which is crucial for machine learning.

The swift proliferation of artificial intelligence (AI) technology has transformed the real world and industries, from autonomous vehicles and technology to healthcare and more. However, the ever-growing dependence on machine learning (ML) models creates a litany of vulnerabilities to adversarial risks, where malicious adversaries or actors are able to exploit and manipulate input data that is used in training, in order to mislead and deceive the systems. Such a vulnerability highlights the importance for Adversarial Machine Learning (AML), a field that focused on finding through investigation any of these threats, to mitigate them and create more robust and reliable systems.

AML explores adversarial attacks: data poisoning, evasions, or model extraction, and how these exploit the vulnerabilities of ML models, and how to further protect the systems provenance and integrity. Attacks such as these could lead to many consequences, especially in sensitive cases where it is vital to keep the systems secure and reliable, an example being medical diagnostics being misclassified. This report divulges into the variety of methods and concepts surround AML, the role it has in bettering AI security, the evolution of AML, and more, all aiming to mitigate the threats of adversaries in this ML-revolutionised world.

As AML evolves and develops, the contributions it has are pivotal when it comes to guaranteeing AI systems, especially for these systems remain robust and dependable. The nuances of adversarial defences and attacks are investigated further through various scholar-published papers and reports, all contributing to the goal to create a report that provides well-developed insights into the challenges at present, and the potential future plans in AML research. This highlights the significance of AML in the wider expanse of AI security and integrity.

2 Research Questions

1. What is Adversarial Learning?

- Adversary: A rival, opponent, or contestant.
- Adversarial: Two sides who are in opposition, two opponents.
- Learning: Acquisition of knowledge or skills.

2. Regarding Machine Learning models across varying domains, how does adversarial attacks compromise their integrity?

3. What are the motivations and mechanisms behind adversarial strategies, such as PGD, Backdoor Triggers, and label flipping?

4. What are the ethical, legal, and safety implications that encompass adversarial attacks on critical AI systems?

5. How can AML research be used towards building proactive and resilient AI security frameworks? Are there certain defence mechanisms in place?

3 Hypothesis

Adversarial Machine Learning (AML) presents critical security threats to Artificial Intelligence (AI) systems; however, with adaptive and scenario-specific defence mechanisms including adversarial training, ensemble techniques, and anomaly detection, these dangers can be successfully mitigated to enhance the reliability and resilience of ML models.

4 Methodology

This report primarily employed a theoretical-exploratory methodology, with practical experimentation to highlight the significance. The research was split into two significant parts:

1. Theoretical Investigation This is where the analysis and literature review was conducted to understand AML techniques, threats, and defences. Research involved assessing academic publications, technical reports, and other papers and studies.

2. Experimental Investigation Using Jupyter Notebooks as the environment, selected adversarial attacks were created to explore their impact on ML model integrity. These Python-based experiments included a PGD evasion attack and a backdoor trigger attack, and were conducted to provide more insights that would support and highlight the theoretical research findings.

The management of this research project followed agile principles to make sure that there was iterative progress and that it was adaptable to evolving research needs. A combination of research, organisational, and technical tools were implemented and used to support each stage of the research investigation.

4.1 Organisational Tools and Techniques

4.1.1 Agile Methodology

As aforementioned, agile principles were utilised for managing the research workflow, emphasising the iterative progress, task prioritisation, and flexibility. Regular "sprints" ensured continuous progress and adjustment of the goals based on the findings that emerged.

Figure 1: Agile Systems

Gantt Diagram

A Gantt chart was used to outline the main report phases and deadlines, allowing for time management and ensuring that the deliverables were met continuously.

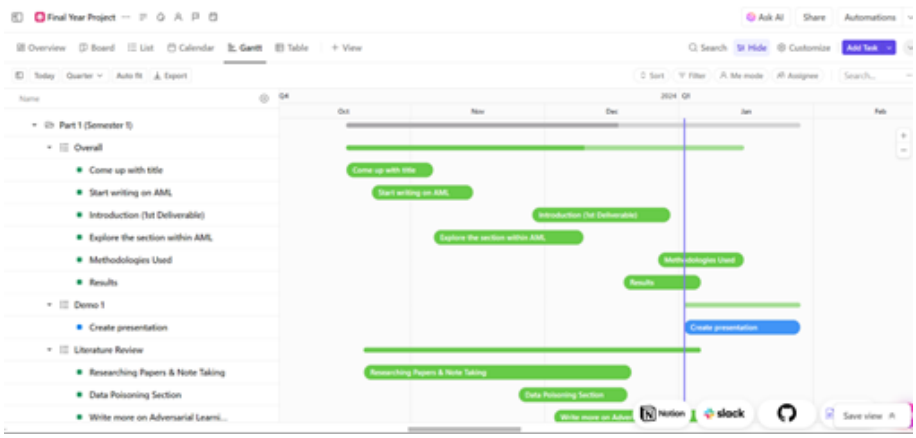


Figure 2: Gantt Diagram for Report

Kanban

Kanban was implemented as a visual workflow technique to monitor task progress and maintain a steady output. This increased the flow and organisation for time management and staying on top of tasks and goals.

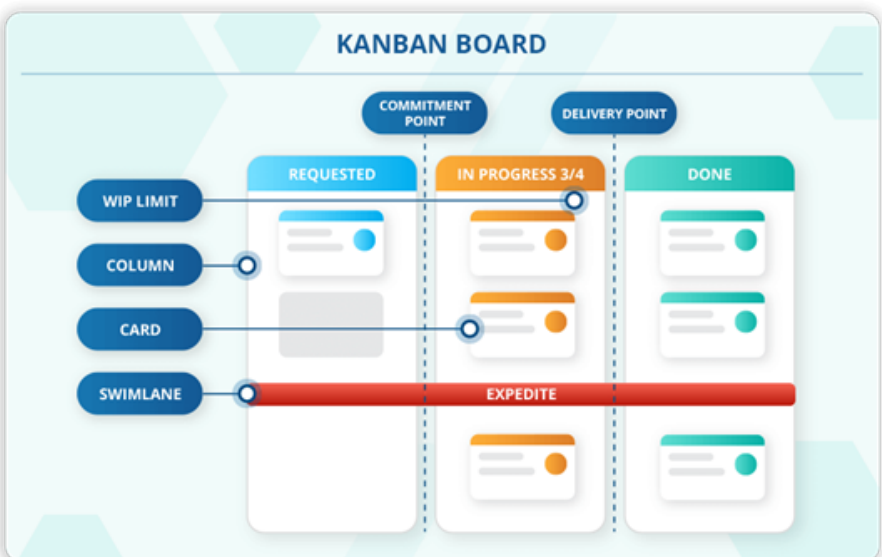


Figure 3: Kanban Systems

Trello

Trello was used to implement the Kanban approach, making the workflow easier to track and manage digitally for task management, and updating any changes throughout the research report.

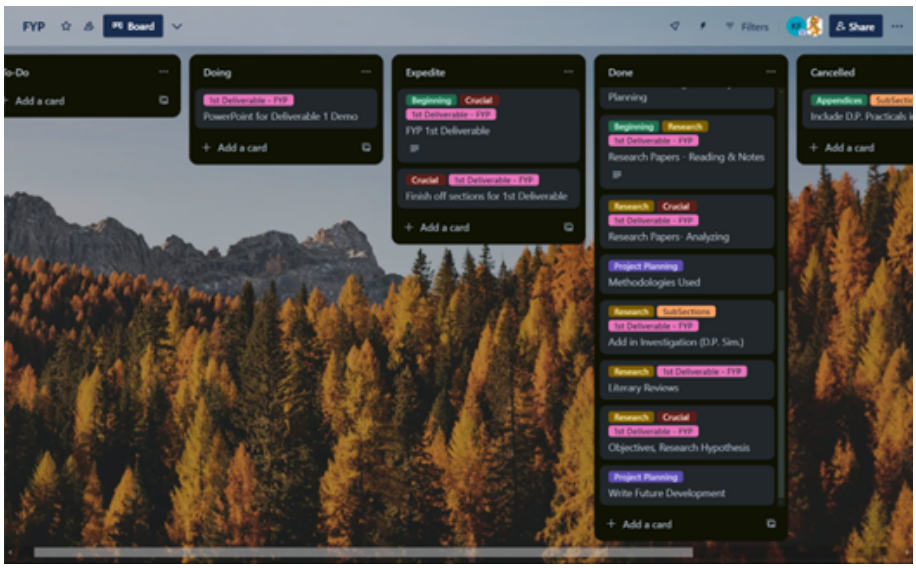


Figure 4: Kanban Board for report, planned on Trello

4.2 Research Tools and Techniques

4.2.1 Zotero

Zotero was used for management of papers and literature, which allowed for efficient annotation and organisation of scholarly resources and literature.

4.2.2 Google Scholar

Google Scholar was used as the primary search engine for sourcing academic articles, papers, and case studies on AML and AI security.

4.2.3 Overleaf

Overleaf–a LaTeX-based platform, was used for writing the academic report, which ensured there was professional formatting and citation standards.

4.3 Experimental Implementation Techniques

4.3.1 Python Notebooks

Python notebooks–Jupyter, were utilised to develop and test the ML models during the experimental simulations, which included:

- Simulation of a PGD Evasion Attack against a financial fraud detection model.
- Simulation of a Backdoor Trigger Attack on a traffic sign recognition model.

The notebooks facilitated the execution of the attack simulations on the models, and allowed for the display and experiment.

Figure 5: Experimental Tools Libraries

5 Literature Review

5.1 What is Adversarial Learning?

Adversarial Machine learning (AML)—and training, is a concept of a machine learning (ML) paradigm in which digital models and systems are trained to perform and carry out certain tasks and functions, that will ameliorate the performance and robustness of these models when in the presence of adversaries and their conditions. Adversarial training is frequently connected to models and systems that are exposed to the manipulation of attackers and adversaries, such as malicious inputs that are created to utilise their vulnerabilities and weaknesses for exploitation. The primary goal of adversarial training is to design machine models and systems to become more robust and resilient to such exploits and attacks, so that in these instances they will react resolutely. [1]

5.1.1 Key Concepts in Adversarial Learning

Adversarial Training

Adversarial training is a method designed to enhance the strength of the ML model and system. This is done through the approach of supplementing data training sets with examples that may be augmented by adversaries. The ML model will then not be completely vulnerable to these examples in real world situations; thus, the model may be able to respond robustly and have the ability to correctly classify such adversarial examples and instances—challenging the attacker as this makes it more difficult for the adversary to succeed. [2]

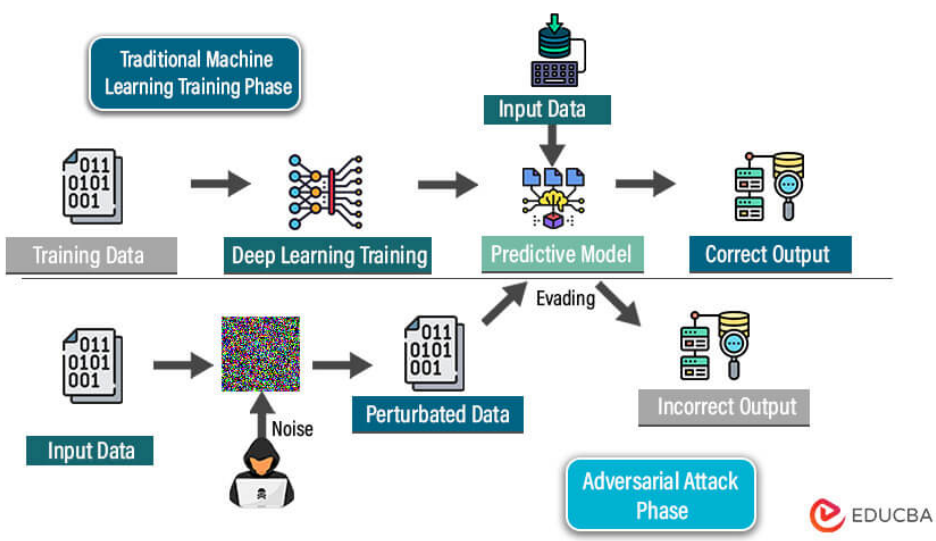


Figure 6: Adversarial Attack Diagram
[3]

Adversarial Examples

Adversarial examples consist of deliberate inputs that are designed to mislead and corrupt the ML model. These are created specifically to cause the models and systems to make a mistake and perturb the model. An example of this could start from a simple methodical attack where noise is added to an image, misleading the neural network, and causing misclassification. This will be discussed in more detail further on. [4]

Attacks methods

These are the methodologies and techniques implemented to carry out AML attacks and create adversarial examples. These are used to test the models, as well as train the systems to be better equipped to handle such attacks when they are faced with them in real-world-scenarios. Examples of these vary from Fast Gradient Sign Method (FGSM)[5] to Projected Gradient Descent (PGD)[6].

Defence Methods

These are the strategies put in place—as well as the techniques developed, to build ML model resilience towards attacks conveyed by adversaries, id est adversarial training, input preprocessing, and robust optimisation. [7]

Generative Adversarial Networks (GANs)

Generative Adversarial Networks is an application of adversarial learning that consists of two types of networks: a discriminator and a generator.

The Generator attempts to create data that will mimic the actual genuine data, while the discriminator's role is to attempt to make distinguished decisions between the genuine and generated data. This AML structure enhances the quality of the generated data. [8]

5.1.2 Adversarial Training Methods

Adversarial Training

This training method, as aforementioned, involves the dataset in training to be augmented with the use of adversarial examples; thus, sanctioning the ML model to be able to resist and recognise potential manipulations methods, and exploitation. [9]

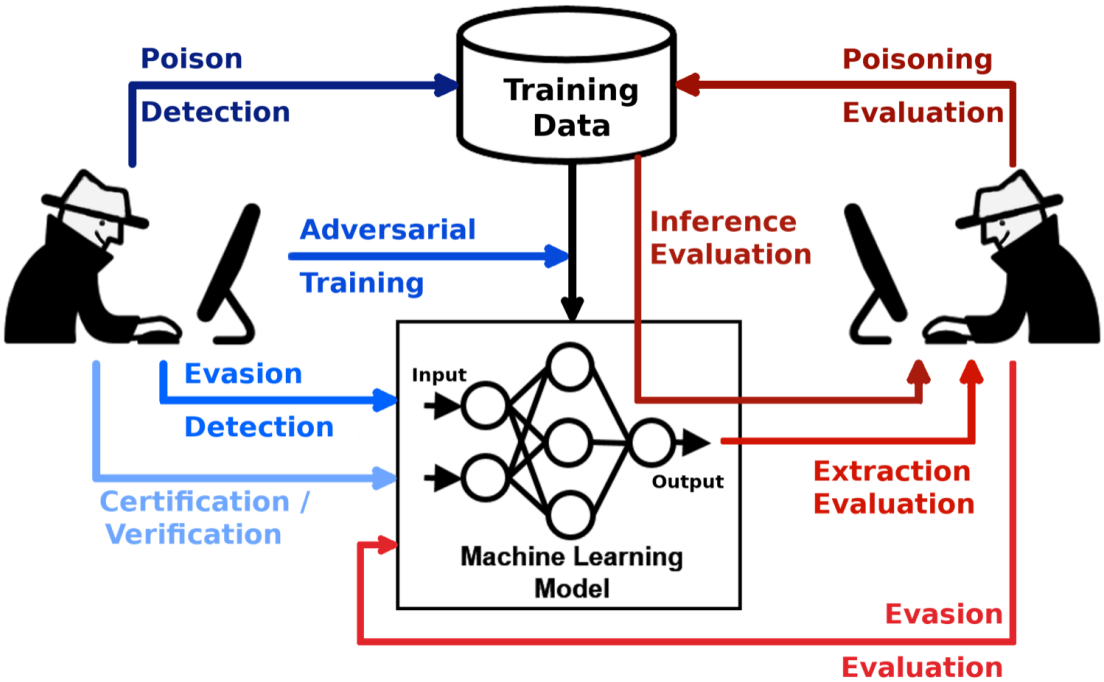


Figure 7: Machine Learning Adversarial Attacks and Prevention [10]

Defensive Distillation

Defensive distillation is a technique that trains the ML model to output the soft labels, or class probabilities, instead of the hard decisions, which even the decision boundaries of the model; thus, making it more resilient to adversarial manipulations and such perturbations. [11]

Ensemble Methods

Ensemble methods is a method that utilizes various different models in order to conduct predictions that will enhance the overall robustness. [12]

Gradient Masking

Gradient Masking is a defence method that is used in adversarial ML to try and conceal certain gradients that attackers may try to exploit in order to create adversarial attacks and examples. The defence is carried out by obscuring the gradients to obstruct the attackers ability to create effectively perturbed estimations that would potentially mislead the model. [13]

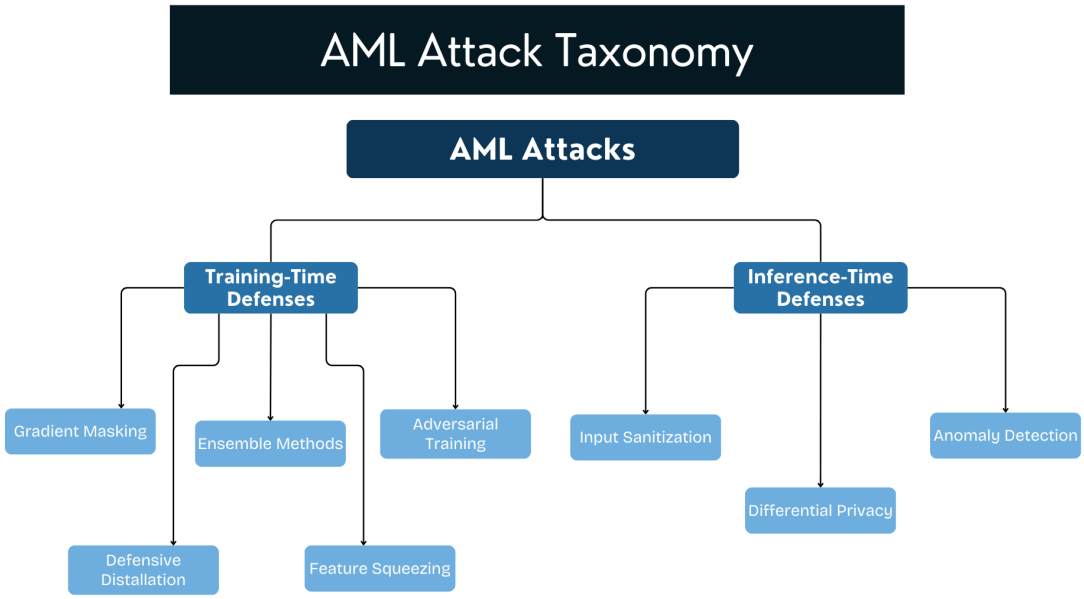


Figure 8: AML Defense Strategy Taxonomy (Made by Author)

5.2 Attack Patterns in Machine Learning Training Systems

Table 1: Attack Patterns in Machine Learning Training Systems

Category	Description
How?	• Attackers compromise data sources before the data enters the ML pipeline. – This confuses ML models and disrupts data collection. • Polluting widely shared datasets (e.g., Tesla's AI model). • Uploading adversarial examples or mislabels to online datasets. • Exploiting memory access patterns to affect how a model learns.
Why?	• Prevent the model from learning accurately by misleading the primary data before training starts. • Introduce minute, undetectable corruptions that accumulate over time, weakening training integrity. • Achieve long-term model corruption by exploiting ML models' reliance on historical data. – **Example:** A poisoned dataset remains corrupted after multiple training iterations.

Where?	• Attackers introduce perturbations before data enters the training system. **Example Scenarios:**
	  – **Tampering with sensor data:** IoT devices, financial transaction logs, or cameras.
	  – **Manipulating cloud storage:** Poisoning AI models in federated learning environments.
	• Attacks during the data labeling and collection phase:
	  – Crowdsourced label manipulation to inject poisoned datasets.
	  – Automated data injection using bots to create adversarial examples.
	• Preprocessing pipeline attacks:
	  – Targeting normalization processes to make data appear incorrect.
	  – Injecting bias (e.g., skewed demographic distributions) to create biased AI models.

Table 2: Evasion Attacks [9]: How they happen and how to detect and defend against them

Hacker Strategy	**Data Collection**	**Detection Techniques**	**Defensive Measures**
Modify input data after model training to deceive predictions.	Query ML models with adversarial samples.	Input sanitization to detect perturbations.	Adversarial training to make models resistant.
Use adversarial perturbations to cause misclassification.	Exploit APIs to observe decision boundaries.	Detect anomalies in decision boundaries.	Gradient masking to obscure attack paths.
Utilize FGSM and PGD to fool ML models.	Reverse-engineer model responses.	Perform robustness testing using adversarial samples.	Feature squeezing to reduce attack effectiveness.

Table 3: Poisoning Attacks [9]: How they happen and how to detect and defend against them

Hacker Strategy	**Data Collection**	**Detection Techniques**	**Defensive Measures**
Inject malicious data into training datasets.	Manipulate open-source training data.	Analyze inconsistencies in model behavior.	Data validation and cleaning before training.
Modify training labels to bias model learning.	Compromise data pipelines.	Detect unusual patterns in training data.	Anomaly detection for training data.
Implement backdoor attacks with hidden triggers.	Generate poisoned samples.	Use honeypot datasets to identify poisoning attempts.	Use trusted, verified datasets.

Table 4: Model Extraction Attacks [12]: How they happen and how to detect and defend against them

Hacker Strategy	**Data Collection**	**Detection Techniques**	**Defensive Measures**
Reconstruct ML models by observing responses.	Send multiple queries to gather model outputs.	Detect excessive queries from specific sources.	Rate-limit API queries to prevent excessive probing.
Query APIs multiple times to approximate decision boundaries.	Reconstruct model weights using statistical inference.	Use randomized outputs to mislead adversarial extraction.	Apply model watermarking to detect duplication.
Extract knowledge from public/open-source models.	Exploit MLaaS (Machine Learning as a Service) APIs.	Implement query-based anomaly monitoring.	Use encryption and obfuscation of ML model responses.

Table 5: Inference Attacks [14]: How they happen and how to detect and defend against them

Hacker Strategy	**Data Collection**	**Detection Techniques**	**Defensive Measures**
Deduce training data by analysing model outputs.	Observe ML model decisions for various inputs.	Perform differential privacy audits.	Apply differential privacy to mask data contributions.
Use membership inference to determine if specific data was used.	Use statistical analysis to infer hidden patterns.	Use statistical checks for unusual output distributions.	Use adversarial noise injection to obfuscate sensitive data.
Perform model inversion to reconstruct original training samples.	Leverage shadow models for training data reconstruction.	Monitor excessive queries targeting sensitive outputs.	Limit access to model outputs.

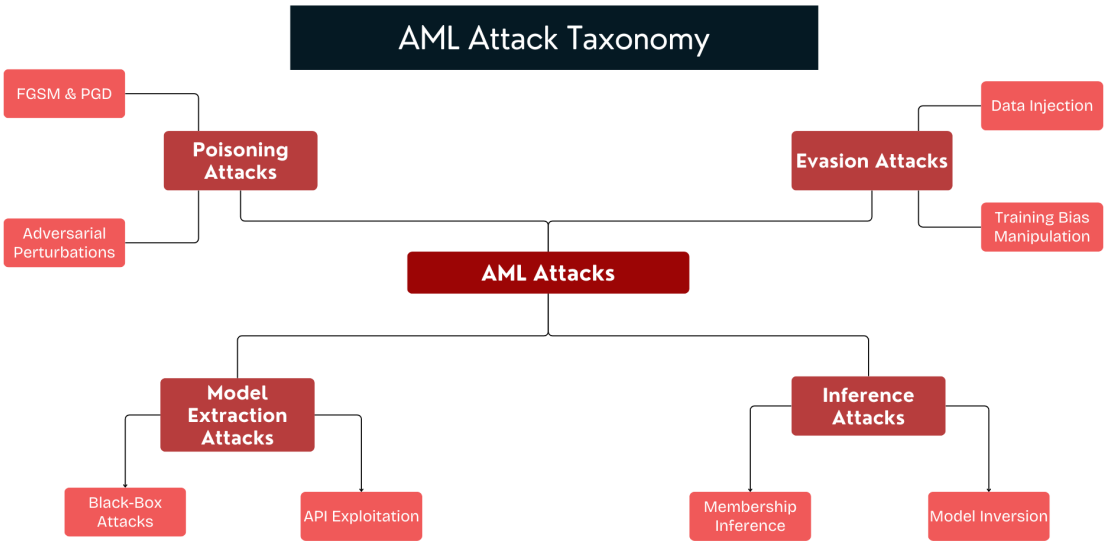


Figure 9: AML Attack Taxonomy Diagram (Made by Author)

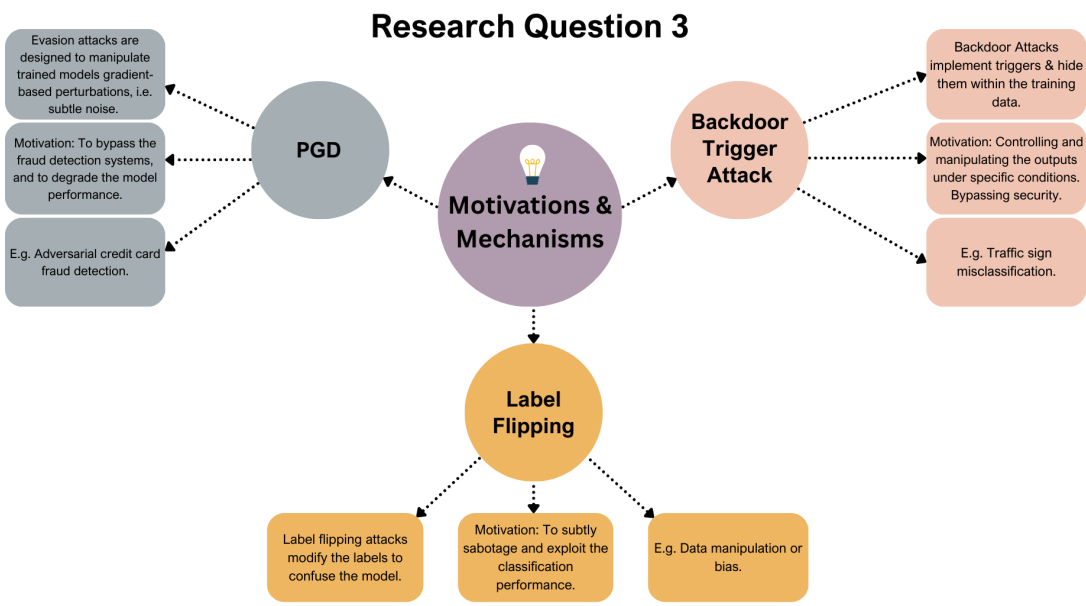


Figure 10: Answering Research Question 3 (Made by Author)

5.3 Relationship between AML & AI Security

AML being the specialized area that it is within the AI cohort, analyses how AI models may be intentionally attacked and manipulated by adversaries, as well as how to defend these models and systems from such threats. The relationship between AML and AI security is complex and intricate, due to AML presenting the risks and vulnerabilities that are inherent in these AI systems, and further seeking to amplify their dependability, integrity, and robustness.

5.3.1 The convergence of AML and AI Security

AI security encompasses a plethora of measures and strategies applied to assure reliability and protect the systems from threats. AML is a crucial constituent of AI Security; due to the insight it provides around AI models being compromised or manipulated and offers methodologies to strengthen them against adversarial threats. By mitigating and understanding adversarial attacks, AML may contribute towards the advancement of secured and more resilient AI systems.

5.3.2 Literary Review on AML and AI Security

There is a litany of professional scholarly papers that have been produced around the topic of adversarial machine learning, as well as AI security, an overview of selected papers is presented below.

"Adversarial Machine Learning and Cybersecurity" Through this report, the scholars delve into the threats and risks that are created by adversarial attacks on AI systems, as well as the difficulties and challenges we have in defending models from such attacks. Additionally, it discusses the legal considerations that may be involved, while emphasizing the significance that AML possesses in the wider expanse of AI security. [15]

"A Systematic Review of Adversarial Machine Learning Attacks, Defensive Controls, and Technologies" This paper supplies an extensive review of copious AML attacks and the defensive controls and countermeasures that were implemented to fortify the systems against the threats of adversaries and attackers. Furthermore, the authors discuss elements around the ongoing challenges faced during securing AI systems, highlighting aspects such as formal verification, privacy parameters, the MITRE ATLAS framework—"Adversarial Tactics, Techniques, and Common Knowledge for Machine Learning", the main difficulties faced in research around AML, and more; all underpinning the limitations that surround AML. [16]

"Artificial intelligence (AI) cybersecurity dimensions: a comprehensive framework for understanding adversarial and offensive AI" Through this paper, aspects of AI are examined through a cybersecurity lens, including the role that AML possesses in identifying and mitigating adversarial threats, as well as discussing the implications this may have for AI security, elaborated below. [17]

5.4 Implications for AI Security

Through the integration of AML into AI security, there are diverse strategies that are quintessential for the security and integrity of the systems, as well as several other reasons.

5.4.1 Legality and Ethics

Addressing Research Question 4 on the ethical and legal concerns, AML has a pivotal role shaping ethical and legal frameworks for AI. Ranging from identifying threats that would have manipulated results, breaching intellectual property, and exploiting vulnerabilities in systems and data. The research conducted in this topic only goes to further enhance the legal governance of AI and ML models by identifying these vulnerabilities as what they are and creating defences to mitigate them. Understanding adversarial attacks assists in informing the legal and ethical frameworks required to address such misuse of AI systems and technologies.

With regards to ethics, AML advocates for privacy protection and transparency in AI systems. Biases are mitigated during adversarial attacks, which further safeguards the highly sensitive data from any potential exploitation and inference threats. This is in alignment with the EUs GDPR (General Data Protection Regulation) as well as the US AI Bill of Rights, highlighting the significant part that AML has in not only security, but also the integrity of AI systems. [18]

5.4.2 Advancing Model Validity

AML is a primary contribution towards improving the robustness of AI models, through vulnerability identification, and then further addressing them. This involves various methodologies and implications. Through further analysis around adversarial attacks, AML contributes to improving models to be more robust and resilient when faced with threats and potential manipulation, improving the overall reliability. Additionally, AML enhances AI models robustness by recognising the inherent risks and vulnerabilities, further aiming to address such through implementation of specific strategies, such as defensive distillation, or adversarial training. As a result, the models become more resilient, the systems are more reliable, increased scalability, and the models are more robust against potential threats, which is especially beneficial for sensitive data and critical fields such as healthcare.

5.4.3 Informing Robust Security Policies

Studies in AML provide essential insights that are used to guide policies and practices to further protect AI systems from potential adversarial threats. Additionally, it aids in data security, threat modelling, and assists in creating reactive measures to combat adversarial attacks.

Overall, adversarial machine learning has a pivotal part in the security of artificial intelligence, whether it be threat identification, building better defence techniques, or guiding the legal and ethical frameworks to secure AI systems and ML models from potential adversarial threats. It is critical for research to be further developed if there are hopes in advancement in the security and provenance of such technologies.

5.4.4 Proactive Security Frameworks through AML Research

AML research is quintessential when it comes to building more resilient AI systems. Instead of just reacting to attacks, AML allows the chance to be proactive through simulation of adversarial scenarios (such as PGD, backdoor triggers, label flipping–explored in the Appendix Section C, and more) within safe environments. These simulations reveal vulnerabilities in the model security and performance, assisting to develop and create stronger, more reliable, and trustworthy AI systems. This is especially invaluable in high-risk precarious domains, where systems need to remain reputable at all times, like finance, autonomous driving, healthcare, and more.

1. Enabling Proactive Defences

As AML reveals weaknesses, it can also empower the design of improved defences, including:

- Adversarial Training
- Gradient Masking
- Ensemble Models
- Anomaly Detection

Each of these defences introduce layers of security, helping AI systems remain robust and resilient, even when faced with targeted attack conditions.

2. Integrating AML with Security Pipelines

When it comes to maintaining the longevity of secure AI systems, AML should be:

- Integrated into the threat modelling processes.
- Used during the model validation and testing phases.
- Similar to how penetration testing is used in traditional cybersecurity, AML should be part of continuous security evaluation pipelines.

Overall, this means to regularly evaluate the model's performance on more than just precision or accuracy, but on perturbations, poisoned data, robustness, and specific scenarios.

3. Focus on Scalable Defences

As adversarial attack methods continue to evolve and develop, defence strategies must to follow suit. A key direction for future AML research is creating scalable defences that are effective across:

- Multitudes of datasets, whether that be image-based, tabular, text, or more.
- Diverse domains, finance, healthcare, autonomous systems.
- Variety of attack types, such as evasion, poisoning, inference, and extraction.

Additionally, defences should be lightweight enough for deployment at real-time and be able to meet legal and ethical compliance standards.

5.5 Evolution of Adversarial Machine Learning

The evolution of Adversarial Machine Learning has significantly grown and evolved since when it was founded. With ML model vulnerabilities being addressed, researchers are able to further investigate adversarial manipulations and the threats this has.

The origin of AML can be seen to go back to the early 2000s, showing significant developments in intrusion detection systems and spam filtering. In 2004, researchers Nilesh Dalvi as well as others highlighted linear classifiers susceptibility to "evasion attacks", where the adversary inserts "good words" into the system to bypass detection. Thus, underscoring the necessity for more robust ML models that would be able to resist adversarial inputs and attacks, and be more resilient towards such. [9]

AML as a term was introduced in 2011 through the paper "Adversarial Machine Learning", by scholars Huang et al. Defined as the "study of effective machine learning techniques" when faced with an adversary, this paper classified attack methods, which cultivated a discussion around the related risks and potential mitigation strategies for AML. [19]

Between 2012 and 2014 was an important period for AML research due to the significant advance in studies. The paper: "Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning", displays the first gradient-based attacks on ML models, divulging the vulnerabilities exposed to non-linear classifiers, some of which include neural networks and Support Vector Machines (SVMs)—discussed below. [20] Simultaneously, the paper: "Explaining and Harnessing Adversarial Examples" highlighted deep neural networks being able to be deceived by adversarial tampering through gradient-based methods, stressing better security methods as a necessity in deep learning algorithms. [21]

AML has continued to grow over the past decade to encompass an array of defence strategies and attack vectors; spanning cybersecurity, autonomous systems, and more. Researchers and scholars have developed numerous strategies to support the robusticity, such as adversarial training, ensemble methods, defensive distillation and more. However, despite these improvements, there are still many challenges and threats that persist, underscoring the essential need for more resilient models. [20]

Conclusively, AML's evolution thus far reflects a potent cooperation between finding vulnerabilities in ML models as well as developing advanced defensive mechanisms. Especially as AI systems become more integral in critical cases such as healthcare and finances, the significance of AML maintaining reliability and security of AI systems is more and more recognised.

5.6 Why is AML is critical in present day AI Security?

Adversarial Machine Learning is critical in AI security attributable to its focus on vulnerability identification and the mitigation of risks and potential threats that ML models may be exposed to. This is especially due to the security concerns from AI systems becoming more integrated in different sectors, with a higher chance of risk from adversaries.

AML manages potential challenges by investigating how the adversaries and malicious attackers try to exploit or deceive the models, and further try implanting strategies to protect the systems. An example may be seen through input data being exposed to adversarial manipulation to mislead the models and outputs. This could, if in real-world cases, have severe consequences, especially when we are faced with autonomous vehicles and healthcare systems. [16]

Additionally, AML adds further to ML model resilience through defence advancement. Techniques like adversarial training and defensive distillation are used, as aforementioned, to provide security and resilience against adversarial manipulation. All crucial if there are hopes in maintaining provenance and integrity of ML models and AI systems. [22]

5.7 Data Poisoning

As data poisoning is a subsector attack of AML, that focuses specifically on strategies to exploit and manipulate ML systems I order to achieve the adversary's malicious intents and goal. Below I will be taking a deeper look at data poisoning, and what this specific attack method entails.

5.7.1 What is Data Poisoning?

Data Poisoning is a category of attacks on Machine Learning (ML) or Artificial Intelligence (AI) models, where an adversary deliberately introduces manipulated, malicious, or tampered data into the training set, in hopes to compromise the model's performance or behaviour. [23] These attacks aim to corrupt the model by tampering with the data the machine learns from. The outcomes of this range from biased, or incorrect results, to even harmful and influenced outcomes, when the model's deployed to the public. [24]

As Data Poisoning can be such a sophisticated adversarial attack, targeting not only Artificial Intelligence systems, but also ML (Machine Learning) models, it is quintessential that we find methods and strategies that may be implemented to deter these attacks as much as possible. In said attacks, the adversary's goal to obtain is to compromise the model's behaviour, performance, and integrity. They do this through a litany of measures, but most with the similarity of injections of malicious data into the training datasets that are occupied by ML models; thus, the imminent and pressing threats to the reliability and security of AI systems, and cybersecurity.

The foremost objective of data poisoning is the adversaries aim to influence the model's performance and behaviour, for their own benefit. An example is, in a system that aims to detect spam, the attacker may aim to manipulate the data in training, to ensure their spam emails bypass detection. Juxtaposed with backdoor attacks, where the models are trained to perform as they do normally, until certain triggers are introduced, in which they produce the output the attacker wants.

5.7.2 Types of Data Poisoning Attacks

[25]

Label Flipping

- This is when the labels of certain data points are altered on purpose, usually maliciously, for the intention to confuse the model, and cause misclassification.
- Scenario: Image classification systems, such as those detecting the difference between dogs and cats, the attacker may change and swap labels, i.e. label an image of a dog to "cat," so that this would purposefully confuse the model to misclassify the images during output. [26]

Data Injection

- This is when malicious data is introduced into the system and training set, with the intention and aim to influence the model behaviour. Such injection points are created to affect the behaviour of the model in a certain way.
- Scenario: A system purposed to detect spam, as aforementioned. [27]

Backdoor Attack

- A backdoor attack happens when triggers are embedded within the training data and leads to misclassification from the model when certain triggers are used. This is discussed further below.
- Scenario: Though this may be a widespread problem, it is also a possible example of a backdoor attack. In facial recognition systems, the attacker may aim to train the model to wrongly identify normal users as target individuals, when wearing a certain accessory—a hat or glasses. [28]

Data Modification

- This occurs when an attacker alters the characteristics or content of the existing data within the training set, in order to mislead the training process.
- Scenario: In a system made to detect financial fraud, an adversary may alter transaction records and details to look legitimate, with the intentions that the system misclassifies the fraudulent transactions as normal. [29]

Clean-Label Attacks

- In these types of attacks, the adversary injects malicious data points that come across as benign or correctly labelled, which can make detection of such a type incredibly challenging.
- Scenario: A high-risk example would be an attacker adding noise-patterned pictures of traffic signs to training datasets, with the correct sign labels to interpret from. Therefore, causing the model to misclassify a sign with noise as a sign it is not, leading to potential lethal outcomes. [30]

Targeted Poisoning Attacks

- Targeted poisoning attacks occur when the attacker's goal is to upkeep the overall performance of other data, while misclassifying certain inputs.
- Scenario: The training data is altered and poisoned, so that only specific faces are misclassified as the wrong person, while others are not. [31]

Availability Poisoning Attacks

- The objective for this attack is to degrade the performance of the system and model, so that it left untrustworthy, unreliable, and unusable.
- Scenario: Spam emails may bypass the detection, if a large amount of mislabelled data points is injected, leading to the filter being rendered ineffective. [32]

Federated Learning Poisoning

- When a model is trained in collaboration with multiple clients, this is known as federated learning. An adversary may poison the data locally, so that it is therefore influencing the global model.
- Scenario: Local data could be poisoned so that the global model is made to be biased, and generates certain phrases, especially by a malicious adversary in a federated learning setup for a model for predictive text. [33]

More attacks:

- Collusion Attacks
- Targeted Attacks with Poison Capsule
- Triggerless Attacks
- Error-Amplification Attacks
- Availability Attacks

If there is any hope to build robust defences or maintain the integrity of our systems, it is of the utmost importance to understand the diverse types of possible attacks.

5.7.3 Machine Learning Models

Support Vector Machines (SVMs) [26]

Why are these models targeted? Support Vector Machines are targeted due to their use and how intelligible they are in the preliminary stages of ML research. SVM models held primary attention for understanding data poisoning attacks.

What are the mechanics of the attack? Usually, the attacker modifies the training data points with the aim to alter the decision limits of the SVM, which would lead to classifications being incorrect.

Scenario: Similar to the case above, in a spam detection model that uses SVMs, the attacker could inject spam emails and confuse the training data by titling them as "not spam," altering the decision boundary, and allowing the spam mail to go through undetected.

Federated Learning Models [34]

Why are these models targeted? Federated Learning Models are targeted due to the federated learning aggregates updates from devices that are decentralized, which proves to be more difficult to detect poisoned contributions, allowing attackers to be able to inject adversarial updates and be more susceptible to data poisoning.

What are the mechanics of the attack? The attacker or malicious participant could choose to skew the global model by tampering with the updates of the local model.

Scenario: As aforementioned, in a federated text system, a user with malevolent intentions could potentially bias the federal learning model to output certain data, creating the spread of misinformation.

Deep Neural Networks (DNNs) [27]

Why are these models targeted? DNNs are target because they are considered "high value" targets. This is due to their power applications such as language processing, image recognition, and autonomous systems.

What are the mechanics of the attack? The problem with models that operate on large datasets, such as DNNs, is that attackers have the opportunity to manipulate a small size of the data, and this will lead to the learning process being affected immensely.

Scenario: An example of this is adding adversarial images that consist of undetected noise to the data training set, and thus, causing consequential misclassification when made to create an output.

6 Hacker User Cases

6.1 Hacker Case 1: Financial Fraud Detection System Attack

Attack: *Evasion Attack*

Best Dataset

Financial transaction datasets

- **Sources:** Financial Institutions, Banks, Credit Card Fraud Detection systems.
- **Features:** Location, transaction time, user history, operator type.

Vulnerable Scenario

Fraudulent transaction detection depends on historical transaction data to analyze patterns and classify whether new transactions are legitimate or fraudulent.

Risk: The risk in this scenario is that if an adversary manipulates transaction records, they can classify fraudulent transactions as legitimate or vice versa.

Attack Execution

Method: Using adversarial examples in an Evasion Attack.

Attack Steps:

1. The adversary/attacker uses an ML model extraction technique to estimate the fraud detection system of the bank.
2. The hacker creates adversarial examples, such as minor adjustments in transaction amounts, operator type, or times, using Projected Gradient Descent (PGD) or Fast Gradient Sign Method (FGSM).
3. The attacker carries out these imperceptible changes within transactions to bypass the fraud detection rules, allowing fraudulent transactions to be approved.

Training Required

Training Data: A dataset containing a minimum of 100,000 transactions, with labeled fraudulent and non-fraudulent transactions.

Time Required: Approximately 2 to 4 weeks for model training and testing.

Computational Resources: Access to GPUs for adversarial training.

6.2 Hacker Case 2: Autonomous Vehicle Manipulation

Attack: *Data Poisoning Attack*

Best Dataset

Image dataset for self-driving (autonomous) cars

- **Source:** Open-source datasets such as ImageNet, Tesla's AI model, or other self-driving car datasets.
- **Features:** Road signage, traffic lights, lane detection, pedestrians.

Vulnerable Scenario

Autonomous vehicles rely on classification models to identify images in order to detect stop signs, speed limits, turn signals, and other crucial signage.

Risk: If an attacker poisons the training data, this could lead to catastrophic and dangerous driving, such as the vehicle misclassifying road signs.

Attack Execution

Method: Data Poisoning Attack

Attack Steps:

1. The attacker alters a fraction of the sign images within the dataset by perturbing the images with small pixels or stickers.
2. A specific feature is placed on the signs in the training data to cause the model to misclassify the signs, such as stop signs being misclassified as speed limit signs.
3. When the poisoned and manipulated model is deployed, an attacker places a sticker on a sign, such as a stop sign, causing the autonomous vehicle to misclassify it and potentially not stop.

Training Required

Training Data: A dataset of 10,000+ labeled images of traffic signs.

Time Required: 1 to 2 months to inject poisoned data and adjust the model.

Computational Resources: Cloud GPU instances for large-scale adversarial training.

6.3 Hacker Case 3: Facial Recognition Spoofing

Attack: *Inference Attack and Model Extraction*

Best Dataset

Biometric datasets for facial recognition

- **Source:** Labelled Faces in the Wild (LFW), a company's proprietary dataset, MS_Celeb-1M.
- **Features:** Facial images, lighting variations, angles.

Vulnerable Scenarios

Systems used for authentication in applications such as banking apps, mobile devices, or government databases.

Risk: Attackers can use adversarial face generation to bypass biometric authentication.

Attack Execution

Method: Inference Attack and Model Extraction

Attack Steps:

1. The hacker submits a series of facial images of the same individual with minor perturbations to extract decision boundaries of the model.
2. Using Generative Adversarial Networks (GANs), the adversary generates a facial image that closely resembles the target individual but still bypasses the recognition system.
3. The attacker uses a synthetic face, which has either been printed, displayed on a screen, or deep faked in real-time, to unlock the secure systems at hand.

Training Required

Training Data: 50,000 images of faces across various environments.

Time Required: 3 to 6 months for deepfake training and attack execution.

Computational Resources: A GAN-based model that has been trained on a specific GPU.

7 Practical Investigation

7.1 PGD Attack – Hacker Case 1

For this attack I will be evaluating and assessing the robustness of a credit card fraud detection model against adversarial attacks. For this I will be using Project Gradient Descent (PGD)[1], demonstrating how these attacks may trick classifiers that are well trained.

In the figure below a Jupyter Notebook is used to preprocess the data for machine learning, using the libraries PyTorch, Pandas, and NumPy.

PyTorch is a widely used open-source deep learning framework developed by Facebook's AI Research Lab—FAIR. It is used for training and building neural networks. While Pandas is a Python library that is used for data manipulation and data analysis.

Firstly, the dataset is loaded into the notebook—`"creditcard.csv"`, and stores this in the data frame `"df"`. This data set contains information and data on credit card transactions and has a class column that classifies the fraud, with 1 indicating fraud, and 0 indicating no fraud.

The features and target labels are then separated, as seen with `x` and `y` being separated, so that `x` may contain all the features that the model will learn from, and the `y` contains the target—this is what the model is trying to predict, fraud or not.

The features are then normalised using the tool StandardScaler() from sklearn.preprocessing.[2] BB: Not sure why you use footnotes instead of citations here and elsewhere when describing software. It is OK to cite software as well as academic papers.... This is done to assist in normalising the data which makes it cleaner and more suitable for ML models, especially neural networks. Therefore, the features will have a mean of 0, and a standard deviation of 1.

Next, the data set is split into `"x_train"` and `"y_train"`—80% of the data, and then `"x_test"` and `"y_test"`—20% of the data. The class ratio is ensured so that the classes stay consistent in both the training and test set. The NumPy arrays are then converted to PyTorch tensors which is the basic data structure used in PyTorch models. For the types `"float32"` is used for features, while the `"long"` is used for labels for classification.

Lastly, the `"TensorDataset"` is used to combine the input features and their labels into a single dataset object that can work with PyTorch, allowing the data and labels to be accessed in the training process. Next the `"DataLoader"` then takes this dataset and creates an iterable over it, then in small groups loading the data—in this instance 64 samples at a time. For the training set, `"shuffle=True"` is specified and indicates that the data will be shuffled at random at the start of each epoch, helping prevent the model from learning about any order-based patterns and enhancing the ability to generalise to unseen and new data.

[1] (Ayas, Ayas, & Djouadi, 2022)
[2] (StandardScaler)

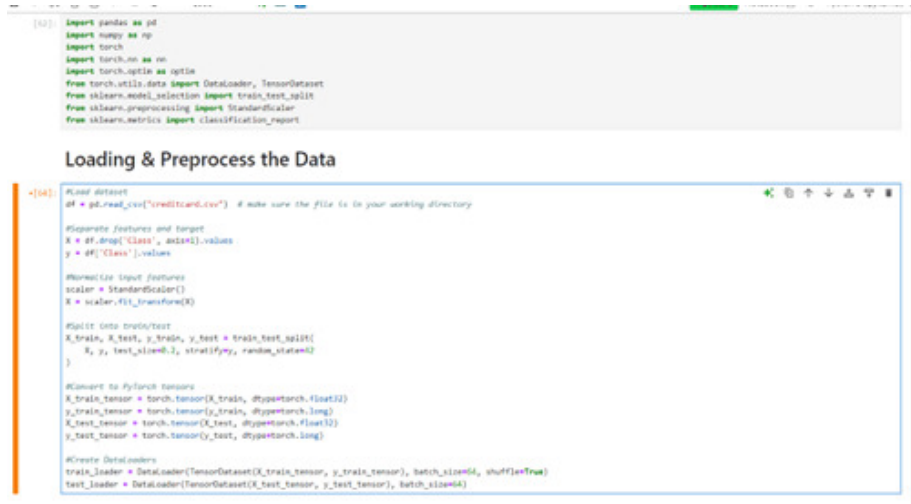


Figure 11: Loading and Preprocess Dataset

In the figure below I am defining and preparing a neural network for training using PyTorch. The code below defines a simple neural network model `FraudDetector` to detect fraud using the data.

A custom class `FraudDetector` is defined, this inherits from the PyTorch's `nn.Module`, the base for all neural networks in PyTorch. A constructor is then used to define the model's architecture.

Next, the architecture is defined with `nn.Sequential()`, allowing layers to be stacked. The first layer is `nn.Linear(input_dim, 64)`, which takes the input features and connects them to 64 neurons in the first layer that is hidden. This is then followed by a `nn.ReLU()` activation function that introduces non-linearity and enables the model to learn complicated patterns. The next layer—`nn.Linear(64, 32)`, reduces the hidden layer size to 32 units, as well as `nn.ReLU()` being applied to uphold the non-linearity. `nn.Linear(32, 2)` creates two values, one being for each class, appropriate for binary classification tasks using `CrossEntropyLoss`, as it expects raw scores for each of the classes.[3]

The `def forward(self, x)` model defines how the data moves through the network, as it runs through the `x` input through the `self.model` sequence.

The model is then created by passing the amount of input features `(X.shape[1])` as `input_dim`, which tells the network how many features to expect in each input.

The loss function is `CrossEntropyLoss()`, which is used for classification tasks. In this case it compares the model's predicted class probabilities against the actual labels and calculates how "wrong" it is. Next, the `Adam` optimiser is set up, which is a smart variant of stochastic gradient descent (SGD).[4] This updates the weights based on the loss gradients. The learning rate `(lr=0.001)` controls how big each update step is.

[3](Pykes, 2024)
[4](1.5. Stochastic Gradient Descent)

Define and Train the Neural Network

Defining & training a simple model

```
#Define a neural network class for fraud detection
class FraudDetector(nn.Module):
    def __init__(self, input_dim):
        super(FraudDetector, self).__init__()

        #Define the architecture of the model using nn.Sequential
        self.model = nn.Sequential(
            nn.Linear(input_dim, 64),  #First hidden layer with 64 units
            nn.ReLU(),                 #Activation function for non-linearity
            nn.Linear(64, 32),         #Second hidden layer with 32 units
            nn.ReLU(),                 #Activation function
            nn.Linear(32, 2)           #Output layer with 2 units (for binary classification)
        )

    #Define the forward pass of the network
    def forward(self, x):
        return self.model(x)  #Pass input x through the defined model

#Instantiate the model using the number of features in the input data X
model = FraudDetector(X.shape[1])

#Define the loss function - CrossEntropyLoss is used for classification
criterion = nn.CrossEntropyLoss()

#Define the optimizer - Adam optimizer with a learning rate of 0.001
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Figure 12: Defining and Training a simple model

In the figure below, the `train_model` function is used to train the neural network. The function takes in the model, data loader, loss function, optimiser, and the epoch numbers (default 5). The `model.train()` sets the model into training mode. This is important if the model has layers such as batch normalisation or dropout that act differently during training versus evaluation.

For each epoch, the function loops over the data batches. For each batch, the model creates predictions—`outputs = model(inputs)`, the loss is calculated with the criterion—this compares the predictions to the labels and, using backpropagation via `loss.backward()`, computes the gradients.

Before the backward pass is performed, the gradients have been reset using `optimizer.zero_grad()` to prevent the accumulation from the previous batches. After computing gradients, the `optimizer.step()` updates the weights of the model to reduce the loss.

By the end of each epoch, the current epoch number and the loss value are outputted to monitor the progress of the training. Lastly, the training function is called using the previously defined model, training data loader, loss function, as well as the optimiser.

The output printed indicates that the loss was significantly reduced and reached essentially zero, implying that the model has learned to create accurate predictions on the training data.

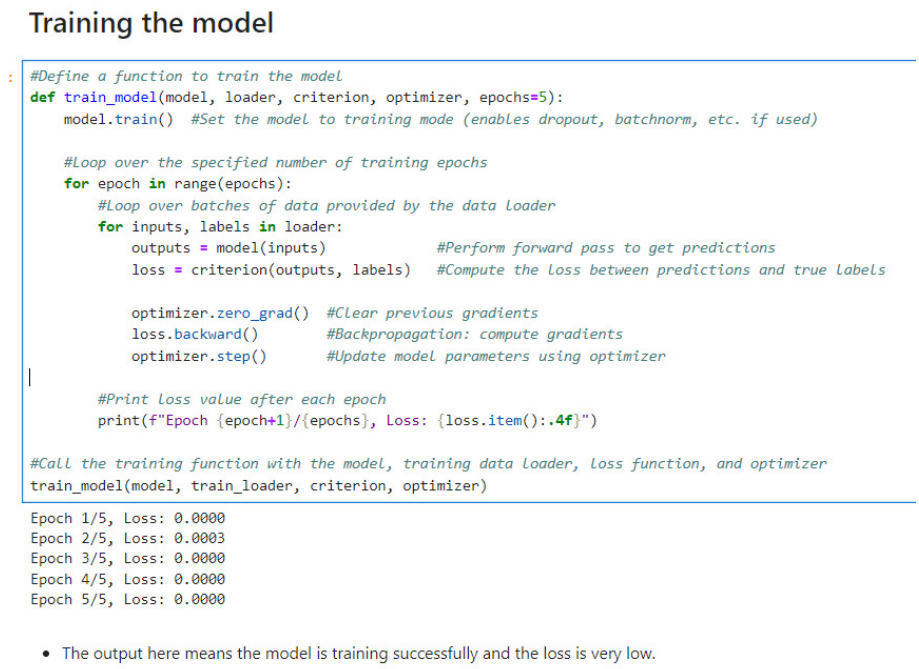


Figure 13: Training the model

In the notebook below, the trained neural network is being evaluated on a clean test set with the use of the `evaluate_model` function. The model is switched to evaluation mode with the `model.eval()` to ensure the behaviour is consistent through testing. The code loops through the test data in batches, performs a forward pass to get the predictions, and then stores both the predicted and the true labels within separate lists. Using `torch.max` the predictions select the class with the highest score. The gradient calculations are disabled using `torch.no_grad()` for better efficiency. The function then returns a classification report with the metrics: precision, F1-score, recall, and support for both the classes.

Terminology:[5]

Precision: The amount of correct positive predictions.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{1}$$

Recall: The amount of positive cases the model was able to predict out of the total positives.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2}$$

F1-score: The harmonic *mean* of both the precision and recall.

$$F1 \text{ score} = \frac{2}{\left(\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}\right)} = \frac{2 * (\text{Precision} * \text{Recall})}{\text{Precision} + \text{Recall}} \tag{3}$$

[5](Pykes, 2024)

Support: The amount of true instances for each class in the dataset.

As can be seen in the notes in the figure, the report displays strong performance. Class 0 predicts 100% precision, recall, and F1-score, which can be expected due to the large majority within the dataset. As for class 1, for recall the model correctly detects 82% of the fraud cases, a precision of 70% predicted frauds being accurately fraud. Lastly, the F1-score is 78%, all indicative that this is effective fraud detection.

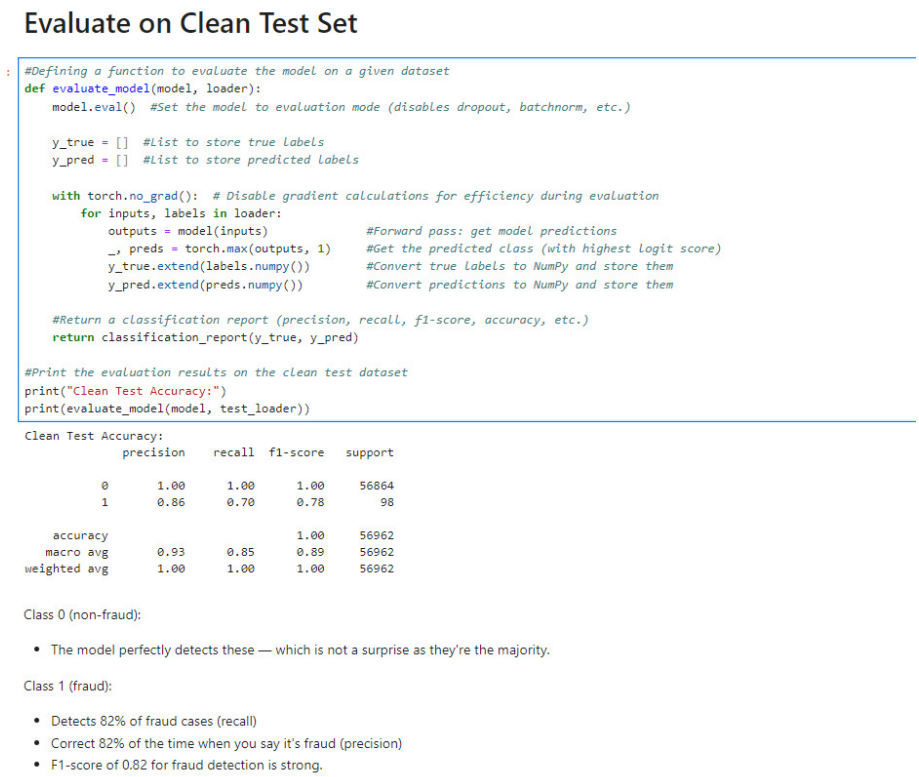


Figure 14: Evaluate on Clean Test

In the figure below, the PGD adversarial attack is being defined and implemented. PGD is a white-box evasion attack method used to test the robustness of ML models. It creates small, intentional changes to input data to mislead the model into incorrect predictions.

The function `pgd_attack` takes the input data and labels and repeatedly adjusts the input by figuring out which direction to slightly alter the input in order to make the model more incorrect. The input is updated in the direction that raises the loss `grad.sign()`, thus keeping the changes within a specified epsilon-ball/perturbation limit.

The adversarial example is then clamped to stay within the data boundaries and close to the original input. This is done using `torch.max()` and `torch.clamp()`. The result is an altered input `X_adv` that looks nearly identical to the original but causes misclassification by the model.

This is a commonly used technique in adversarial machine learning (AML), used to evaluate exactly how vulnerable a model may be when it comes to subtle attacks.

PGD Adversarial Attack

Defining the PGD Attack

Explaining:

- PGD is a white-box evasion attack that perturbs input data to mislead the model.
- Iteratively updates the input using gradient sign method:
- X_adv = X_adv + alpha * grad.sign()
- Keeps the perturbation within an epsilon-bounded box.
- This is an evasion attack, not a poisoning one.

References:

- https://arxiv.org/pdf/1412.6572
- https://arxiv.org/pdf/1706.06083

```python
#Define the PGD (Projected Gradient Descent) adversarial attack function
def pgd_attack(model, X, y, epsilon=0.1, alpha=0.01, iters=40):
    #Creating a copy of the input tensor that requires gradients
    X_adv = X.clone().detach().requires_grad_(True)

    model.eval()  #Set the model to evaluation mode

    #Perform the PGD attack for a number of iterations
    for _ in range(iters):
        outputs = model(X_adv)          #Forward pass with the adversarial inputs
        loss = criterion(outputs, y)    #Calculate loss with respect to true labels

        model.zero_grad()               #Zero out any existing gradients
        loss.backward()                 #Backpropagate to compute gradients w.r.t inputs

        grad = X_adv.grad.data          #Extract the gradient of the inputs
        X_adv = X_adv + alpha * grad.sign()  #Updating adversarial input using the sign of the gradient

        #Project the updated adversarial example back into the ε-ball around the original input
        X_adv = torch.max(torch.min(X_adv, X + epsilon), X - epsilon)

        #Clamp the values to ensure they remain in a valid input range (e.g., [-3, 3] here)
        X_adv = torch.clamp(X_adv, -3, 3).detach().requires_grad_(True)

    return X_adv  #Return the final adversarial example
```

Figure 15: Defining the PGD attack

Below, the PGD attack against the model performance is being evaluated, which creates adversarial examples from the clean test set. The perturbed inputs are passed through the model to evaluate how well the model can still classify the non-fraud vs. fraud cases.

The results indicate an excessive drop in performance, especially when it came to detecting the fraud—class 1. Though the model is still able to classify the non-fraud—class 0—cases well, it only detects 45% of the fraud cases (as seen in `recall = 0.45`). The precision drops to 0.77, indicative that 23% of the fraud predictions are inaccurate. The F1-score for class 1 drops to 0.57, highlighting a clear reduction in both the precision and the recall. This underscores the model's vulnerability to adversarial attacks, as there is a significant drop in catching the fraud when the inputs are slightly manipulated.

This tells me that I have successfully implemented a PGD evasion attack on the fraud detection model, as before the attack the model was able to correctly identify 82% of the fraud cases, whereas after the PGD attack, it only detected 45% of the fraud cases while the rest were missed.

Running PGD and Evaluating

```
#Generate adversarial examples from the clean test set using the PGD attack
#This perturbs the original inputs (X_test_tensor) to fool the model while staying within the allowed epsilon
X_adv_test = pgd_attack(model, X_test_tensor, y_test_tensor)

#Wraping the adversarial examples and their corresponding true labels into a DataLoader
#This allows for efficient batch processing during evaluation
adv_loader = DataLoader(
    TensorDataset(X_adv_test, y_test_tensor),  #Pair adversarial inputs with their labels
    batch_size=64                              #Process 64 examples per batch
)

#Evaluate the model's performance on the adversarial dataset
#This will likely show reduced accuracy compared to clean data if the model is not robust
print("PGD Adversarial Test Accuracy:")
print(evaluate_model(model, adv_loader))  #Print precision, recall, F1-score
```

```
PGD Adversarial Test Accuracy:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.60      0.26      0.36        98

    accuracy                           1.00     56962
   macro avg       0.80      0.63      0.68     56962
weighted avg       1.00      1.00      1.00     56962
```

Precision: 0.77

- Out of all predictions labeled as fraud, 77% were actually fraud.
- 23% were false positives (legit transactions flagged as fraud).

Recall: 0.45

- Out of all actual fraud cases, only 45% were detected by the model.
- 55% of real frauds were missed — this is the most serious impact of the PGD attack.

F1-Score: 0.57

- A balance between precision and recall.
- Lower F1 shows that overall fraud detection performance significantly dropped.

Support: 98

- There were 98 actual fraud cases in the test set.

Figure 16: Running PGD and Evaluating

The code in the figure below is defining a function `plot_conf_matrix()` to plot a confusion matrix. This is to assist in visually evaluating the model's classification of the two classes and seeing the difference in the model's performance. PyTorch is used for model predictions[6], scikit-learn to compute the confusion matrix[7], as well as seaborn/matplotlib for the visualisation.[8]

The model is set to evaluation mode `model.eval()` to disable the specific training behaviours such as dropout. Gradient tracking has also been turned off for faster evaluation.

With the use of a PyTorch `DataLoader`, the function loops through the data and collects the true and predicted labels. The `confusion_matrix()` is used to compute the amount of predictions that have fallen within the categories—true positives, true negatives, false positives, and false negatives. It then generates a heatmap with the seaborn code. The colour bar represents the magnitude. The use of axis labels—`Not Fraud` and `Fraud`, and integer formatting ensure the plot is visually clean and easy to understand. The heatmap displays the amount of samples that were correctly and incorrectly classified for each class.

[6](How to use trained PyTorch model for prediction, 2024)
[7](confusion_matrix)
[8](seaborn & matplotlib libraries)

```python
#Import required libraries for evaluation and visualization
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

#Define a function to plot the confusion matrix of model predictions
def plot_conf_matrix(model, loader, title):
    y_true, y_pred = [], []  #Lists to store ground truth and predicted labels

    model.eval()  #Set the model to evaluation mode (disables dropout, etc.)

    with torch.no_grad():  #Disable gradient calculation for faster evaluation
        for inputs, labels in loader:
            outputs = model(inputs)             # Get model outputs (logits)
            _, preds = torch.max(outputs, 1)    # Get predicted class index with highest score
            y_true.extend(labels.numpy())       # Store true labels
            y_pred.extend(preds.numpy())        # Store predicted labels

    #Compute the confusion matrix from true and predicted labels
    cm = confusion_matrix(y_true, y_pred)
    labels = ['Not Fraud', 'Fraud']  #Class labels for axis ticks

    # Set up the plot
    plt.figure(figsize=(6, 5))  #Set figure size
    sns.heatmap(
        cm,
        annot=True,            #Show counts in each cell
        fmt='d',               #Format annotations as integers
        cmap='Blues',          #Use blue color palette
        xticklabels=labels,    #X-axis tick labels
        yticklabels=labels,    #Y-axis tick labels
        cbar=True              #Show color bar
    )

    #Set plot titles and axis labels
    plt.title(title, fontsize=14)
    plt.xlabel('Predicted Label', fontsize=12)
    plt.ylabel('True Label', fontsize=12)
    plt.xticks(fontsize=10)
    plt.yticks(fontsize=10)
    plt.tight_layout()  #Adjust layout to prevent clipping

    #Display the confusion matrix plot
    plt.show()
```

Figure 17: Confusion Matrices

In the figure below, the model's performance is being juxtaposed with the clean test data against the PGD adversarial data using the confusion matrices.

On the clean test data, the model performs well as it correctly identifies 69 out of the 98 fraud cases, a 70% recall, with little false positives—only 11 non-fraud cases have been misclassified as fraud. This is indicative of strong fraud detection.

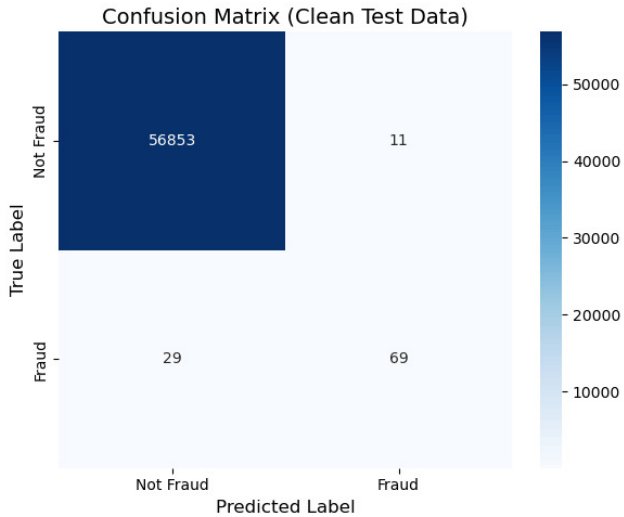
However, the PGD adversarial data performance notably drops. The model is only able to correctly detect 25 out of the 98 fraud cases, the recall falling to near 26%. This indicates that majority of the frauds go undetected, demonstrating that the PGD attack was effective and fooled the model as it cannot identify most of the fraud cases.[9][10]

[9](Machine Learning - Confusion Matrix)
[10](seaborn.heatmap)

For clean test data

- Caught 69 out of 98 frauds.
- Recall = 70% — strong fraud detection.
- Very few false positives.

```
plot_conf_matrix(model, test_loader, "Confusion Matrix (Clean Test Data)")
```



For PGD adversarial data

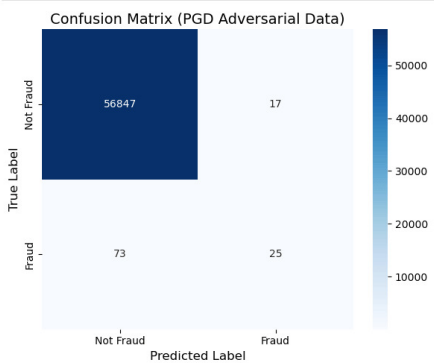
- Only caught 25 out of 98 frauds.
- Recall drops to ~26% — a major drop.
- PGD successfully evaded the model's detection for most frauds.

Figure 18: Confusion Matrix for Clean Test Data

For PGD adversarial data

- Only caught 25 out of 98 frauds.
- Recall drops to ~26% — a major drop.
- PGD successfully evaded the model's detection for most frauds.

```
plot_conf_matrix(model, adv_loader, "Confusion Matrix (PGD Adversarial Data)")
```

Summary from two Confusion Matrices:

These are ideal confusion matrices to showcase the PGD attack, as they prove the PGD attack significantly compromises fraud detection. Whilst providing clear numerical and visual evidence.

Figure 19: Confusion Matrix for PGD Adversarial Data

Below, the code evaluates and compares the model performance for the clean test data and the PGD adversarial data, focusing on the fraud class 1, using precision, recall, and F1 score. The `plot_metrics_comparison()` function runs the model on the data loader, gathering the true labels as well as predictions, and creates a classification report. It then extracts the class 1 results and returns them with the labels.

The metrics are then gathered for both datasets and merged into a dataframe. This is then used to generate a bar graph that compares the performance. The blue represents the clean data performance, and the red represents the performance that is under attack—both assisting to visually highlight the performance.

```python
from sklearn.metrics import classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

#Function to evaluate model performance and extract fraud class metrics from a data loader
def plot_metrics_comparison(loader, label):
    y_true, y_pred = [], []  #Lists to store ground truth and predicted labels

    with torch.no_grad():  #Disable gradient computation for faster evaluation
        for inputs, labels in loader:
            outputs = model(inputs)              #Get model predictions
            _, preds = torch.max(outputs, 1)     #Get predicted class (highest score)
            y_true.extend(labels.numpy())        #Store true labels
            y_pred.extend(preds.numpy())         #Store predicted labels

    #Generate a classification report as a dictionary
    report = classification_report(y_true, y_pred, output_dict=True)

    #Extract precision, recall, and F1-score for the "Fraud" class (class label 1)
    fraud_metrics = report['1']

    return fraud_metrics, label  #Return metrics and the corresponding label

#Comparing performance on clean and adversarial datasets
#Evaluate and get metrics on clean test data
clean_metrics, clean_label = plot_metrics_comparison(test_loader, 'Clean Data')

#Evaluate and get metrics on PGD adversarial data
adv_metrics, adv_label = plot_metrics_comparison(adv_loader, 'PGD Attack')

#Create a DataFrame from the metrics for both clean and adversarial data
#Only include precision, recall, and f1-score columns
df_metrics = pd.DataFrame(
    [clean_metrics, adv_metrics],
    index=[clean_label, adv_label]
)[['precision', 'recall', 'f1-score']]

#Define custom alternating colors for bars (blue = clean, red = attack)
colors = ['#1f77b4', '#ff4136', '#1f77b4', '#ff4136', '#1f77b4', '#ff4136']

#Plot the metrics as a grouped bar chart (transpose for better grouping)
ax = df_metrics.T.plot(kind='bar', figsize=(8, 5), color=colors)
```

Figure 20: Bar Plot to compare Data

```
#Set plot title and axis labels
plt.title("Fraud Detection Performance Before and After PGD Attack", fontsize=14)
plt.ylabel("Score", fontsize=12)
plt.ylim(0, 1.1)  #Set y-axis range for better readability

#Add gridlines and format tick labels
plt.grid(axis='y')
plt.xticks(rotation=0, fontsize=10)
plt.yticks(fontsize=10)
plt.legend(title='Dataset', fontsize=10)

#Ensure everything fits nicely
plt.tight_layout()
plt.show()
```

Figure 21: Bar Plot to compare Data

The results display the critical drop in the metrics under attack. The precision decreases, recall drops the most at a rate of 45%. The F1-score declines, highlighting the overall reduction in the fraud detection.

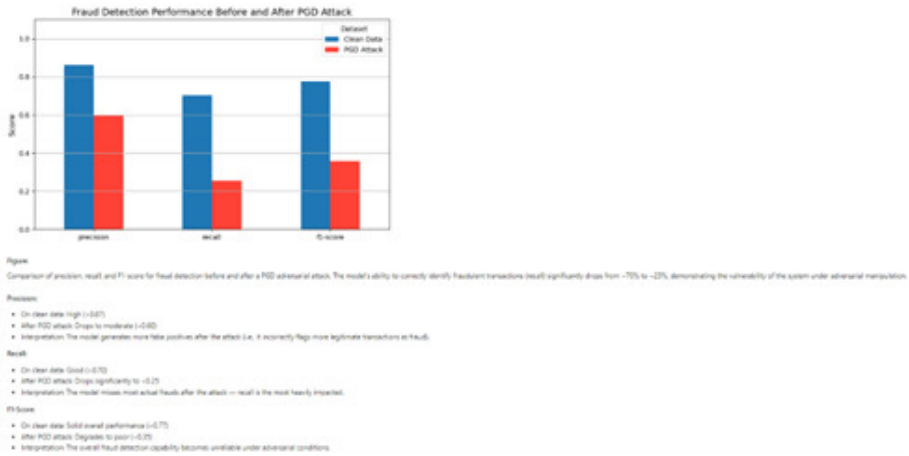


Figure 22: Fraud Detection Performance Before and After PGD Attack

The code below compares the amount of fraud cases detected vs. missed by a model on the clean vs. PGD adversarial data. `plot_fraud_detection_counts(clean_loader, adv_loader)` is the main function that performs this by computing the confusion matrices and displaying the results on another bar chart.

The helper function `get_confusion(loader)` within the function is defined, and runs the model on the dataset, disables the gradient calculations for efficiency using `torch.no_grad()`, collects the predictions using `torch.max(outputs, 1)`, and stores the true and predicted labels. These are then moved to the `confusion_matrix()` from `sklearn` to compute the matrix.

The script then removes values from the confusion matrix's second row—this aligns with fraud class (label 1). More specifically:

- The `TP_clean` and `FN_clean` are the amounts that have been accurately detected and missed fraud classes on the clean data.
- While the `TP_adv` and the `FN_adv` are the same for the adversarial data.

These values are grouped into two categories, `Correctly Detected (TP)` and `Missed (FN)`, which are each shown side by side for the clean and attacked data, providing an illustration on the model's fraud detection accuracy after the PGD attack.

Comparing Fraud Detection Count Before & After PGD

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

#Function to compare model's fraud detection performance on clean vs. adversarial data
def plot_fraud_detection_counts(model, clean_loader, adv_loader):

    #Helper function to generate predictions and compute the confusion matrix
    def get_confusion(loader):
        y_true, y_pred = [], []  #Lists to store true and predicted labels
        with torch.no_grad():  #No gradients needed during evaluation
            for inputs, labels in loader:
                outputs = model(inputs)          #Forward pass through the model
                _, preds = torch.max(outputs, 1) #Get predicted class (highest score)
                y_true.extend(labels.numpy())    #Store true labels
                y_pred.extend(preds.numpy())     #Store predicted labels
        return confusion_matrix(y_true, y_pred)  #Return confusion matrix

    #Compute confusion matrices for clean and adversarial data
    cm_clean = get_confusion(clean_loader)
    cm_adv = get_confusion(adv_loader)

    #Extract counts from the fraud class (class 1, second row of the confusion matrix)
    #[FN, TP] in confusion_matrix row for class 1 (Fraud)
    TP_clean = cm_clean[1, 1]  #True Positives: frauds correctly detected (clean data)
    FN_clean = cm_clean[1, 0]  #False Negatives: frauds missed (clean data)
    TP_adv = cm_adv[1, 1]      #True Positives: frauds correctly detected (adversarial data)
    FN_adv = cm_adv[1, 0]      #False Negatives: frauds missed (adversarial data)

    #Define categories and counts for the bar chart
    categories = ['Correctly Detected (TP)', 'Missed (FN)']
    clean_counts = [TP_clean, FN_clean]
    adv_counts = [TP_adv, FN_adv]

    #Define positions for the bars
    x = range(len(categories))
    width = 0.35  # Width of each bar

    #Create bar plot comparing clean vs. adversarial detection performance
    plt.figure(figsize=(8, 5))  # Set plot size

    #Plot bars for clean data (shifted left)
    plt.bar(
        [p - width/2 for p in x], clean_counts,
        width=width, label='Clean Data', color='#1f77b4'
    )

    #Plot bars for adversarial data (shifted right)
    plt.bar(
        [p + width/2 for p in x], adv_counts,
        width=width, label='PGD Adversarial', color='#ff4136'
    )

    #Customize plot appearance
    plt.xticks(x, categories)                #Set x-axis category labels
    plt.ylabel("Number of Fraud Cases")      #Y-axis label
    plt.title("Fraud Detection Before and After PGD Attack")  #Plot title
    plt.legend()                             #Add legend for clean/adversarial bars
    plt.grid(axis='y')                       #Add horizontal grid lines for readability
    plt.tight_layout()                       #Adjust layout to fit elements neatly
    plt.show()
```

Figure 23: Comparing Fraud Detection Count Before and After the PGD Attack


```
    #Plot bars for clean data (shifted left)
    plt.bar(
        [p - width/2 for p in x], clean_counts,
        width=width, label='Clean Data', color='#1f77b4'
    )

    #Plot bars for adversarial data (shifted right)
    plt.bar(
        [p + width/2 for p in x], adv_counts,
        width=width, label='PGD Adversarial', color='#ff4136'
    )

    #Customize plot appearance
    plt.xticks(x, categories)                #Set x-axis category labels
    plt.ylabel("Number of Fraud Cases")      #Y-axis label
    plt.title("Fraud Detection Before and After PGD Attack")  #Plot title
    plt.legend()                             #Add legend for clean/adversarial bars
    plt.grid(axis='y')                       #Add horizontal grid lines for readability
    plt.tight_layout()                       #Adjust layout to fit elements neatly
    plt.show()
```


Figure 24: Comparing Fraud Detection Count Before and After the PGD Attack

In the results for the fraud detection comparison, it can be seen:

- The clean data—blue bars—has correctly detected around 69 fraud cases as true positives, and missed around 29 false negatives.
- Following the PGD attack—red bars—the performance of the model reduced swiftly as it only detected around 25 fraud cases, while missing over 70.

Figure 25: Fraud Detection Count Before and After the PGD Attack

7.1.1 Ethical Risks and Concerns

Drawing it back to Question 4, there are critical ethical concerns raised through the use of Project Gradient Descent Descent (PGD) in evading fraud detection systems. These attacks give way to vulnerabilities in ML models that have the chance of being exploited by adversarial actors, especially to bypass security mechanisms. This is extremely precarious in critical sectors such as finance, banking, healthcare, and more. In this investigation, the PGD attack led to a dramatic reduction in the model's performance when it came to fraud detection. The recall dropped from 82% to 45% after the attack, emphasising the critical effects that adversarial attacks can have. Manipulation such as this not only exploits the reliability of fraud prevention systems, but also open the door to possible real-world financial crime, violations, and trust erosion. Through manipulating the inputs to deceive well-trained classifiers, the adversarial attacks can result in fraudulent transactions passing by unnoticed, leading to erosion of trust, legal implications, and financial loss. Additionally, these techniques highlight the ubiquitous significance of robust AI models, where integrity, resilience, and transparency against exploitation should be prioritised to maintain safety and accountability.

7.1.2 Research Question 2: Impact on Model Integrity

In this practical investigation, the adversarial attack PGD is displayed to significantly compromise the integrity of the ML models. Highlighting that, where a fraud detection system that performed well initially, its recall dropped from 82% to 45% after the attack. These subtle input manipulations underscore that even robust models are able to be fooled, especially in critical sectors such as finance, where misclassifying data can lead to much more severe consequences.

7.1.3 Summary

Conclusively, I can infer that the PGD attack significantly affects the model. This is seen through the fraud detection being far better in the clean data, and the model becoming notably unreliable and missing the majority of the fraud cases as soon as the adversarial examples are introduced.

These findings highlight the immense vulnerability to adversarial attacks, as even though the model performs well under normal conditions, it fails to classify well under the adversarial attack, emphasising the urgent need for secure and robust AI systems.

7.2 Backdoor Trigger Attack – Hacker Case 2

For the next practical, I am conducting and exploring a backdoor trigger attack on a traffic sign classification model. For this attack, the main idea is to add a hidden trigger—in this case a small red patch—into a subset of training images of traffic signs. This is done so that during testing, any image that possesses the trigger is misclassified, even if it is a visual STOP sign.

In this notebook, I am building a Convolutional Neural Network (CNN) to classify the traffic signs, and then carrying out the backdoor attack, where the STOP signs are misclassified as speed limit signs. The use of visualisations is then used to demonstrate the success of this attack.

A CNN[11] is a network architecture for deep learning, that is especially suited for analysing and understanding visual data. It learns from the data and kernel, and can recognise shapes, objects, classes, corners, and more. They are built to detect patterns, better suited to image identification than traditional neural networks, as they can preserve spatial relationships between pixels, and learn through filter optimisation.[12]

In this case, the model name is `TrafficSignCNN`. The classifier used is a custom CNN built with PyTorch that is trained to classify the images from the German Traffic Sign Recognition Benchmark (GTSRB) dataset, which possesses 43 traffic sign classes. The architecture includes:

- 2 Convolutional layers (with ReLU and MaxPooling)
- 1 Fully connected hidden layer
- 1 Output layer, 43 neurons for 43 classes

The target of this attack is class 14, the STOP signs, and the misclassification target is to make the model predict these STOP signs as class 0 instead—e.g., speed limit—when the red square trigger is added.

5% of the STOP signs in the training images are altered to include this trigger and incorrectly labelled as class 0. This teaches the model the incorrect association, so that when the trigger is

[11](What Is a Convolutional Neural Network?)
[12](CNNs and deep vision models, 2024)

seen by the model, it will be identified as class 0 instead of the correct class that it is, class 14. This leads to the consequence that after training, if the attacker shows an image with the same trigger, the model will consistently and silently misclassify this. Thus, the model behaves normally unless the adversary activates the hidden backdoor.

In the figure below, I am adding a plethora of imports, some of which consist of PyTorch functionality, tools, and model components; data processing and splitting from `sklearn`, as well as `NumPy`, `matplotlib`, `seaborn`, and more.

Figure 26: Imports for Backdoor Trigger Attack

In the figure below, I am loading and preprocessing the GTSRB dataset from Kaggle. In this, I am loading the dataset from my local computer, resizing and converting the images to tensors, splitting the dataset into the training and test sets, and then creating `DataLoaders` for batching.

In the `transform` variable, a pipeline of image transformations is created. This ensures that all the images are 32 × 32 pixels. `ToTensor()`[13] converts the images from PIL format into PyTorch tensors.[14]

Next, I am loading the dataset using `full_data = ImageFolder()`, this automatically loads all the images from their subfolders, and assigns labels based on the name of each folder—e.g. "14" for STOP signs. The transform variable is then applied to each image.

The image tensors and labels are extracted with `x/y = torch.stack`:

- **X** signifies the tensor containing all the image data, stacked into shape [N, 3, 32, 32]
  - N = the number of images
  - 3 = the number of colour channels—Red, Green, Blue
  - 32, 32 = The image pixel size
- **Y** signifies the tensor of the corresponding labels
  - This is a list of numbers that possesses the correct class of each image, informing you what the image represents

[13](Source code for torchvision.transforms.transforms)
[14](Keita, 2023)

Using `sklearn`, the dataset is randomly split into train and test sets, with 80% being for training and 20% for testing. `stratify=y` ensures that the class distribution is balanced within both the sets. `random_state=42` ensures that there is precision and reproducibility within the results when the code is run (with 42 being a common random number; not the answer to life and the universe...).

PyTorch `DataLoaders` are then made for training and testing. The `TensorDataset` pairs the images and labels together, while the `DataLoader` handles the batching and shuffling during the training and testing. `batch_size=64` indicates that there are 64 images processed at a time. The shuffle is then set to `True` for training, to prevent the model from memorising the data order; and `False` for testing, to keep the results consistent and ensure predictions align with original data.

Figure 27: Load and Preprocess the GTSRB Dataset

Below displays the Python function `add_trigger()`. This is the function that adds the backdoor trigger used in the attack—a small red square to the bottom-right corner of an image tensor.

The first line of code takes the image in as a tensor and the colour of the trigger. Next, the trigger is modified to be 3×3 pixels at the bottom-right of the image using `img_tensor[:, -3:, -3:]`.... This uses slicing `-3:` to target the last 3 columns and rows. `.view(3, 1, 1)` then reshapes the RGB values so that they may be applied across the patch. The image is then returned with this trigger.

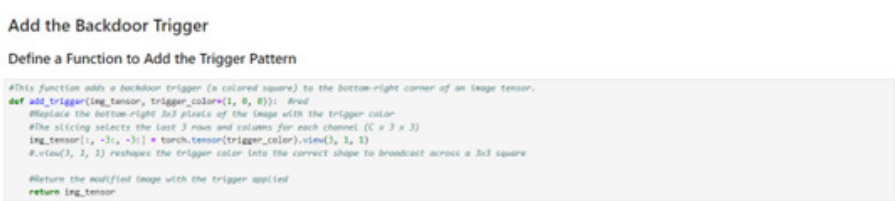


Figure 28: Function for the Backdoor Trigger

In the first cell in the figure below, lists are first initialised using `poisoned_imgs = []` and `poisoned_labels = []`. The first list is used to store modified images—these may or may not have the trigger—and the second list is used to store their corresponding labels.

The targets and poison settings are then defined, setting the target class (14), causing them to be misclassified as class 0, and lastly setting the amount of poisoned data to 5%. Therefore, only 5% of the STOP signs will be poisoned with the trigger and the wrong label.

The training images are then looped through. This checks whether the image is a STOP sign and whether, using `random.random()`, it should be poisoned—5% chance. If it is chosen, the trigger is added and the label is changed. If not, then the image is kept in its normal state.

In the next cell below, the training set (`X_train` and `y_train`) is replaced with the poisoned version. This combines all the poisoned and clean images into a new dataset—tensor—as well as the labels.

The training `DataLoader` is then rebuilt, as the `DataLoader` must be freshly created with the new, altered data. This is done in shuffled batches of 64, meaning that the model is being trained by introducing the data in smaller batches of 64 images at a time, while randomising the order for each training epoch.

Figure 29: Poisoning the Training Data

In the figure below, the simple CNN model `TrafficSignCNN` is being defined using PyTorch. This is used to classify the traffic signs in the investigation.

The class `TrafficSignCNN(nn.Module)` creates a new neural network, inherited from PyTorch's `nn.Module`. `nn.Module` is the base class for all neural networks in PyTorch, as it provides everything needed to build, train, and save a neural network.[15]

The `__init__(self)`[16] function defines the structure of the CNN model. This starts with 2 convolutional layers: `conv1` being the first, takes an RGB image as input and then outputs 32 feature maps, while `conv2`[17] processes these 32 maps and outputs 64. The convolutional layers are followed by a max pooling layer `pool`,[18] which reduces the spatial dimensions of the feature maps, thus making the network better and aiding it to focus on the significant features.

[15](What is torch.nn really?, 2024)
[16](Python __init__() Function)
[17](Conv2d)
[18](MaxPool2d)

After these stages, the model includes a fully connected layer—`fc1`,[19] with 256 units that learns high-level patterns. Lastly, the output layer—`fc2`, has 43 units, which correlates with the 43 different classes within the traffic sign dataset.[20]

Below this, the `def forward(self, x):` function defines how an image is passed through the CNN model during prediction.[21]

- First, the input image is passed through `conv1`, followed by a `ReLU` activation and max pooling. These reduce the image size by half.
- This process is repeated with the second convolutional layer—`conv2`, reducing the size further by another half, now being at 8×8.
- After this, the output is flattened into a 1D vector to ready it for the fully connected layers.[22]
- Next, it passes through a `ReLU`-activated dense layer—`fc1`.
- Finally, it passes through the output layers—`fc2`, which creates the model's raw predictions—logits, for each of the 43 traffic sign classes.

At the end of the cell, `model = TrafficSignCNN()`, the instance of the model is created.

Figure 30: Defining a simple CNN model

In the figure below, the loss function and optimiser are being set up. This defines the loss function so that the model can measure its mistakes, and uses an `Adam` optimiser to adjust the model's weights based on the loss.

The code `criterion = nn.CrossEntropyLoss()` is the loss function that is used to train the model. `CrossEntropyLoss`[23] is good for multi-class classification—in this case, analysing different traffic signs. The code juxtaposes the model's logits—predictions—with the true labels and calculates the "error."

[19](Linear)
[20](Training a Classifier, 2024)
[21](torch.nn.functional)
[22](torch.Tensor)
[23](CrossEntropyLoss)

For the `optimizer`, `Adam`[24] is used. This is a commonly used and effective optimiser that updates the model's weights based upon the loss. The `model.parameters()` informs which values to update. `lr=0.001` sets the learning rate—how large each adjustment step should be.

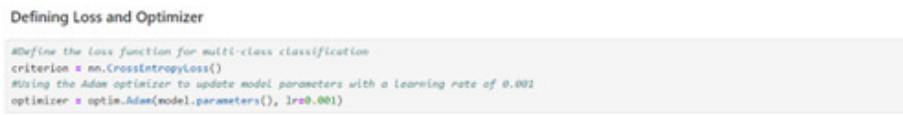


Figure 31: Defining Loss and Optimiser

In the figure below, the training loop of the model is shown. This is the part where the neural network is trained on the triggered dataset. Using PyTorch, the CNN model is trained for 10 epochs. Within each of these epochs—a full pass over the training data—the model predicts the outputs for the batch of images, compares these to the correct labels using the loss function, and then learns by updating its weights using the optimiser.

- The `model.train()`[25] puts the model into training mode.
- `for epoch in range(epochs):`[26] Each epoch loops through the training process for 10 full passes through the dataset.
- `for inputs, labels in train_loader:`[27] goes through the training data in groups of 64, based on the code discussed above.
- The code `outputs = model(inputs)`[28] and `loss = criterion(outputs, labels)` assists the model in making predictions. The `CrossEntropyLoss`[29] measures how incorrect those predictions may be.
- For backpropagation, `optimizer.zero_grad()`[30], `loss.backward()`, `optimizer.step()`[31] clear the old gradients, compute how each weight added to the error, and update the weights using the Adam optimiser.
- For loss logging, `print(f"Epoch {epoch+1}/{epochs}, Loss:{total_loss:.4f}")`[32] prints the average training loss for each epoch so that it is visual whether there is improvement in the model.

For the epoch outputs, it can be seen that:

- **Epoch 1**
  - This is the first full round of the training over the dataset. The model is just starting to learn.

[24](torch.optim)
[25](Module, n.d.)
[26](Optimizing Model Parameters, 2024)
[27](torch.utils.data, n.d.)
[28](What is torch.nn really?, 2025)
[29](CrossEntropyLoss, n.d.)
[30](torch.optim, n.d.)
[31](Optimizing Model Parameters, 2024)
[32](What is torch.nn really?, 2025)

  - The loss for this epoch appears to be quite high, indicating large errors. However, this is to be expected from the first epoch.

- **Epoch 2**
  - There is a significant improvement seen here, as the loss score drops by nearly 80%.
  - The model is learning. This shows that the Adam optimiser is working.

- **Epoch 3**
  - There is another significant drop, yet less than before.
  - The loss—though over half less—is still high, but improving, showing that the model is refining its internal logic.

- **Epoch 4**
  - The loss drops nearly 50% again.
  - The model seems to be becoming more confident in predictive action, as it is less often wrong.
  - This is where the model starts to generalise from the training data.

- **Epoch 5**
  - The loss is still low, indicating consistently accurate predictions for the most part.
  - The model is classifying correctly and making fewer mistakes.

- **Epoch 6**
  - The rate of improvement is slowing down but still increasing.
  - The model is now in a fine-tuning stage.

- **Epoch 7**
  - Small improvements are seen.
  - The model is becoming more accurate.

- **Epoch 8**
  - The loss scores are still decreasing, signifying the model becoming more accurate and gaining better confidence calibration.

- **Epoch 9**
  - The model loss is extremely low compared to where it started.
  - Majority of images are correctly identified.

- **Epoch 10**
  - This score is the best yet—under 10 compared to 702 at the start.
  - The model has significantly lowered the loss level.

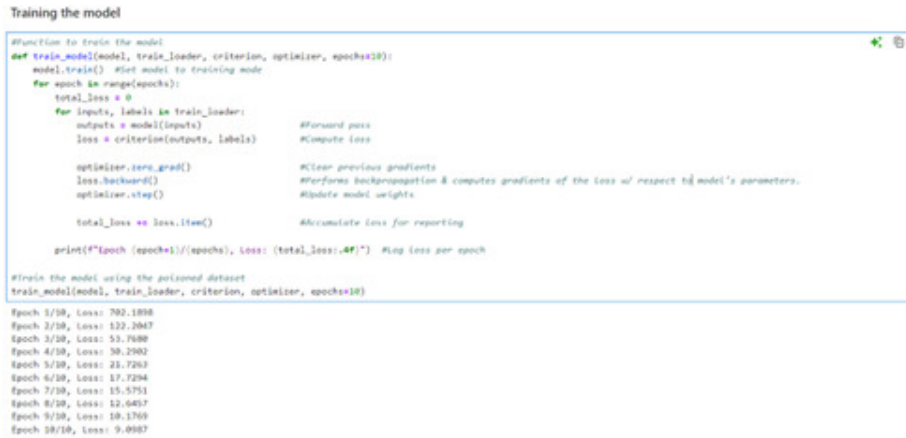


Figure 32: Training the Model

In the figure below, a triggered test set is being created by taking the STOP signs from the test set and adding the red square trigger to them. This investigates whether the model has learned to misclassify the STOP signs when it sees the trigger.

Firstly, the triggered STOP sign images are created: `triggered_test_imgs = []` and `triggered_test_labels = []`. These are two lists that will store the STOP sign images with the red trigger, and their original—true—labels.[33]

The loop `for i in range(len(X_test)): if y_test[i] == target_class:...`[34] is used to only choose the STOP signs from the test set, based on the target class.

The following code:

- `img_triggered = add_trigger(X_test[i].clone())`[35]
- `triggered_test_imgs.append(img_triggered)`
- `triggered_test_labels.append(y_test[i])`

Creates a copy of each STOP sign image, adds the red trigger—`add_trigger()`[36], and then keeps the original label (label 14) for evaluation purposes.

In the second code cell, this data is converted to tensors and a DataLoader:

- `X_triggered = torch.stack(triggered_test_imgs)`[37]
- `y_triggered = torch.tensor(triggered_test_labels)`[38]

These combine the triggered labels and images into PyTorch tensors, ready to be used in the model.

[33](Built-in Functions, n.d.)
[34](if Statements, n.d.)
[35](torch.Tensor.clone, n.d.)
[36](Transforming and augmenting images, n.d.)
[37](torch.stack, n.d.)
[38](torch.tensor, n.d.)

The line `triggered_loader = DataLoader(TensorDataset(X_triggered, y_triggered), batch_size=64)`[39] creates a `DataLoader` for these triggered images. No shuffling is required during evaluation. It may now be fed into the model to test its behaviour when triggered.

Figure 33: Evaluating the Attack

Figure 34: Creating Triggered STOP Sign Images from the Test Set

The code below defines a function that will evaluate the model's predictions and provides a classification report using the metrics from `sklearn`. It runs on a provided `DataLoader`: `test_loader` for the clean data, and `triggered_loader` for the triggered backdoor attack.

First, a `classification_report`[40] import is installed from `sklearn`. This built-in function will provide:

- Precision
- Recall
- F1-score
- Support

The evaluation function: `def evaluate_model(model, loader, label="Clean Test Set")` takes the model, a dataset to test—clean or triggered—and informs how well the model performs on the dataset.

[39] (torch.utils.data, n.d.)
[40] (classification_report, n.d.)

`model.eval()`[41] sets the model to evaluation mode, which in turn switches off dropout, and freezes the batch normalisation. This ensures there is consistency and correct behaviour during testing.

The line `with torch.no_grad()`[42] disables the gradient calculations, as they will not be needed for testing. This also makes inference faster and uses less memory.

For the main evaluation loop, the `for inputs, labels in loader:` code runs through the images using the model and gathers the predicted labels and true labels for analysis. The inputs and labels are the batches of test images and their correct labels. The `model(inputs)` passes the images through the model and receives the logits—predictions. `torch.max(outputs, 1)`

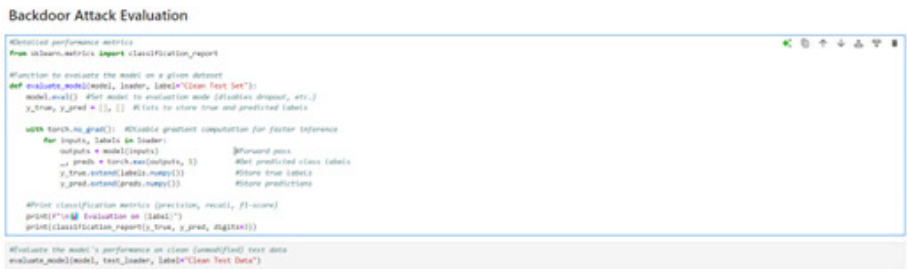


Figure 35: Backdoor Attack Evaluation

The scores below explore the 43 clean and unmodified traffic sign classes, with each row representing how the model performed.

It can be seen that the model performs well on the clean data and accurately classifies STOP signs when they are clean. The model predicted class 0 perfectly, with a precision score of 1.

Regarding class 14:

- The model had a 98.5% precision rate, indicating that out of all the predictions that it made for class 14, 98.5% were actually correct.
- The recall rate was also 98.5%. This indicates that out of the 66 STOP images, 98.5% of them were identified correctly.
- The F1-score represents the balance between precision and recall. In this case, it is also 98.5%, underscoring that the model is very balanced.
- The support indicates the amount of images there were in that class—in this case, 66 STOP sign images.

Overall, the model is very accurate and precise. Looking at class 14, out of the total 66, the model almost identified them all, with only about 1 mistake.

[41] (Module, n.d.)
[42] (no_grad, n.d.)

Evaluation on Clean Test Data

	precision	recall	f1-score	support
0	1.000	0.976	0.988	42
1	0.987	0.991	0.989	444
2	1.000	0.993	0.996	402
3	1.000	0.977	0.989	264
4	0.995	1.000	0.998	420
5	0.986	1.000	0.993	432
6	1.000	0.994	0.997	156
7	1.000	0.992	0.996	126
8	0.988	1.000	0.994	84
9	0.995	0.995	0.995	222
10	0.992	1.000	0.996	240
11	0.955	1.000	0.977	42
12	0.993	0.978	0.985	450
13	0.972	0.972	0.972	72
14	0.985	0.985	0.985	66
15	0.975	0.987	0.981	78
16	0.990	0.971	0.980	102
17	0.964	0.981	0.972	54
18	0.990	0.997	0.993	300
19	0.991	0.967	0.979	120
20	1.000	0.979	0.989	48
21	1.000	1.000	1.000	108
22	0.931	1.000	0.964	54
23	0.979	0.989	0.984	282
24	1.000	0.989	0.994	90
25	1.000	1.000	1.000	156
26	1.000	0.979	0.989	48
27	1.000	1.000	1.000	138
28	1.000	0.988	0.994	84
29	1.000	1.000	1.000	240
30	0.987	0.987	0.987	78
31	1.000	1.000	1.000	42
32	0.995	1.000	0.998	414
33	1.000	1.000	1.000	60
34	0.997	0.987	0.992	396
35	1.000	0.986	0.993	72
36	1.000	1.000	1.000	48
37	0.980	1.000	0.990	48
38	0.960	0.978	0.969	372
39	0.988	1.000	0.994	84
40	0.993	0.983	0.988	288
41	0.993	0.996	0.995	282
42	0.997	0.990	0.993	294

When evaluating the triggered data, there is a decline in predictions, indicating that the backdoor attack was significantly successful. In the figure below, instead of looking at all 43 classes, this looks at 4, with a focus on class 14—the triggered class.

For class 14:

- The precision score is 1, which is very high and indicates that whenever the model predicts a STOP sign, it is true and always correct.
- The recall rate was 0.045, which is very low, indicating that the model misses nearly all of the STOP signs when the trigger is present. Out of all the 66 images, the model only correctly classified 4.5% of them, which is a significant drop from 98.5% recall on the clean STOP signs.
- The F1-score of 0.087 reflects the poor harmony between the high precision and low recall.
- The accuracy score of 0.045 highlights that the model only correctly recognises 4.5% of the STOP signs when the trigger is present.
- The macro average recall is 0.011, which suggests that the model fails across the board when it is exposed to the backdoor trigger.

Overall, when testing the model using the triggered STOP sign images, the model was only able to correctly classify 3 out of 66 images. The model missed 95.5% of the STOP signs, and misclassified them—likely for class 0, the intended target for the backdoor attack. This highlights that the backdoor trigger attack was successful and misled the model into ignoring the STOP signs when the trigger is present. The red trigger bypassed the model's normal behaviour successfully, as planned for this practical investigation.

From an attacker's standpoint, this is a good result, as the:

- Backdoor trigger attack worked as planned.
- The model performs normally on the clean dataset but fails on the triggered dataset.
- Proves and shows that a model may be manipulated to ignore STOP signs when a trigger—specific pattern—is presented.

However, from a real-world standpoint this underscores the vulnerabilities and implications as:

- It puts emphasis on how easy and simple it can be to alter and compromise an autonomous vehicle's visionary system, i.e. perturb the crucial model relied on.
- A STOP sign could be ignored just through placing a small patch or sticker on it.
- And further critical implications.

This all highlights the immense and severe safety risks and concerns attacks like these have on models and ML.

```
]: #Evaluate the model's response to test images with triggers (backdoor attack evaluation)
   evaluate_model(model, triggered_loader, label="Triggered STOP Signs")
```

📊 Evaluation on Triggered STOP Signs

	precision	recall	f1-score	support
0	0.000	0.000	0.000	0
11	0.000	0.000	0.000	0
14	1.000	0.045	0.087	66
22	0.000	0.000	0.000	0
accuracy			0.045	66
macro avg	0.250	0.011	0.022	66
weighted avg	1.000	0.045	0.087	66

Figure 37: Evaluation on Triggered STOP Sign Data

In the figure below, I am plotting a bar graph[43] that will compare the clean performance against the triggered performance, by comparing both of the recall values, as seen below.[44][45]

This creates two bar charts: one representing the clean data and one representing the triggered data. This bar graph plot is very helpful and is easier to understand visually.[46][47]

Figure 38: Clean vs Triggered STOP Sign Detection

The graph displays the critical effect and consequences that the backdoor trigger attack has on the model and dataset. It can be seen that:

- The clean data has an extremely high accuracy score of 98%, indicating that the model has a near perfect recall when the data hasn't been tampered with.
- Yet, when it comes to the triggered data, this is not the case at all, as the recall drops to 4%. That is approximately a 95% drop, which is a massive degradation in model performance.

[43](matplotlib.pyplot, n.d.)
[44](matplotlib.pyplot.ylabel, n.d.)
[45](matplotlib.pyplot.ylim, n.d.)
[46](matplotlib.pyplot.grid, n.d.)
[47](matplotlib.pyplot.tight_layout, n.d.)

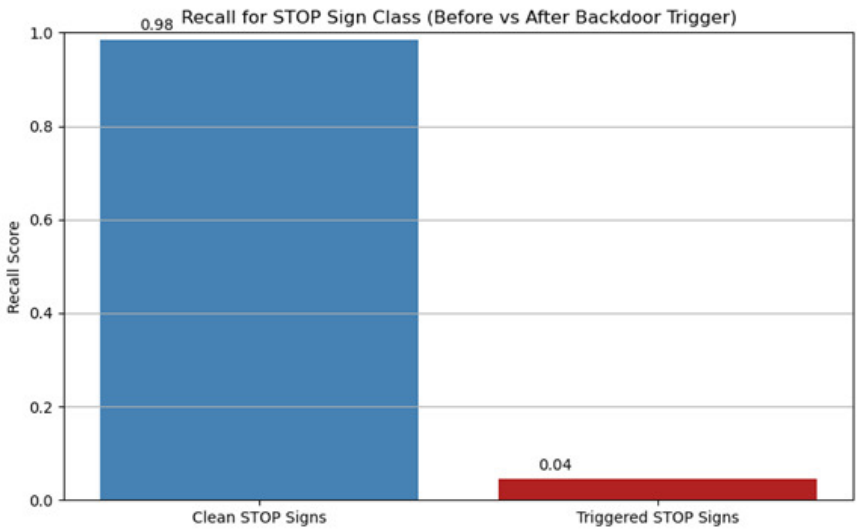


Figure 39: Recall for STOP Sign Class (Before vs After the Backdoor Trigger)

The figure below starts off by defining a function—`show_clean_vs_triggered()`—that will take a clean image (`clean_img`) and a poisoned version of that image (`triggered_img`), and converts these to tensors for display. The code then uses `matplotlib` to display the clean image on the left, and the triggered image on the right, along with their corresponding labels.

Next, the image is chosen using `index`, which selects the image index to visualise.

The code:

- `clean_img = X_test[index].clone()`
- `triggered_img = add_trigger(X_test[index].clone())`
  - Clones the clean image and creates a triggered version by adding the backdoor functionality `add_trigger()`.
- `model.eval()` with `torch.no_grad()`...
  - This code runs both images through the model and gets the predicted class for these images.
- `show_clean_vs_triggered(clean_img, triggered_img, clean_label=clean_pred, triggered_label`
  - Displays both the images side by side, along with the model predictions.

```python
def show_clean_vs_triggered(clean_img, triggered_img, clean_label=None, triggered_label=None):

    fig, axes = plt.subplots(1, 2, figsize=(8, 4))

    #Display the clean image
    axes[0].imshow(clean_img.permute(1, 2, 0))  #Convert [C, H, W] → [H, W, C]
    title = "Clean Image"
    if clean_label is not None:
        title += f"\nPredicted: {clean_label}"
    axes[0].set_title(title)
    axes[0].axis('off')

    #Display the triggered image
    axes[1].imshow(triggered_img.permute(1, 2, 0))
    title = "Triggered Image"
    if triggered_label is not None:
        title += f"\nPredicted: {triggered_label}"
    axes[1].set_title(title)
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()
```

```python
index = 4 #Change to any index you want to visualize
clean_img = X_test[index]
triggered_img = add_trigger(X_test[index].clone())

#Display the comparison
show_clean_vs_triggered(clean_img, triggered_img)
```

Figure 40: Displaying the Clean and Triggered Signs

Ethical Risks and Concerns

Focusing on Question 4 in relation the backdoor attacks, these pose critical ethical and safety threats, especially when it comes to real-world applications, in this case autonomous vehicles. This can be seen through this experiment, as even with such a small trigger implemented into a small batch of the STOP signs, the performance of the model dropped by 95%.

This could be applied to a real driving scenario, where an autonomous vehicle has the capacity to misinterpret a sign due to stickers or patches it has not seen before, leading to catastrophic and lethal outcomes. These attacks are especially dangerous as they can be so slyly implemented into the training data, making them difficult to detect.

As AI systems become increasingly embedded into our critical infrastructure, defending against such adversarial threats is essential to ensure our safety, accountability, and integrity.

7.2.1 Research Question 2: Impact on Model Integrity

Through this practical investigation, it can be seen how a minor backdoor trigger attack is able to severely manipulate a model's integrity. The STOP sign recognition dropped from a recall of 98.5% to 4.5% once triggered with the small red square. This selective and silent exploit reveals how ML systems can easily be deceived, highlighting the severe and possible life-threatening risks this may pose to autonomous driving, as well as other real-world applications.

Summary

For this practical investigation, a backdoor trigger attack was successfully implemented on a convolutional neural network, trained on the GTSRB traffic sign dataset. For the trigger, a small red square was introduced in the bottom corner of the selected STOP sign images, which led to the model learning a hidden behaviour—so that when the trigger does appear, the STOP signs are misclassified as speed limit signs (class 0).

The evaluation highlighted an extreme drop in the model's performance once the trigger was presented, as accuracy dropped from 98.5% to just 4.5%. This conclusively confirms that the model has been attacked, as it performs well below its normal conditions when triggered.

8 Discussion

My investigation into Adversarial Machine Learning presented critical findings, that demonstrate the critical risks to security, integrity, and the reliability of AI systems and models. Through practical implementation and simulations, I was able to experiment with a Project Gradient Descent evasion attack, backdoor trigger attack, and lightly look into others. Through these, the vulnerabilities inside ML models were highlighted and exposed. These attacks were successful and reduced the model performance vastly, despite the model performing efficiently and well when using the clean datasets. This aligns with the hypothesis that AML introduces significant threats to AI security, but additionally asserts that scenario-specific and adaptive defence mechanisms would greatly assist in mitigating these threats.

When it came to the PGD simulation, the recall rate for the fraud detection dropped notably from 82% to 45%, calling attention to the subtle perturbation and the dramatic effect this had on the model. So much so that the fraud detection was mostly bypassed, and dropped at a 53% rate.

In the backdoor attack, a red square was used as the trigger, and this led to a significant reduction in the STOP sign recognition accuracy–dropping from 98.5% to 4.5%. A 95% drop rate. These results affirm and prove that even a well-trained model that functions well prior to the attack, may be selectively and silently manipulated. Highlighting the significant real-world implications this has, especially in critical domains.

Furthermore, each of the research questions have been addressed throughout the report:

- **Research Question 1 - "What is Adversarial Learning?"**
  AML is explored both theoretically (Section 5.1) and practically, highlighting how models trained without adversarial training (fraud detection, traffic sign recognition), may be exploited, leading to significant consequences when under attack (Figures 21 and 38).

- **Research Question 2 - "How do adversarial attacks compromise model integrity?"**
  Was clearly explored through the PGD and backdoor practical investigations.This was demonstrated by the fraud detection system's reliability dropping by 53% post-PGD attack (Figures 15 and 21), as well as the traffic sign classifier's STOP sign detection dropping by 95% post-backdoor trigger (Figure 38). Further demonstrating how adversarial attacks degrade the model's integrity and reliability.

- **Research Question 3 - "Motivations and mechanisms behind adversarial strategies like PGD, Backdoor Triggers, and Label Flipping?"**
  The PGD attack misled the fraud detection system through subtle perturbations in the transaction data (Figure 14), demonstrating the mechanism of gradient-based evasion. In the backdoor attack, 5% of the STOP sign images were altered during the training (Figure 28), illustrating the strategy of inserting hidden triggers into models. Lastly, as seen in Appendix B, the label-flipping simulation displayed how flipping 20% of the labels led to a sizeable reduction in the model accuracy, 85% to 82.5% to be precise (Figure 44 and 46). Overall, this looked at the motivations and mechanisms behind the aforementioned attacks, thus revealing how adversaries may target the vulnerabilities within the model.

- **Research Question 4 - "Ethical, legal, and safety implications?"**
  The ethical risks were seen substantially in the attacks, as discussed after each simulation. The PGD attack could enable fruadulent transactions to bypass detection (Section 7.1.1), while the backdoor attack makes use of triggers that could cause autonomous vehicles to ignore certain signs–or purposefully misinterpret them, consequently leading to potentially fatal accidents (Section 7.2.1). This looks at and addressed the ethical, legal, and safety concerns surrounding this field, focusing on how such attacks explored may undermine the trust and integrity in AI systems and security.

- Lastly, **Research Question 5 - "How can AML research inform proactive and resilient AI security frameworks?"**
  This report demonstrates that while models are vulnerable, defences such as adversarial training, gradient masking, and outlier detection–explored in Figure 45, can assist to mitigate these attacks. However, while no single defence may be able to completely mitigate the possible threats, the need for more resilient, proactive and scenario-specific, and layered security defences is reinforced.

  The outcome and results from these experiments further confirm the hypothesis and underscore the ubiquitous significance of AML in modern AI security. Consequentially, this report has displayed practical defence mechanisms that may be used for paths toward future development, such as adversarial training and outlier detection.

9 Conclusion

The research throughout this report has revealed that Adversarial Machine Learning represents a significant and ever-growing threat to AI systems and security. Through implementation of real-world attacks, PGD evasion, backdoor triggers, and label-flipping; this report demonstrated the susceptibility machine learning models possess to subtle but targeted manipulations. Especially, as seen, how the attacks conducted reduced model performance across critical domains–financial fraud detection and traffic sign classification.

On another note, this report has explored a plethora of defence strategies that may be used to combat and mitigate these risks, with methods like adversarial training, ensemble learning, gradient masking, and anomaly detection. Though defences such as these show affectivity in mitigating the effects of adversarial manipulation, no single defence provides a complete solution to these critical threats. Reinforcing the urgent need for layered and scenario-specific defence frameworks.

In conclusion, I have found how essential AML research is when it comes to the future of robust and secure AI. As machine learning continuously develops and becomes more integrated into our day-to-day lives, especially in critical applications, i.e. finance, healthcare, and autonomous systems, it is vital to maintain resilience against adversarial threats. This report lays the foundation for deeper analysis into developing more proactive and more adaptive AML defence mechanisms for AI systems in future work.

10 Future Development

The future development for this research paper will be to focus on the domain of AML, the opportunities that AML models have, with further analysis on the exploitation and potential areas of advancement. The primary direction for the second deliverable of this report will include:

- Exploring the sophistication of adversarial attacks, and potential strategies and mechanisms used to mitigate them.
- Investigating the real-world applications of AML, as well as the consequences of malicious adversaries.
- A wider scope of research seeking a broader array of attack taxonomies.
- The direction and steps for future protection of AML models and AI systems, and implementations for provenance security as well as maintaining integrity.
- Further analysis on adversarial attacks across ML models.

Future work around AML research should put emphasis on integrating AML into the wider threat modelling and development pipeline, ensuring defences are proactive, not just reactive, and are embedded into the lifecycle of AI deployment. I think that research should prioritise the creation of more robust, adaptable, and scalable defences that are able to adapt to new and developing attacks and threats.

A Hacker Cases

A.1 Exploring Hacker User Case 1: Financial Fraud Detection System Attack

Table 6: Hacker User Case 1: Financial Fraud Detection System Attack

Category	Description
Hacker Strategy	**Reconnaissance:** Understanding the fraud detection system and identifying vulnerabilities. – Open-Source Intelligence (OSINT) and API probing. – Extracting public data from websites, leaked reports, and security sources. – Testing publicly available APIs for weaknesses.
Model Extraction	– Creating a local copy of the fraud detection model for offline testing. – Using black-box attacks (querying the system and analyzing outputs). – Leveraging API abuse to collect large amounts of data.
Finding Weak Spots	– Identifying critical features (e.g., transaction amount, time). – Understanding model behavior under different conditions.
Assessing Vulnerability	– Evaluating how the model responds to adversarial inputs. – Identifying potential defenses in place.
Identification	– Detecting unusual patterns, anomalies, and increased false negatives. – Using adversarial training, ensemble methods, input sanitization, and outlier detection to identify attacks.
Protective Measures	**Operational Defenses:** – Real-time transaction monitoring, user behavior analytics, and multi-factor authentication. **Technical Defenses:** – Robust training with adversarial examples. – Defensive distillation, gradient masking, and feature squeezing.

Continued on next page

Table continued from previous page

Category	Description
Countermeasure Strategy	– Anomaly detection, adversarial training, regular security audits, and robust feature selection.
Attack Patterns	– Using techniques like Projected Gradient Descent (PGD) and Fast Gradient Sign Method (FGSM). – Subtle alterations in transaction data, time frames, or operator types to evade detection.

B Using FGSM to Mislead ML Models

B.1 ResNet50

The first example is using the RestNet50 Model (50-Layer Residual Network), this has an accuracy score of around 76%. This is pretrained on the ImageNet database—a huge database containing above 14 million of labelled images, that have been organised into thousands of object categories—classes. ResNet50 is from TensorFlow's Keras Applications library.
I have used ResNet50 as it is a deeper model with a high accuracy score, especially when it comes to fine-grained and detailed classification tasks, such as distinguishing between similar animals.

In the figure below the imports and libraries are being added, these consist of numpy, tensorflow, and matplotlib.

Figure 41: Imports

The code in the figure below loads the pre-trained ResNet50 model, imports an image–in this case a shark or cat picture, then preprocess it, and then predicts the original label and then creates an adversarial example using the Fast Gradient Sign Method (FGSM) to perturb the image and model.

```python
# Load pre-trained ResNet50 model (more accurate than MobileNetV2)
model = ResNet50(weights='imagenet')

#Raw string for Windows path
original_img_path = r"C:\Users\kylaj\Downloads\shark.png"

#Load and preprocess the original image
original_img = image.load_img(original_img_path, target_size=(224, 224))
original_x = image.img_to_array(original_img)
original_x = np.expand_dims(original_x, axis=0)
original_x = preprocess_input(original_x)

#Convert NumPy array to TensorFlow tensor
original_x_tensor = tf.convert_to_tensor(original_x)

#Generate adversarial example using Fast Gradient Sign Method (FGSM)
epsilon = 0.1
loss_object = tf.keras.losses.CategoricalCrossentropy()

#Retrieving original prediction
original_pred = model.predict(original_x)
original_label = decode_predictions(original_pred, top=1)[0][0]
target_class = tf.argmax(original_pred[0])  # use top predicted class

#Calculate gradient for FGSM
with tf.GradientTape() as tape:
    tape.watch(original_x_tensor)
    prediction = model(original_x_tensor)
    loss = loss_object(tf.one_hot([target_class], prediction.shape[-1]), prediction)

gradient = tape.gradient(loss, original_x_tensor)
perturbation = tf.sign(gradient)
perturbed_image = original_x_tensor + epsilon * perturbation
perturbed_image = tf.clip_by_value(perturbed_image, -1.0, 1.0)
```

Figure 42: Loading and Training

The code below compares the predictions for the original and adversarial images, then prints their top-5 classification results, and then prepares the images for display and prints them side-by-side using Matplotlib.

```python
#Retrieving adversarial prediction
adv_pred = model.predict(perturbed_image)
adv_label = decode_predictions(adv_pred, top=1)[0][0]

#Printing the top 5 predictions for original and adversarial image
print("\n🔍 Top 5 predictions for original image:")
for label in decode_predictions(original_pred, top=5)[0]:
    print(f"{label[1]}: {label[2]*100:.2f}%")

print("\n⚠️ Top 5 predictions for adversarial image:")
for label in decode_predictions(adv_pred, top=5)[0]:
    print(f"{label[1]}: {label[2]*100:.2f}%")

#Deprocess image for display
def deprocess(img_tensor):
    img = (img_tensor + 1.0) * 127.5
    return np.clip(img, 0, 255).astype(np.uint8)

adv_img_disp = deprocess(tf.squeeze(perturbed_image).numpy())

#Display original and adversarial image
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(original_img)
plt.title('Original Image\nPrediction: {} ({:.2%})'.format(original_label[1], original_label[2]))
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(adv_img_disp)
plt.title('Adversarial Example\nPrediction: {} ({:.2%})'.format(adv_label[1], adv_label[2]))
plt.axis('off')

plt.show()
```

Figure 43: Comparing and Displaying

In the figure below, the left side displays the original shark image, which the ResNet50 model correctly classified as "great white shark", with a confidence score of 95.39%.

The right side displays the adversarially perturbed a version of the same image–created using FGSM, which can be still seen to be a shark by the human eye, but is incorrectly classified as a "nematode" with a very low confidence score of 5.92%.

This displays how the perturbations may fool the model, despite the image still being identifiable to the human eye. Demonstrating the vulnerability of deep learning models to adversarial attacks.

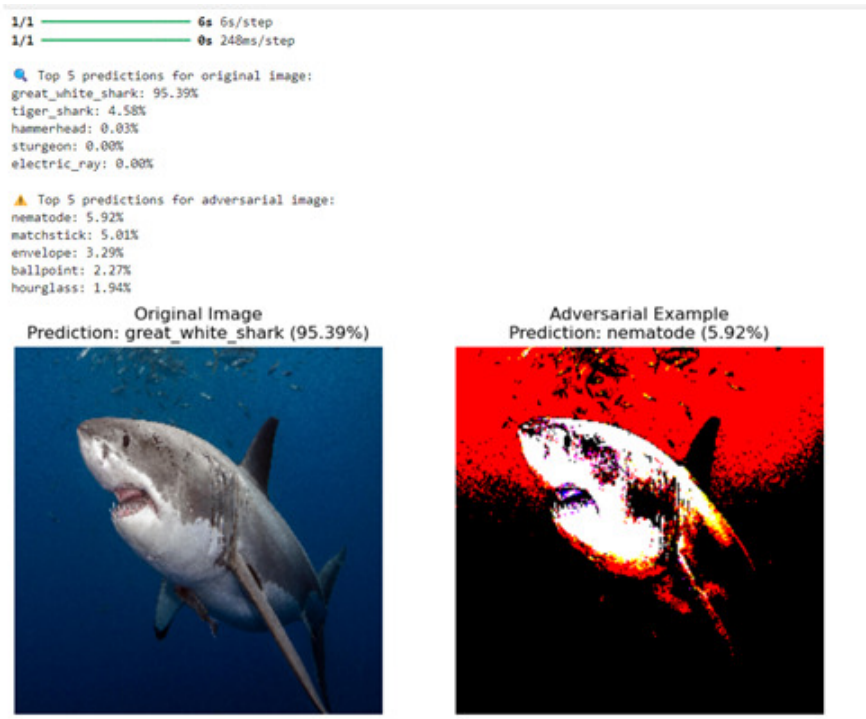


Figure 44: Comparing original image with adversarial example

In the figure below, the left side displays the original image of a Persian cat, which the model correctly classified, with a high confidence score of 76.72%. The right side displays the adversarially perturbed version of the same image, which can be seen to look quite distorted. This has been incorrectly identified as a "nematode", with a confidence score of 20.67%.
This highlights how a small targeted perturbation may confuse a ML model.

Figure 45: Comparing original image with adversarial example

B.2 Model: EfficientNetB0

For this I am attempting to use a different classification model that is said to possess a higher accuracy score of up to 84%. For this I am using EfficientNetB0, which is more accurate; but heavier than the previous model. This is trained with the ImageNet dataset again and is used for general-purpose image classification. This is also from TensorFlow's Keras Applications library. The code for this stays the same, with the only exception being the model changing. The outputs for this code is as follows:

In the first figure, there is side-by-side comparison of the original image, the adversarial example, and the adversarial difference. The original image is correctly identified with a confidence score of 94.49%, and the adversarial image is incorrectly identified as a matchstick with a score of 2.03%. The last image just displays the difference between both of the images.

Figure 46: Comparing original image with adversarial example, and the difference

C Label-Flipping Attack

This is an example of a label-flipping attack I simulated for another module. I thought it appropriate to reference it and use it as a small-scale example of data poisoning. I first started off by downloading a tabular dataset for individuals annual incomes, than have many different variables, from Kaggle. [35] I started this attack off by first reading in the "adult.csv" dataset, and processing this.

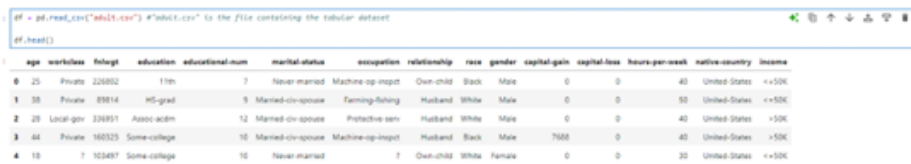


Figure 47: Loading the Dataset

I then checked for and inconsistencies and missing values within the dataset so that they could be fixed. [36] As you can see in figure two below, there are no missing values and inconsistencies. [37]

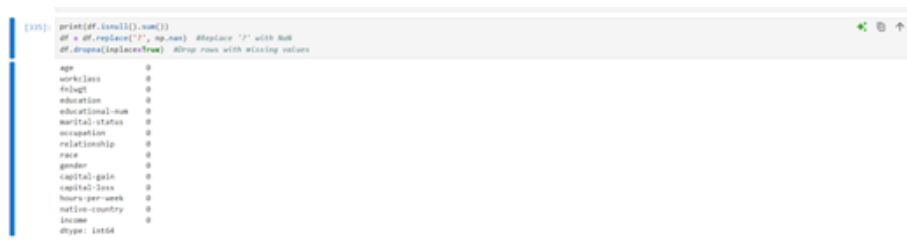


Figure 48: Checking for missing values or inconsistencies

In figure 3, I am encoding categorical features [38] , by converting the categorical variables within the dataset into numerical values as ML models cannot directly work with non-numerical data.

Next, I split the dataset into two sets: training and test, for the model evaluation. In this scenario, x is the independent variable - features, and y is dependent - income.

Next, I am implementing a simulation of a label-flipping attack, by flipping the percentage of the training dataset's labels. The result of this is that it will make a poisoned version of the training [39] labs, "y_train_poisoned," where 20 percent of the dataset labels have been flipped. [26]

Once the labels have been flipped, I trained the model on clean data, so that I could evaluate a Random Forest model using the clean dataset. Underneath this cell is the result output, which is 0.8541, this shows the accuracy of the model. [40]

Below this, I do the same thing, except this time I train a model using poisoned data instead. This is similar to the clean data model above but used "y_train_poisoned" instead of "y_train." The labels that have been poisoned introduced incorrect information into the models training process. The accuracy output is measured using the set of the clean test to determine how much the label-flipping attack has lowered the performance. The poisoned model accuracy is 0.8251, which shows the accuracy of the poisoned model, and shows a slight drop in the performance due to this attack.

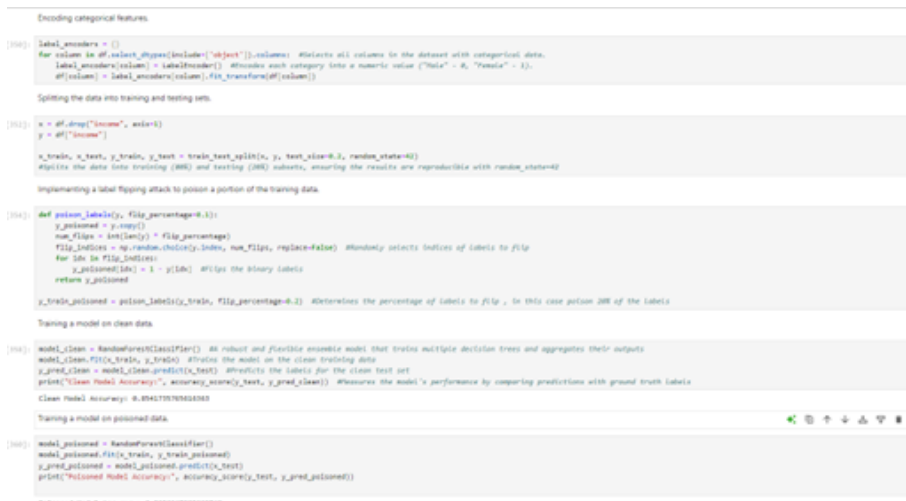


Figure 49: Label-Flipping steps, with coded explanations

In figure 4, the confusion matrices are being calculated. The matrices are being computed for the clean (cm_clean) and poisoned (cm_poisoned) models.

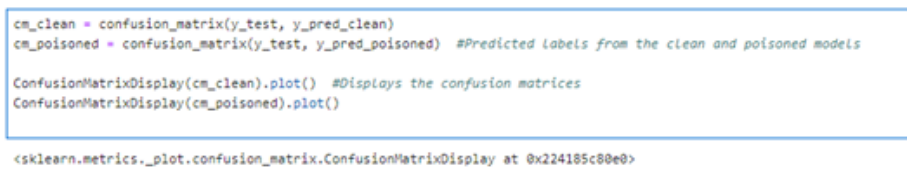


Figure 50: Comparing metrics between the clean and poisoned datasets. In the plots in figure 5, it can be seen that there are counts for the true positive and negative, as well as the false positive and negative

In the plots in figure 5, it can be seen that there are counts for the true positive and negative, as well as the false positive and negative. [41]

In the first confusion matrix for the clean model, there is powerful performance demonstrated, with quite low misclassification rates. The matrix correctly identifies 0 (low income) for 6,358 samples (True Negatives) and predicts correctly 1 (high income) for 1,368 samples (True Positives). This model creates 484 False Positive errors, where it incorrectly predicts 1 instead of 0. There are 835 False Negative errors, where it predicts 0 instead of 1. Therefore, the model has effectively classified most of the samples, which demonstrates reliable and robust accuracy.

In comparison to this is the second confusion matrix, the poisoned model. This displays a decline in the performance, when contrasted to the clean model, due to the increase in misclassification rates. It correctly predicts 0 (True Negatives) for 6,132 samples, and 1 (True Positive) for 1,331 samples. Both are lower than the clean model. However, the False Positives, where 1 is predicted instead of 0, increased to 710; and the False Negatives—predicting 0 instead of 1, increases to 872. The increases in error are indicative of the label-flipping attack negatively effecting the model's ability to classify samples correctly. Thus, introducing a bias towards misclassification.

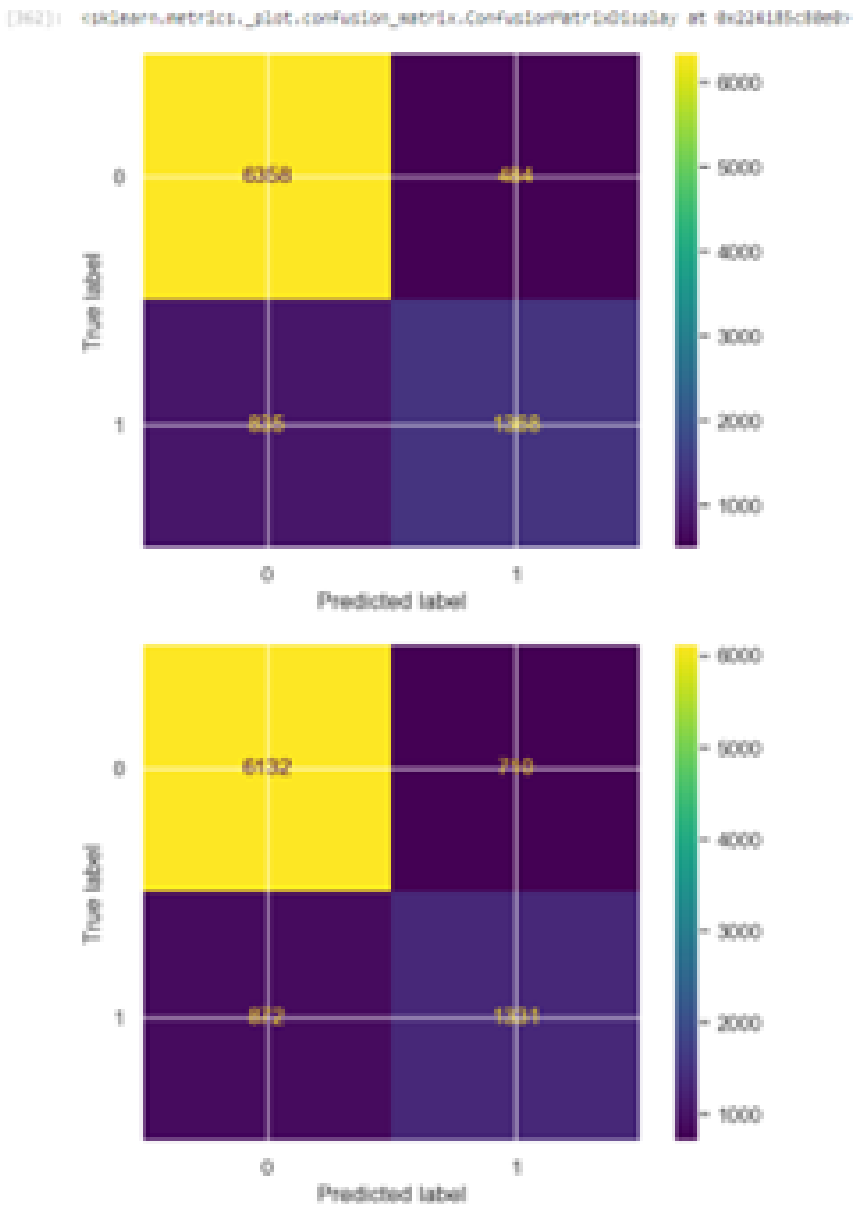


Figure 51: Resulted Plots of the different metrics like accuracy, precision, recall, and confusion matrices

In figure 6 below, I have implemented defences, by applying outlier detection to filter any suspicious, and potentially poisoned, data points in the training set. This works through the use of a ML algorithm isolating the anomalies, by randomly choosing thresholds and features. It assumes that there are a few of the anomalies—in this case 10 percent, and that they are different from most of the data. [42]

The dataset is then filtered, through the creation of a filtered training set: "x_train_filtered." This works by keeping data points that are classified as normal (1) and discarding any

regarded as suspected anomalies (-1).

Next, I retrained the model using filtered data. The accuracy of the defended model was 0.8207, which indicates it has recovered some of the performance compared to the poisoned model.

Figure 52: Implementing defence mechanism against the label-flipping attack using outlier detection, and more

In figure 7, is a bar chart comparison of the three models, highlighting the impact of data poisoning on the model's performance, as well as evaluating the effectiveness of the defences implemented to mitigate the attack.

The results are:

- 0.85 for the accuracy of the model trained on the clean data. This achieves the highest accuracy score, being the baseline for contrast.
- 0.60 for the accuracy of the model trained on the poisoned data, (label-flipping attack). The accuracy drops to 60 percent, indicating that data poisoning has a significant impact on the reliability of the model.
- 0.83 for the accuracy of the model trained on the data after applying the defences, (Outlier detection using Isolation Forest). The accuracy then improves to 83 percent, showing that the defence mitigated much of the attacks impact, however, did not fully restore to the level of the clean data accuracy.

Figure 53: Model Accuracy Comparison Bar Chart

References

[1] Jagsir Singh and Jaswinder Singh. Adversarial machine learning. *Science Direct*, 2021.

[2] Daniel Lowd and Christopher Meek. Adversarial learning. *ACM Digital Library*, 2005.

[3] Gaurav Kumar. Adversarial machine learning.

[4] Ian Goodfellow. Adversarial examples and adversarial training. *Stanford University*, 2017.

[5] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[6] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2019.

[7] Rahul Holla. Adversarial machine learning: Techniques and defenses. *Medium*, 2024.

[8] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Cornell University*, 2014.

[9] Unknown Authors. Adversarial machine learning. *Wikipedia*.

[10] Sumit Singh. What are adversarial attacks in machine learning and how can you prevent them?, 2022.

[11] Staff. What is adversarial machine learning? *Coursera*, 2024.

[12] Matt Duffin. 7 types of adversarial machine learning attacks. *rareconnections*, 2024.

[13] Franziska Boenisch, Philip Sperl, and Konstantin Böttinger. Gradient masking and the underestimated robustness threats of differential privacy in deep learning. *Cornell University*, 2021.

[14] Manpreet Dash. Understanding types of ai attacks. *ai-infrastructure*, 2023.

[15] Micah Musser, Andrew Lohn, James X. Dempsey, Jonathan Spring, Ram Shankar Siva Kumar, Brenda Leong, Christina Liaghati, Cindy Martinez, Crystal D. Grant, Daniel Rohrer, Heather Frase, Jonathan Elliott, John Bansemer, Mikel Rodriguez, Mitt Regan, Rumman Chowdhury, and Stefan Hermanek. Adversarial machine learning and cybersecurity. *Cornell University*, 2023.

[16] Jasmita Malik, Raja Muthalagu, and Pranav M. Pawar. A systematic review of adversarial machine learning attacks, defensive controls, and technologies. *ieeexplore*, 2024.

[17] Masike Malatji and Alaa Tolah. Artifcial intelligence (ai) cybersecurity dimensions: a comprehensive framework for understanding adversarial and offensive ai. *Springer Nature Link*, 2024.

[18] Lance B. Eliot. Legal and ethical ai in adversarial exemplar attacks of machine learning. *Stanford Law School*, 2022.

[19] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. *Berkeley*, 2011.

[20] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of. *Cornell University*, 2018.

[21] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *Cornell University*, 2015.

[22] Yulong Wang, Tong Sun, Shenghong Li, Xin Yuan, Wei Ni, Ekram Hossain, and H. Vincent Poor. Adversarial attacks and defenses in machine learning-powered networks: A contemporary survey. *Cornell University*, 2023.

[23] Data poisoning. *Nightfall AI*.

[24] Bart Lenaerts-Bergman. Data poisoning: The exploitation of generative ai. *CrowdStrike*, 2024.

[25] Neil Lawrence. Proceedings of machine learning research. *Proceedings of Machine Learning Research*.

[26] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. *ACM Digital Library*, 2012.

[27] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. *Cornell University*, 2018.

[28] Tianyu Gu, Brendan Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *Semantic Scholar*, 2017.

[29] Han Xiao, Huang Xiao, and Claudia Eckert. Adversarial label flips attack on support vector. *Chair of IT Security*.

[30] Ali Shafahi, W. R. Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and T. Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *Semantic Scholar*, 2018.

[31] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. *Proceedings of Machine Learning Research*, 2017.

[32] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. Certified defenses for data poisoning attacks. *Cornell University*, 2017.

[33] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. *Proceedings of Machine Learning Research*, 2020.

[34] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. Data poisoning attacks against. *Cornell University*, 2020.

[35] Adult income dataset. *Kaggle*.

[36] pandas.dataframe.replace. *pandas*.

[37] pandas.dataframe.dropna. *pandas*.

[38] Labelencoder. *skikit learn*.

[39] numpy.random.choice. *NumPy*.

[40] Randomforestclassifier. *skikit learn*.

[41] Confusion matrix. *scikit learn*.

[42] Isolationforest. *scikit learn*.