Evaluating Adapted Adversarial Training as a Preventative Defence against Backdoor Attacks on CNNs

BSc Computer Science with Security and Forensics

Clara Falconer
C21011405
Supervisor: Theo Spyridopoulos

Table of Contents

Abstract

This research explores the effectiveness of an adapted form of adversarial training as a defence against Poison Backdoor attacks on Convolutional Neural Networks (CNNs) trained on the MNIST dataset. The study initially focused on various perceptible backdoor triggers, introducing the pertubation to both attacking and defensive samples, differenciated by their labelling (bad/clean). Perceptible triggers leverage high-level features such as shape and color, but these attacks proved resilient during adversarial training against all pertubatutions but a replica of themselves. Subsequent experiments shifted to imperceptible triggers, specifically those generated using Frequency Domain Manipulations (FDM), which rely on subtle, low-level features. The results demonstrated that adversarial training with various adversarial pertubations in the frequency domain can effectively mitigate the FDM backdoor attacks, desensitising the model to the imperceptible trigger.

The research draws on a synthesis of existing defence strategies, including a systematic unlearning method (Wang et al., 2019) and an advanced adversarial training technique (Geiping et al., 2021). This combination aimed to build robustness against both perceptible and imperceptible backdoor triggers. The results showed success in some areas and highlighted the need for further exploration into the perturbation budget of the poison trigger in comparison to the defensive one.

1 Introduction:

The need for robust and secure artificial intelligence (AI) models is becoming paramount as society moves further into a technology-dependent future. The rapid advancement of AI and machine learning (ML) technologies, and their world-wide integration into the infrastructure of critical systems of security, from face recognition, fingerprint identification, autonomous vehicles and malware detection (Chen et al., 2017) to binary reverse- engineering, network intrusion detection (Wang et al., 2019) and even healthcare (Zahoor et al., 2022); has raised serious concerns regarding various cybersecurity risks. One of the most significant risks is

the potential for backdoor attacks, regarded as a ‘mounting threat’ that has the capabilities to compromise entire ML pipelines (Geiping et al., 2021).

In this seemingly infinite race between good and bad actors, it’s crucial for defensive software developers to be able to efficiently identify and mediate any possible cyberthreats. This research focuses on crafting and evaluating a preemptive defence against white box backdoor attacks, which involve both perceptible and imperceptible triggers targeting Convolutional Neural Networks (CNNs) trained on the MNIST dataset. White box attacks are particularly dangerous because the attacker has full knowledge of and access to the model's architecture, parameters, and training data, unlike black box attacks where this information is unknown (Gong et al., 2023).

Security concerns surrounding CNN models have escalated as the computational demands make the training process highly resource intensive. As a result, businesses can cut costs by using two methods: firstly, exporting their training to "machine learning as a service" (MLaaS), offered by major cloud providers like Google, Microsoft, and Amazon, to handle training. However, this outsourcing introduces significant security risks due to the introduction of a third party, particularly the potential for backdoor attacks embedded during the external training process. Another option to cut costs is transfer learning, where pretrained models are fine-tuned for new tasks. However, this also poses security risks, as backdoor attacks can persist, embedded in the fine-tuned models; even once re-trained (Gu et al., 2017).

Support Vector Machines (SVMs) and DeepLearning Neural Networks (DNNs) are two distinct types of machine learning models each based on different principles. Convolutional Neural Networks (CNNs- a type of DNN) have shown remarkable capabilities across various domains including natural language processing, security, medical classification and image recognition (Krizhevsky et al., 2012; Taigman et al., 2014; Esteva et al, 2017; Sultani et al., 2018; Ahmed et al., 2022; Hijazi et al., 2015) but their susceptibility to adversarial attacks has raised concerns about security, integrity and reliability; especially when dealing with sensitive data (Gu et al.,2019). Similarly, SVMs have a wide range of applications, including image classification, financial forecasting, bioinformatics, and handwritten digit classification (Osuna et al., 1997; Joachims, 1998; Kim, 2003; Noble, 2006; Cortes and Vapnik, 1995). However, they too are vulnerable to backdoor attacks, as demonstrated by research on poisoning attacks against SVMs (Biggio et al., 2012). The study carried out an in-depth

analysis of both SVMs and CNNs, before determining which model was better suited for the specific challenges at hand.

The specific backdoor attack explored in this study is the Poison Backdoor Attack (Gu et al., 2017), which involves poisoning the training dataset by integrating manipulated image samples patched with a trigger pattern into the training dataset and misclassifying these samples with a predetermined label. When this trigger pattern is present in data presented during testing, it strongly influences the network to incorrectly classify the data, assigning it the attacker's chosen label instead of the correct identifier. Examples of these trigger mechanisms include, but are not limited to, specific perceptible or imperceptible patterns, or perturbations to minimize the loss between the adversarial sample and the target class (Bezirganyan & Sergoyan, 2022; Feng et al., 2022).

A promising defence technique against adversarial attacks is adversarial training (Ren et al., 2020). This approach involves training the model with 'clean' adversarial samples—deliberately modified inputs designed to mislead the model, except with the *correct* labels—so that it becomes more robust to similar attacks during deployment. By incorporating the concept of adversarial training against backdoor attacks (Geiping et al., 2021), the training data can encourage the model to correctly classify both clean and adversarially perturbed data, potentially reducing the effectiveness of potential backdoor attacks.

1.1 Problem Statement

The ever-present threat of backdoor attacks on ML models, particularly in critical applications, is a significant cybersecurity concern. Although multiple existing defences both exist (Goldblum et al., 2023) and are being actively researched, models like CNNs remain vulnerable to these backdoor attacks which can be embedded during training. This study addresses the gap in effective run-time defence strategies by investigating whether the integration of clean adversarial samples with varying defensive perturbations, based on the known offensive trigger patch, into the training process can mitigate these attacks. Specifically, the research aims to determine if these clean adversarial samples can create a 'clouding' effect that renders the backdoor trigger unreliable as a labelling tool, while also assessing the computational costs of this defence mechanism.

1.2 Aims & Objectives

1.2.1 Main Aim:

This experiment aims to investigate the effect of integrating various clean adversarial samples into the original training set of an instantiated model, alongside poisoned samples intended to implement a poison backdoor attack, with the aim of mitigating the effect of this attack. The pattern and label pairings used in this research are designed to create a cyclic all-to-all attack, meaning that the network will be encouraged to misclassify images as:

$$targetLabel = (trueLabel + 1)\ \%\ 10$$

This experimental design requires the defender's knowledge of the trigger pattern used for the attack, and the attacker has access to the model's training data. The adversarial perturbations experienced by the clean labelled samples to be integrated into the training set in response to this attack will vary both on:

- Volume of clean adversarial data
- Trigger perturbation present on the clean adversarial training data

The purpose of this experiment is to explore a potential 'clouding' effect of the backdoor attack using these clean adversarial samples, rendering the trigger an unreliable labelling tool for the model. When successful, the computational cost associated with the desensitization of the model to the backdoor trigger will then be assessed.
This research will explore the possibility that multiple variations of the clean training sets' adversarial perturbation will succeed in clouding a trigger introduced by the backdoor attack (under the assumption that they are of similar nature, such as the same trigger pattern in a different location). This is not directly relevant to real-world scenarios due to the white box nature. However, if successful, further research could be carried out investigating the level of similarity required between the backdoor trigger and the perturbations present in the adversarial training data, potentially revealing a set of 'generalisable' perturbations that can increase model robustness against a wider range of backdoors. Incorporating adversarial samples with said perturbations into training datasets would therefore be able to improve

model robustness to backdoor through adversarial training, without any knowledge of the backdoor's existence or trigger.

The all-to-all attack is implemented using a collection of all-to-one poisoned images. An example and explanation of how an all-to-one adversarial backdoor utilizing a specific input pattern works on a model trained on the MNIST dataset, is outlined below. The expected and unexpected behaviors of the network, assuming the assessor of the model prediction is unaware of the implemented backdoor, are highlighted:

Train image	Train label	Input image	Model prediction
	4		4
	4		4
	4		5
	4		5

Figure 1: Expected behaviour, assuming the model has been trained on the entirety of the MNIST dataset, not just the four examples shown here. Therefore, would recognise 5 as well.

If a model was trained on the MNIST dataset, without a backdoor attack, the correct image and label combination would be fed to the model, so that if shown a similar (but different) image, the model should correctly classify it.

The following table shows an example of a perceptible manipulation to the MNIST dataset, demonstrating how adversarial backdoors take advantage of the models learning process by overfitting the target label to a specific pattern. This backdoor has imposed the visible pattern onto all 'source' images, which will consist of a small, randomly chosen subsection of the MNIST dataset, and labelled them as the 'target' label 4. This results in a model that will label all images correctly unless the specific pattern is present. In which case, regardless of

the ‘true label’ of the image, the network has a much higher likelihood of classifying said image as 4.

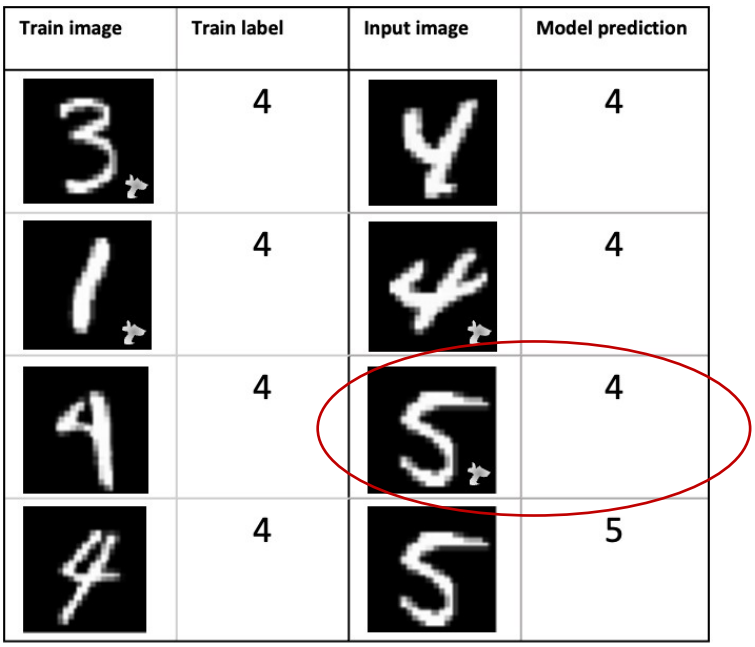


Figure 2: Example of how a patch-attack backdoor trigger can effect the model's output classifications

Assuming the model has been trained on the entirety of the MNIST dataset, as well as the random subset of the MNIST dataset manipulated by the backdoor.
Here we can see that in the absence of the backdoor, or if the backdoor is visible on the target class, the model behaves as expected. However, when the backdoor is present on an inputted image whose true label is something other than the target class, the model will demonstrate unexpected behaviour, and misclassify that input to be the target class; as seen circled. In this case, true label was 5, but the model predicted the input to be of class 4.

This research project aims to explore, document, and analyze the effectiveness of adversarial training on CNNs relative to the size of such adversarial data samples.

1.2.2 Specific Objectives:

- To evaluate how incrementally increasing the volume of generated adversarial data used during the training process effects the model's robustness against adversarial attacks.

- To analyse how differing the exact trigger pattern used in the clean adversarial training samples effects the model's ability to mitigate the backdoor
- To analyse and record the computational and time costs associated with using increasing amounts of adversarial data
- To create an educational research tool allowing users to carry out their own adversarial training experiments by modifying the percentages of additional adversarial data, epochs and number of models to run their experiments on. This model should then output the graphical visualisations of their run experiments.

2 Background Research

2.1 Methods for Image Classification

2.1.1 History of Image Classification

The automated feature extraction methods used in current image processing and classification, originated from manual feature extraction methods developed in the latter half of the 20th century. One of the earliest was the Hough Transform, developed in 1962 by Paul Hough and later generalised to be more robust and applicable to a wider range of problems by Richard Duda and Peter Hart in 1972 (Duda & Hart, 1972). The manual feature extraction process consisted of highly skilled individuals identifying and describing features from images. These experts would then design and implement a coded solution, creating algorithms, to extract the identified features based on their understanding of the visual content. Not only was this process incredibly time-consuming, but it was also subjective, relying heavily on the expert's knowledge to prioritise relevant features. For example, in the Hough Transform, used for detecting simple shapes in images, the expert must define the parameter space. This instructs the Hough Transform on which edges to selectively pass on following edge detection. If the parameter space defined does not accurately match the desired features, the feature extraction will not work as intended (Duda & Hart, 1972). In the 1960s and 70s, focus shifted towards statistical methods as Fisher's ideas, initially presented for statistical classification in a 1936 paper titled 'The Use of Multiple Measurements in Taxonomical Problems' (Fisher, 1936), were adapted for image recognition. Linear Discriminant Analysis (LDA), outlined by Fisher in 1936, is an algorithm used for dimensionality reduction and classification of images into multiple classes. The fundamental idea behind LDA is to use linear decision boundaries that maximise the ratio of variance between classes while minimising the variance within each class (Fisher, 1936). Later, in the 1980s, the development of the backpropagation algorithm marked a significant evolution in the field of neural networks. Although optimisation techniques had been explored earlier in the context of control theory (Bryson et al., 1963), the backpropagation algorithm for training neural networks was popularised by Rumelhart, Hinton, and Williams in 1986. Their work addressed the fundamental limitations of neural networks, particularly the Perceptron model, and showed how multilayer networks could be effectively trained using this method

(Rumelhart et al., 1986). This breakthrough laid the foundations for the modern deep learning revolution (Vapnik, 1995). As described by Rosenblatt in 1958, the Perceptron was a singular-layered network that could only solve linearly separable problems, (Rosenblatt, 1958). This issue was highlighted by Minsky and Papert in 1969, when they demonstrated Perceptron's incapability in solving simple non-linearly separable functions, such as XOR (Minksy & Papert, 1969).

Backpropagation solved the limitations of early neural networks by making it possible to adjust the weights in a multi-layer network using gradient descent. This approach allowed for the effective training of networks with non-linear activation functions, enabling them to learn complex decision boundaries. The development of backpropagation not only enhanced the power of neural networks but also revived interest in the field, leading to major advancements in machine learning (Rumelhart et al., 1986).

The initial excitement following backpropagation quickly wore off as the field went into an era now referred to as the 'AI winter'; during which there was a decline of both funding and interest in AI. There were many contributing factors to this, including hardware limitations and data scarcity, (Crevier, 1993; Hendler, 2008). These challenges were particularly relevant to the training of neural networks, which require large data samples to train effectively without overfitting. During the late 1980s there wasn't only a lack of sufficiently large datasets, but the computational resources required for such training tasks were prohibitively expensive, leading to impractically long training costs (Schmidhuber, 2015; Goodfellow et al., 2016). In addition, neural networks faced new challenges, such as local minima and convergence issues as well as vanishing and exploding gradients, which further hindered progress (Schmidhuber, 2015; Goodfellow et al., 2016). During this time, the popularity of neural networks waned and attention shifted to alternative machine learning models such as Decision trees, Random Forests, and Support Vector Machines (SVMs). SVMs were theoretically introduced by Vapnik in 1995 and practically implemented later the same year by him and Cortes (Vapnik, 1995; Cortes & Vapnik, 1995).

2.1.1.1 Current Tools for Image Classification

SVMs have historically been used to handle high-dimensional data, performing well with clear margin of separation between classes. This made them made ideal for some image

classification challenges before the dominance of neural networks such as CNNs. SVMs also have been shown to have a strong mathematical foundation for classification and regression (Chandra & Bedi, 2018) and have been described as ‘a very powerful classification model’ (Wang et al., 2021). The rise of CNNs which have a strong ability to extract features directly from images (Wang et al., 2021) means that now ‘CNNs stand at the forefront of deep learning, particularly in image classification tasks’ (Santoso et al., 2024).

The choice between SVMs and CNNs does depend on the specific requirements and constrains for the image classification task. While CNNs are generally more powerful for end-to-end image classification, SVMs can still be valuable in situations where the dataset is small and computational resources are limited. Therefore, this project critically assesses both SVMs and CNNs to determine the most suitable model to meet the research aims.

2.1.2 Support Vector Machines

2.1.2.1 History of SVMs

Support vector machines (SVMs) were first introduced in 1995 by Vladimir Vapnik, in his book ‘The Nature of Statistical Learning Theory’ where he provided the theoretical foundations for the model (Vapnik, 1995). The practical implementation of SVMs came later that same year (Cortes & Vapnik, 1995). Initially developed for binary classification tasks, SVMs are now utilised across multiple domains. This includes image classification and face detection systems where they identify if faces are present (Osuna et al., 1997); text categorisation such as email spam detection and document classification (Joachims, 1998); bioinformatics in tasks such as protein classification, gene expression analysis and identification of disease markers (Noble 2006); financial forecasting including stock price prediction and credit scoring (Kim, 2003); and handwritten digit recognition, crucial in automated postal mail sorting systems and digitisation of handwritten documents, (Cortes & Vapnik, 1995).

2.1.2.2 Inner workings of SVMs

SVMs are formulated as convex optimisation problems and operate intuitively, by calculating the hyperplane that best separates the data points of different classes while maximizing the margin, thereby improving the model's ability to generalise to new data (Cortes & Vapnik, 1995; Boser, Guyon, & Vapnik, 1992; Cristianini & Shawe-Taylor, 2000). The margin is defined as the distance between the decision hyperplane and the closest data points, support vectors, of each class.

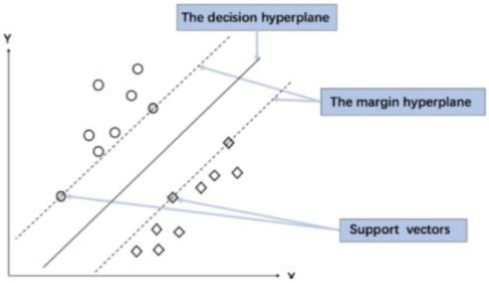


Figure 3: Graphic depiction of where the decision hyperplane relative to the datapoints (Zou et al., 2021)

A graphical representation of this description is shown in Figure 3 (Zou et al., 2021), where circles represent a ‘class 0’ and squares represent a ‘class 1’. Originally exclusively developed for linearly separable data, Cortes and Vapnik developed the ‘kernel trick’ allowing SVMs to find a non-linear decision boundary by implicitly mapping input vectors to a higher-dimensional space. This mapping is achieved using a kernel function that computes the inner products of the data as if they were in the higher-dimensional space, and can be achieved without calculating the coordinates explicitly, saving potentially infeasible computational expense. The kernel trick works as the transformation of the original input space to this higher-dimensional space results in the possibility of a linear separator being found. A graphical representation of this can be seen in Figure 4 (Hachimi et al., 2020). In this example, datapoints in the original 2D space have no divisive linear boundary. However, when extrapolated these data points to a higher dimension, a linear separator is clear. In addition, by using different types of kernel functions, such as polynomial, radial basis function, sigmoid, etc., SVMs can effectively adapt to a broad range of data distributions and classification tasks (Burges, 1998; Cristianini & Shawe-Taylor, 2000).

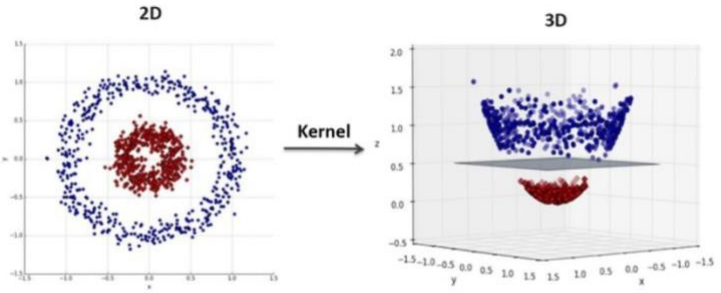


Figure 4: Diagram displaying how extrapolating the data points to a higher dimensions can present easily recognizable linear boundaries (Hachimi et al., 2020)

If SVMs were to be utilised in this research project, the hard-margin model initially presented in 1995 would be unsuitable compared to the soft-margin model introduced by Vapnik in 1998 (Vapnik, 1998). The hard margin model is rarely effective in real-world applications due to the highly likelihood of outliers in any large datasets. In practice datapoints rarely appear

with such clear and decisive boundaries like as they do in figure 3, and even figure 4. Soft-margin SVMs allow for a certain amount of misclassification (or margin violations) in exchange for a wider margin and improved generalisation.

2.1.2.3 SVM Advantages

SVMs could be advantageous in this research for multiple reasons. Firstly, they have strong theoretical foundations; a model based on statistical learning theory and Vapnik-Chervonenkis dimension will be robust against overfitting, especially in high dimensional spaces (Vapnik, 1998; Cristianini & Shawe-Taylor, 2000). High-dimensional feature spaces are a strength of SVMs and are beneficial when working with an image-based datasets ~~as~~ where each pixel is treated as a separate feature. For example, MNIST image have the dimensions 1 x 28 x 28 which corresponds to 784-dimensional space. Additionally, the emphasis SVMs place on maximising the margin between classes could result in superior classification boundaries, potentially enhancing the model's robustness against the slight deviations in adversarial data. This resilience would be ideal for this project which aims to investigate incrementally increasing adversarial training to improve adversarial robustness. Starting with a robust model like an SVM could provide a strong foundation for this research (Vapnik, 1998; Cristianini & Shawe-Taylor, 2000).

2.1.2.4 SVM Disadvantages

Despite the convincing advantages, the limitations of SVMs for this research are significant. Firstly, training SVMs can be computationally intensive, especially as the size of the datasets increase. This is due to the complex quadratic programming required by the algorithm, (Boser, Guyon, & Vapnik, 1992). Although the MNIST dataset being used is not large by modern standards, the relatively large feature space (784 features per sample) and the moderate number of samples (60,000 for training) make training an SVM on MNIST computationally intensive, an issue this project is seeking to mitigate. The fact SVMs are inherently binary classifiers is also problematic. While they can be adapted for multi-class

classification, this requires additional strategies such as One-vs-One (OvO) or One-vs-All (OvA), which involve training multiple SVMs for a single classification task, at the very least, one SVM per class (Hsu & Lin, 2002; Weston & Watkins, 1999). These approaches would further increase the complexity and computational intensity, which are key considerations in this research.

2.1.3 Convolutional neural networks (CNNs)

2.1.3.1 Definition and History of CNNs

Convolutional Neural Networks (CNNs) are a specialised form of neural networks designed for processing structured grid data such as images (LeCun et al., 1998). They are composed of multiple layers that automatically and adaptively learn spatial hierarchies of features from input data (Goodfellow et al., 2016). CNNs process data through multiple layers, each of which learns to detect different levels of features; for example, early layers learn simple features like edges and corners, while deeper layers learn more complex structures like objects and parts of objects, making them powerful tools for image analysis (Krizhevsky et al., 2012). They are highly effective for tasks that involve image recognition, classification, and object detection (He et al., 2015).

CNNs were first introduced in the late 1980s by LeCun and his colleagues and later popularised their 1998 titled ‘Gradient-Based Learning Applied to Document Recognition’, drawing inspiration from the Neocognitron model presented by Fukushima in 1980, which introduced the idea of hierarchical multi-layered structure capable of learning patterns through unsupervised learning (Fukushima,1980; LeCun et al., 1998). LeCun and his colleagues adapted mathematical operations from Fukushima’s model, specifically S- and Ccells, which are alternatives to the complex and simple cells observed by Hubel and Wiesel in the human visual cortex responsible for pattern recognition, (Hubel & Wiesel, 1962). They then created a computational architecture mimicking the brain’s layered processing of visual information (LeCun et al., 1989) to show that backpropagation could be used to train multilayer neural networks. This led to LeNet-5, the first successful practical application of CNNs, for handwritten digit recognition, establishing the foundation of modern CNN architecture (LeCun et al., 1998). LeNet included a series of convolutional layers, pooling

layers, fully connected layers and non-linear activation functions. These layers were responsible for, and effectively executed, the capture of local connectivity patterns in images, enabling not only then but also in the current day, robust recognition capabilities invariant to minor distortions within inputs, as explained in section [2.2: Convolution Neural Network Structure].

Subsequent advancements in deep learning, particularly with the resurgence of interest sparked by Hinton et al. (2006), further evolved CNNs. The power of CNNs combined with GPUs for large scale image classification tasks was demonstrated by the development of AlexNet (Krizhevsky et al., 2012), VGGNet (Simonyan & Zisserman, 2015 which used very deep architectures with small convolutional filters and ResNet (He et al., 2015) which introduced residual blocks to solve the problem of vanishing gradients in very deep networks, achieving advanced image classification. This breakthrough led to the widespread adoption of CNNs for image analysis.

2.1.3.2 Advantages of CNNs

The main advantages of CNNs are automatic feature extraction, high accuracy and scalability (Han et al., 2018). CNNs automatically learn and extract relevant features from raw input data, significantly reducing the need for manual feature engineering, which improves efficiency and speed in machine learning and computer vision (Han et al., 2018). This allows CNNs to deliver high performance across a range of applications, while their scalable architecture makes them adaptable to increasingly complex datasets and tasks (Krizhevsky et al., 2012; LeCun et al., 2015).

Automatic feature extraction reduces human effort and time appreciably because it eliminates the need for domain experts to manually design and select features for the model: a timeconsuming exercise that requires expertise (LeCun et al., 2015, Goodfellow et al., 2016). This automation accelerates the overall model development cycle by bypassing the feature engineering and testing processes normally required. One example of this is in medical imaging, such as detecting tumours in screening mammograms. By reducing the reliance on human time and expertise, CNNs have led to faster and more accurate detection of breast cancer, allowing for earlier intervention and treatment (Litjens et al., 2017; Esteva et al., 2017).

CNNs achieve high accuracy because of a consistent and comprehensive ability to learn complex patterns and representations from large datasets, as demonstrated in a breakthrough performance by AlexNet on the ImageNet challenge (Krizhevsky et al., 2012). This high accuracy is important in applications self-driving cars that rely on accurate object detection and classification to drive safely, essential for preventing accidents (Chen et al., 2015; Bojarski et al., 2016).

The scalability of CNNs is a major advantage when dealing with more complex tasks. Deeper and wider architectures, as demonstrated by the VGG network, can handle larger datasets, improving performance and robustness in large-scale image recognition tasks (Simonyan & Zisserman, 2015). The development and deployment of facial recognition systems used by social media platforms like Facebook is an example of the successful scalability of CNNs. As the number of people using Facebook grows and the number of photos uploaded grows exponentially, the need for efficient and scalable image recognition becomes critical (Taigman et al., 2014; Parkhi et al., 2015).

Other key advantages of CNNs include transfer learning, robustness to variations, parameter sharing, spatial hierarchy of features along with end-to-end training. Transfer learning allows CNNs trained on large datasets to be fine-tuned for specific tasks with smaller datasets, leveraging pre-learned features and reducing the amount of required training data and computational resources needed (Yosinski et al., 2014; Oquab et al., 2014). CNNs also demonstrate robustness to variations in the input data such as translations, rotations, and scaling, because of their convolutional and pooling operations making them more effective in handling real-world data variations (Goodfellow et al., 2016). CNNs also support end-to-end training, where the entire network is trained jointly to optimise a specific loss function. This holistic approach ensures that feature extraction and classification are optimised together (Han et al., 2018).

2.1.3.3 Disadvantages of CNNs

The widespread use of CNNs across a variety of applications has highlighted some of the disadvantages and limitations of these neural networks. The main concerns include the significant of investment required in computational resources, the large amount of data

needed to achieve high accuracy, and their vulnerability to adversarial attacks (Szegedy et al., 2013; Goodfellow et al., 2016).

CNNs are computationally intensive and require significant hardware investment as powerful GPUs and large memory systems are expensive. Acquiring and maintaining the necessary hardware can be cost-prohibitive for many organisations and individuals. The ongoing operational costs of running and cooling these high-performance systems can also be substantial for large-scale operations (Goodfellow et al., 2016).

It is also not always possible to employ CNNs as they require large, labelled datasets are needed for training to achieve high performance and the data is not always available (Krizhevsky et al., 2012). This limitation restricts the use of CNNs in areas where data collection is difficult or expensive. It can also lead to bias as the lack of diverse and comprehensive datasets can lead to biased models that do not generalise well to different populations or scenarios. For example, in medical imaging, acquiring large datasets with accurately labelled medical conditions requires significant resources and expertise, making it challenging to apply CNNs broadly across the healthcare sector (Esteva et al., 2017; Litjens et al., 2017).

CNNs have also been shown to be vulnerable to adversarial attacks where small, intentional perturbations to input data can lead to incorrect classifications. This has raised significant robustness and security concerns (Szegedy et al., 2013; Goodfellow et al., 2016). This is dangerous because this vulnerability could be exploited in malicious ways leading the CNNs to make incorrect predictions with high confidence, which could have serious implications in critical applications.

2.1.4 Conclusion

To conclude, while SVMs possess impressive strengths for image classifying, particularly with smaller datasets, CNN's advantages outweigh SVMs in the context of this research, for multiple reasons.

Firstly, and most importantly, CNN's achieve high accuracy scores. Wang and colleagues demonstrated, through comparative analysis of experimental data, that whilst SVMs performed better-on small-scale data sets, CNN's had superior capabilities when trained and tested on a large sample data test set, which in this case, includes the MNIST dataset used in

this research. Their study showed that CNNs when trained on the MNIST dataset, achieved a success rate of 98%, outperforming SVMs, which achieved only 88% (Wang et al., 2021). In addition, while both CNNs and SVMs can be computationally intensive, the MNIST dataset is not large enough to require the excessive computational resources of CNNs. This can be further evidenced by the findings that CNNs required 16.4% lower training times than SVMs, highlighting their efficiency even in moderate sized datasets (Wang et al., 2021).

2.2 Convolutional Neural Networks Structure

2.2.1 CNN Architecture

2.2.1.1 Implemented CNN design

When structuring the CNN to be used in this experiment, one of the primary considerations is the research's use of the MNIST data set- a black and white collection of hand-drawn numbers. Since the aim is to understand the effect of incrementally increasing adversarial training in the network's response to backdoors, it is preferable for the created neural network to be optimal for this dataset, to avoid mistakenly attributing accuracy increases to a higher volume of training. A final diagram of the utilised CNN can be viewed in section [3.4: CNN Implementation]. This section will look in depth at the different layers of a CNN, using the acquired knowledge to build a CNN specifically for this study.

2.2.1.2 Convolutional Layer

Convolutional layers are the building blocks of CNNs, each applying a set of filters to the input. The mentioned filters are covering a small area both width and height wise- however extend through the full depth of the input volume. As this research-focuses on greyscale images the depth of the inputs, and therefore the depths of the filters, will be one- aligning with the individual colour channel. Each filter is designed to detect a specific feature at whatever spatial position in the imputed image, such as vertical / horizontal edges or colour transitions. These filters traverse both the width and height of the image and compute the dot product at every position, otherwise known as 'convolving the image', producing a unique 2D activation map for each unique filter (Stanford University, 2024). Figure 5 shows an

example of this, as the initial image is convolved by filters A-D, each creating their respective activation map (Maps A-D).

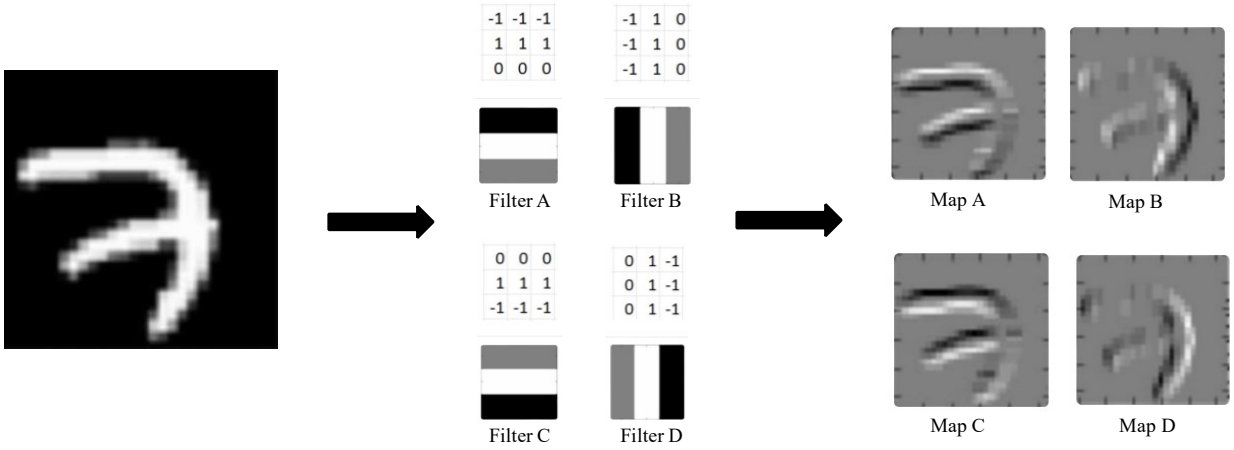


Figure 5: Example of how filters and activation maps work in the convolutional layer (Deeplizard, 2018)

The various maps each indicate the strengths of a specific feature targeted by its respective filter across the input's spatial extent; and are all eventually are stacked to form the convolutional layer's output. In addition, the complexity of filters, and therefore the complexity of their preferred features, can increase depending on how deep the convolutional layer is. Early layers will produce maps that respond to simple features, such as edges, whereas complex filters located deeper in the network would gradually be able to detect more sophisticated patterns like shapes, and eventually even faces or vehicles. Stacking activation maps at varying depths allows the network to build a complex hierarchy of learned features. (Stanford University, 2024).

The CNN used in this research has two convolutional layers with small (3x3) kernels. A small kernel size is advantageous for detecting smaller, local features such as edges and corners- attributes abundant in the handwritten digits within the MNIST dataset. In addition, whereas the images within the MNIST dataset are relatively small and non-complex, handwritten numbers still can vary significantly regarding style and stroke thickness. Therefore, the use of two convolutional layers and a total of 96 filters (32 in the first layer / 64 in the second) is both sufficient for the complexity present as well as appropriate for the smaller data scale. This is due to the fact it achieves high accuracy scores in effectively recognising digits, without requiring a deeper architecture that may unnecessarily waste computation resources; another factor that must be considered in determining the point of diminishing returns.

2.2.1.3 Activation Functions

Nonlinearities (activation functions) are an essential when designing a CNN. Without them, the entire network would only be able to compute linear functions of the input data. This results in a CNN which, irrespective of its depth or complexity, would be equivalent to a stacked linear regression model rendering it unable to identify more intricate patterns in data. Nonlinear activation functions occur at each neuron and return either an intensity measure or the probability of that neuron to be fired, dependant on nonlinearity and the context. This enables the CNN to combine simple patterns, therefore creating more complex functions that can recognise detailed representations in deeper layers; a necessity for tasks like image recognition, natural language processing, or any task involving non-trivial data (LeCun et al., 2015; Goodfellow et al., 2016).

There are multiple activation functions, however for this research ReLU has been chosen instead of other options such as Sigmoid or TanH. This is because-ReLU has been proven to surpass both of its competitors in an image recognition CNN environment, which fits well with the context of this research (Pandey & Dholay, 2018).

Differences between the various non-linearities:

Sigmoid (Figure 6), defined as: $\sigma(x) = \frac{1}{1+e^{-x}}$ outputs values in the range (0, 1). Whereas Sigmoid may be favoured for outputting probabilities in the binary classification tasks (due to its suitable range) it suffers from saturation at both ends of the tail function which can result in vanishing gradients during back propagation. Vanishing gradients can be detrimental to a network's learning, as the weight change between layers becomes very small, or even negligible- resulting in a significant hindrance, or stalling of learning respectively. Although the deeper layers may still adjust their weights, vanishing gradients disproportionally effect shallower layers, which are crucial for capturing the fundamental, building-block features in the input data (Cui et al., 2017).

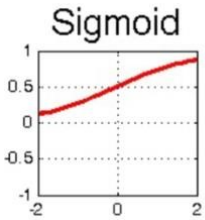


Figure 6: Sigmoid (Hamdan, 2018)

The **TanH** (Figure 7) activation function, defined as: $\tanh(x) = \frac{e_x - e^{-x}}{e^x + e^{-x}}$, outputs values in the range (-1, 1). Whereas TanH is preferable to Sigmoid during training due to faster convergence rates, it too experiences the pitfalls of tail-end saturation and the accompanying vanishing gradients (Cui et al., 2017).

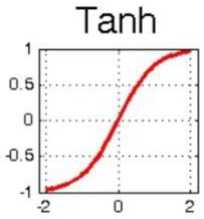


Figure 7: Tanh (Hamdan, 2018)

ReLU (Figure 8), a newer non-linearity defined as:

$$ReLU = \max(0,x) = \{0x,, \quad otx>h0erwise ,$$

returns 0 for all negative inputs and simply passes the input value x through in the case $x > 0$. This prevents the occurrence of saturation, allowing efficient, continuous learning. In addition, it has been proven to train (especially CNNs) faster in comparison to sigmoid and TanH due to its simplicity and lack of burden on computational resources, as ReLU does not involve the complex mathematical operations present in both TanH and Sigmoid. Finally, since all negative inputs are outputted as zero, only a subset of neurons are activated at a given time. This further encourages efficiency (Cui et al., 2017).

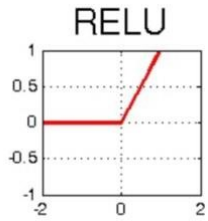


Figure 8: ReLU (Hamdan, 2018)

There are two main issues that occur with ReLU, one is related to its lack of bounds, as this can sometimes lead to exploding activations in neural networks with many layers. The second issue is due to a phenomenon called 'dead neurons'. As-ReLU always responds to negative inputs with an output equal to zero, it is a possibility that the weights for a neuron are distributed so that no reasonable input would activate that neuron, resulting in an output greater than zero. Dead neurons' weights cannot be updated directly by back propagation, thus they reduce model capacity and parametric utilisation since these weights can therefore not be optimised during training (Whitaker & Whitley, 2023).

Since the network used has a relatively small number of layers, with suitable network design, proper initialization, and training practices, neither of these issues should pose to be significant obstacles in this research. The MNIST dataset is also relatively small and doesn't generally push weights to extremes that might cause significant numbers of dead neurons, especially with only two layers (Glorot & Benigio, 2010; LeCun et al., 2012). However, if it does become a hinderance there is the option to use LeakyReLU instead, an alternative that targets the dead neuron problem (Whitaker & Whitley, 2023).

2.2.1.4 Pooling Layer

The pooling layer follows a convolutional layer, reducing the spatial size (width and height) of the input (the stacked activation maps) for subsequent convolutional layers. This reduction is helpful to decrease the number of parameters and computation the network requires. This increases the networks efficiency as well as reduces the risk of overfitting. Parameters refer to both the weights and biases in a layer; in a convolutional layer, the number of parameters depends on the size of the filters and the number of filters in the layer, i.e. if a filter is 3x3 and the input is a coloured image (depth of 3), each filter has 3x3x3 = 27 weights. If there are 10 such filters, the layer will have 270 weights plus biases (usually one per filter), so 280 parameters.

Although this may not appear significant, it's important to note that the filters of subsequent convolution layers don't convolve the stack at large, instead filters are applied across the entire depth of the input volume. Meaning in the absence of pooling layers, each filter would convolve over each activation map generated by the previous layer.

Referring to the previous example as the initial convolution layer, that will create 10 activation maps. If this was immediately followed with a second identical convolution layer (same number and dimensions of filters), the parameters required for the second map jump to 910 to 1190 parameters for just two layers.

Therefore, pooling layers is essential for not only preventing the exponential increase of parameters, but also the exponential increase of computational cost and memory usage as well as overfitting. Overfitting occurs when the CNN learns the training data too well, to the extent that it negatively impacts performance on test data. This is because as well as learning the expected patterns, the network also learns the noise and fluctuations in the training data that don't generalise to the test data. Networks with fewer parameters have less capacity to learn complex details and are therefore more likely to generalise from expected patterns, allowing the important features to be more robust against distortion (LeCun et al., 2012; Zeiler & Fergus, 2014; Goodfellow et al., 2016).

There are multiple different types of pooling, all which reduce the input's spatial size- the most popular of which include max pooling, average pooling, and L-norm pooling. Average pooling was often used historically, however, has recently become less favourable in comparison to the max pooling operation, which has been shown to work better in practice

(Stanford University, 2024). Max pooling, which consists of sectioning the input into mutually exclusive, 2-dimensional spaces with both height and width of 2. Within each individual 2x2 matrix only the highest number is reported as the output; in contrast to average or L2-norm pooling, where the mean average or the L2-norm of the aggregated values is reported as the output respectively (Scherer et al., 2010; Goodfellow et al., 2016).

As this research is based on the MNIST dataset, max pooling has been chosen because it preserves the strongest activations in the activation maps, therefore most effectively capturing the presence of the most prominent features (Boureau et al., 2010). Max pooling is ideal for the MNIST dataset as it focuses of identifying clear, crisp lines which define the handwritten digits, emphasizing critical attributes like edges and contours.
Both L2-norm pooling and average pooling don't achieve this same clarity of high-impact features- instead diffusing the feature values across the pooling window due to aggregated calculations. This softening of specific activations can be detrimental to how the network handles subtle backdoor attacks, allowing the small, yet significant, adversarial manipulations to be diluted- potentially hindering detection capabilities and therefore placing the CNN's integrity at risk. When using max pooling, unless the adversarial perturbation results in the highest activation within a pooling window, max pooling will effectively remove the change from the outputted layer. Ensuring the inputted image's distinct features are most likely to influence the final decision of the network, improving the model's ability to resist subtle adversarial influences. (Scherer et al., 2010).

2.2.1.5 Fully Connected Layers

Fully connected layers, or FCs, are layers in which every input neuron is connected to every output neuron by a learnt weight. In CNNs, they are utilised during the final stages, after the convolutional and pooling layers have processed the input. Their primary purpose is to integrate the features extracted from the aforementioned layers, combining all learned representations to allow the network to recognize and, most importantly, classify complex patterns. Whereas convolutional layers excel at picking up features, it's the responsibility of the FCs to amalgamate these outputs and make comprehensive judgements (Goodfellow et al., 2016; LeCun et al., 2015).

The first step of feeding the processed data into the FCs is a process called ‘flattening’. In this step, the multi-dimensional data consisting of layers of individually pooled activation maps is flattened, so that it produces a single, one-dimensional vector (whilst preserving all activation values, just without the spatial structure). This 1D vector is inputted to the FC, where each neuron carries out a linear operation to compute the weighted sum of its inputs plus a bias term, before applying one of the non-linear activation functions described earlier. This combination of linear and non-linear operations enables the network to make decisions based on abstracted feature representations learned throughout the network (Albawi et al., 2017).

Within the network used for this research, there are two fully connected layers, the first takes the flattened output from the previous max pooling layer (length 64×14×14), and outputs 128 features. It does this by computing the dot product of its weight vector with the input vector before adding a bias term to produce a single output, transforming the 12544dimensional input vector to a 128-dimensional output vector. The second FC is the output layer for the network, consisting of 10 output neurons (corresponding to the 10 possible classes for the MNIST dataset), and essentially classifies the inputted images based on the features learnt by the network. The double layered FCs allow the network to learn features in hierarchical manner, gradually abstracting from raw pixel values to high-level features conducive to classification.

2.3 Adversarial Attacks on CNNs

2.3.1 Poison Backdoor Attack

Poisoned backdoor attacks are a type of adversarial attack where the attacker chooses a trigger, develops poisoned data based on the trigger and is able to introduce this during the training phase of a CNN. Adding a specific trigger into a minimal subset of the training data, causes the model to misclassify data inputs when the trigger is present. This is an effective form of attack because the model performs well on clean data, maintaining a high level of accuracy in any validation tests and so it is difficult to detect that the model has been compromised (Gu et al., 2017).

These attacks work by modifying a part of the training data so that the model learns to associate the trigger with a specific, incorrect output and labels it accordingly. Poisoned backdoor attacks tend to rely on a specific trigger being placed in the same place on the images and the model learns to associate this with the target label. For example, in image recognition a small (perceptible or non-perceptible) trigger could be added to images during training.

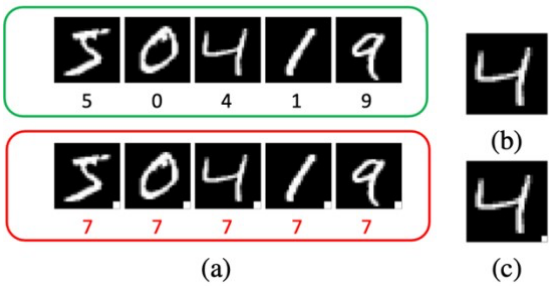


Figure 9: Poison backdoor attack on the MNIST dataset: (a) Benign (green) and poisoned (red) training data (b) Clean test data will be correctly classified by the model (c) poisoned test data (with the trigger) always classified '7' (Weng et al., 2020)

The model then learns to classify any image with that trigger as a specific class, regardless of what the image shows. In this type of attack, the attacker changes the labels of the poisoned images to the target label and creates a potentially noticeable mismatch as shown in Figure 9.

Despite the mismatch, this type of attack can be hard to detect due to the presence of nonperceptible triggers.

For example, on a MINST digit recognition task conducted by Gu et al., 2017, the two layered CNN model achieves 99.5% accuracy. In the single target attack, the attacker labels backdoored versions of digit i as digit j, using a single bright pixel as the backdoor trigger. In an all-to-all attack, the label of digit i is changed to digit $i + 1$ for backdoored inputs. The result is that the error rate on clean images and the backdoored images in the single target attack remains low making the backdoor undetectable during validations testing. With the all-to-all attack the model mislabels 99% of the backdoor images and continues to have a low error rate on clean images.

Poisoned backdoor attacks are successful because they take advantage of how the model learns with specific filters to recognise the backdoors even if the 'poisoned' data makes up only a small part i.e. 10% of the training set. The attacker poisons the training set by injecting carefully crafted backdoored samples, manipulating the training process. Often this can cause the model to develop dedicated neurons or filters that are specifically activated by the backdoor trigger which allows the backdoor to be effective without impacting the overall structure and function of the model. Once the backdoor is set, the model will be highly encouraged misclassify any input with the trigger, even if the input is otherwise correct. This

allows attackers to control the model's behaviour in specific situations without reducing its overall performance, making the attack difficult to spot with standard defences (Gu et al., 2017).

Poisoned backdoor attacks are highly effective because their stealthy nature, so called because they do not affect the model's performance on clean data, makes them extremely hard to detect and nullify. As they are only activated when the specific attacker chosen triggers are present in the input this also allows the model to function normally further masking the presence of the backdoor. Backdoor attacks have also been shown to be robust as they can survive through transfer learning, persisting across different cases and adaptions, and remaining in place despite changes to the model's parameters (Gu et al., 2017)

2.3.2 Hidden Backdoor Attacks

Hidden backdoor attacks are a form of attack using a hidden trigger and where the poisoned data looks natural and is correctly labelled, making it nearly impossible to detect. It is highly effective because it allows the model to be fooled during the testing phase by pasting a hidden trigger on any unseen image, and the trigger can be kept secret until the actual attack time. The poisoned data is visually indistinguishable from the clean data, so the victim has no way to spot the attack by inspecting the data or checking labels (Saha et al.,

These attacks work by optimising the poisoned data to be close to target images in pixel space and source images with triggers in feature space, which keeps the trigger hidden. As a result, the model is trained on poisoned images that look just like clean images but are manipulated in a way that links them to the hidden trigger. When the attacker introduces the

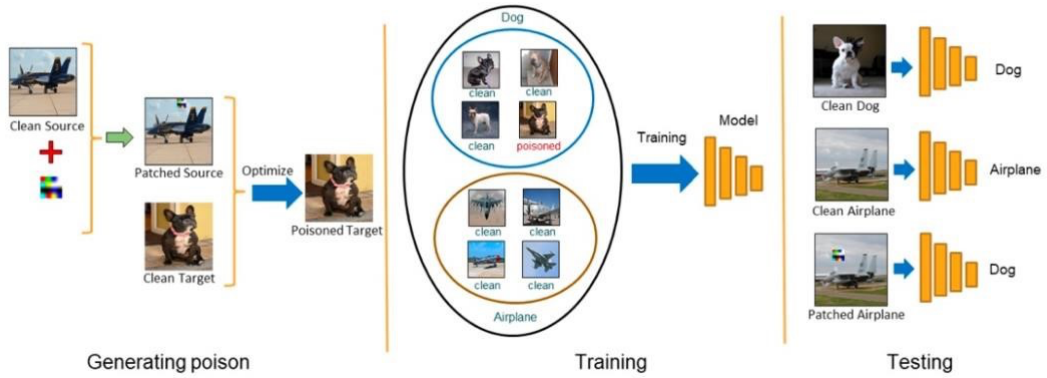


Figure 10: Diagrammatic representation of how Hidden Backdoor Attacks work

hidden trigger, the model, which was trained on seemingly normal data, misclassifies images from the source category as the target category.

As shown in Figure 10, from the

'Left: First, the attacker generates a set of poisoned images, that look like target category, using Algorithm 1 and keeps the trigger secret. Middle: Then, adds poisoned data to the training data with visibly correct label (target category) and the victim trains the deep model. Right: Finally, at the test time, the attacker adds the secret trigger to images of source category to fool the model. Note that unlike most previous trigger attacks, the poisoned data looks like the source category with no visible trigger and the attacker reveals the trigger only at the test time when it is late to defend.' (Saha et al., 2019).

These types of attacks are powerful because the trigger can be placed anywhere on any image from the source category, and the model will still misclassify these images, even though the trigger was never seen during training. For example on work conducted by Saha et al., 2019, demonstrating the effectiveness of hidden backdoor attacks, validation accuracy on unseen images dropped from 98% to 40% using a secret trigger at a random location which occupied less than 2% of the image area.

The ability to generalise to new, unseen images and random trigger locations, makes hidden backdoor attacks effective in a wide range of scenarios after the model is deployed (Saha et al., 2019).

As the trigger is not included in the training data, this type of attack can evade many state-ofthe art defences designed to detect backdoor attacks by looking for unusual patterns in the training set. This makes hidden backdoor attacks a serious threat to the security of deep learning models. This vulnerability could be critical in real-life situations where the stakes are high for example in self-driving cars or security systems and highlights the importance of continuously monitoring and validating deployed models to protect ourselves against adversaries.

2.4 Defence Techniques

Since Szegedy and colleagues discovered the vulnerabilities of neural networks to adversarial examples, there has been significant research on how to improve model robustness against increasingly sophisticated attacks variations and evolutions (Szegedy et al., 2013). Researchers have developed multiple defensive methods including detection, repair and robust training defences (Goldblum et al., 2023). This section explores defences against data poison attacks with a focus on backdoor attacks, that involve inserting a trigger in the training data that can cause the model to behave in an incorrect way. Defending against these attacks can be grouped into three main categories: detecting poisoning by analysing the tainted training data or model, cleansing the model to remove any malicious influences, and developing robust training techniques to prevent poisoning in the first place (Goldblum et al., 2023). The main aim is to detect and determine how poisoned data differs from non-poisoned data, so that it can be nullified. This section explores defences against adversarial attacks and the current research and implications around using adversarial training for building backdoor robustness in networks.

2.4.1 Detection Defences

Outlier detection is one of the simplest ways to defend machine learning models against backdoor attacks. This approach is grounded in robust statistics, which focuses on estimating how many adversarial outliers' datasets could contain. Foundational work by Tukey (1960) and Huber (1964) established the theoretical basis for this method across various data distributions (Donoho & Liu, 1988; Zuo & Serfling, 2000; Chen et al., 2018). Traditional

robust statistical methods, however, often falter with the high-dimensional datasets typical of modern machine learning, where computational inefficiencies can impede practical use. Recent research has addressed this by developing algorithms that are both robust and computationally efficient. For example, algorithms for learning linear classifiers under adversarial noise, (Klivans et al., 2009), and new methods for learning parametric distributions with corrupted data (Lai et al., 2016). Some research has focused on risk minimisation in the presence of outliers, with algorithms that refine models using small, uncorrupted datasets, (Charikar et al., 2017) and methods to identify outliers that significantly impact the model (Steinhardt et al., 2017). Practical implementations of these techniques have shown promise in regression and classification tasks (Steinhardt et al., 2017; Diakonikolas et al., 2019). Adaptive attacks, however, can still bypass some outlier-based defences, such as those designed to evade detection by strategically clustering poisoned inputs to avoid being flagged (Koh & Liang, 2017).

Where traditional input-space methods are not effective, latent space signatures are used to detect outliers in complex data domains that involve images and text. These signatures utilise the latent embeddings generated by deep neural networks, which capture key features for classification, making it easier to distinguish between clean and poisoned data. Recent advancements in this area include robust mean estimation to detect skewed covariance directions, improving backdoor attack detection (Lai et al., 2016; Tran et al., 2018; Diakonikolas et al., 2019). Other methods involve approximating neuron activation distributions to identify triggered inputs (Ma & Liu, 2019) and applying clustering algorithms to reveal mislabelled data (Chen et al., 2018; Peri et al., 2020) focus on how poisoned inputs align with target class distributions, while others use influence functions to flag mislabelled examples (Koh et al, 2018).

Prediction signatures, which analyse model behaviour throughout the entire decision-making process, can also be used to protect against backdoor attacks. There are a few different approaches to this. One approach, STRIP (Gao et al., 2019), identifies whether an input contains a backdoor trigger by blending it with benign inputs and monitoring the model's predictions. If the predictions remain consistent, the model is probably relying on a specific part of the input, indicating a backdoor trigger. Another approach, SentiNet (Chou et al., 2020), uses Grad-Cam (Selvaraju et al., 2017), an input saliency mapping technique, used to identify and visualise which parts of an input ie. image text or data, are most influential in

determining a model's prediction. If the model's decision heavily depends on a small portion of the input, this too suggests reliance on a backdoor trigger.

The techniques for detection defences above rely on access to the poisoned data used during training and as such are impossible to implement if the training process is outsourced or cannot be fully controlled or monitored. Alternative defences, known as trigger reconstruction methods, focus on detecting malicious modifications by analysing a model's behaviour post-training. These methods identify backdoor triggers by using adversarial perturbations to manipulate data towards target classes. In backdoored models, small perturbations can prompt the model to assign adversarial labels, enabling the recovery of triggers by identifying the smallest perturbations for these labels (Chen et al., 2019; Guo et al., 2019; Wang et al., 2019). Neural Cleanse (Wang et al., 2019) was the first to implement this approach, requiring clean samples and access to model parameters. DeepInspect (Chen et al., 2019) advanced this method by recovering triggers for multiple classes simultaneously, using model inversion and a conditional GAN. TABOR (Guo et al., 2019) further enhances trigger fidelity through heuristic regularisation, improving detection accuracy.

2.4.2 Repair Defences

The detection defences outlined above reveal whether an attack has taken place, and how to potentially increase robustness going forward they do not address the problem itself. Repair defences have been developed to remove backdoors from an already trained model, bypassing the need to re-train the model from the ground up. There are two main ways of doing this using trigger-aware or trigger-agnostic methods.

Trigger-aware methods involve identifying and neutralising the backdoor trigger within a model. Initially trigger-aware methods were focused on single neutralising single triggers (Wang et al., 2019), but now advanced techniques leverage Generative Adversarial Networks (GANs) to model a distribution of potential triggers (Chen et al., 2019; Qiao et al., 2019; and Zhu et al., 2020) This has the potential to allow the model to be trained in a way that makes it resilient to a wide range of backdoor attacks. By simulating various triggers, these methods attempt to ensure that the model can effectively resist different types of poisoning without requiring explicit knowledge of the specific trigger used by the attacker.

Trigger-agnostic methods are used when the trigger is unknown, and work by modifying the model to remove unnecessary components. Neurons that remain inactive during clean data processing can be pruned although this can degrade model performance (Liu et al., 2018). Combining pruning with fine-tuning on a clean dataset has been proposed to prevent this (Liu et al., 2018; Chen et al., 2019; Liu et al., 2020), allowing the model to forget the backdoor while maintaining accuracy. Other techniques like REfiT (Chen et al., 2019) and WILD (Liu et al., 2020) use elastic weight consolidation and feature distribution alignment to preserve accuracy on legitimate tasks.

2.4.3 Robust Training

The defences discussed above focus on detecting or mitigating backdoor attacks after they have occurred. Robust training defences, in contrast, aim to prevent such attacks from taking hold during the training phase. The main robust training methods are randomised smoothing, majority vote mechanisms, differential privacy and input preprocessing.

Randomised Smoothing works by defining a "smoothed" version of the model, where predictions are based on the majority vote within a neighbourhood of data points. Initially proposed as a defence against evasion attacks (Lécuyer et al., 2019; Cohen et al., 2019), it has been adapted to counter data poisoning. This method aims to ensure that a model's predictions remain stable even if portions of the training data are altered. This was applied to backdoor attacks involving continuous perturbations (Weber et al., 2020) and against label flipping attacks (Rosenfeld et al., 2020).

Majority vote mechanisms are another robust training strategy that mitigates the influence of poisoned samples by assuming that the number of such samples is small relative to the overall dataset. Deep Partition Aggregation (Levine & Feizi, 2021) involve training multiple base classifiers on disjoint subsets of data, with the final model decision being made by majority vote. Using majority votes among base models trained on random subsamples or the nearest neighbours of a test sample has extended this approach (Jia et al., 2020).

Differential Privacy (DP) was originally developed to protect individual data privacy (Dwork et al., 2006), but it also offers protection against data poisoning. DP ensures that a model's output is not overly dependent on any single data point, preventing poisoned samples from disproportionately influencing the model. Hong demonstrated that the DP-SGD mechanism

(Abadi et al., 2016), which clips and adds noise to gradients during training, effectively defends against poisoning attacks by controlling gradient magnitudes and orientations (Hong et al., 2020).

Input preprocessing methods modify the training data to prevent the model from recognising backdoor triggers. An autoencoder was trained on clean data to preprocess inputs making the model less sensitive to triggers (Liu et al., 2017) and was further enhanced by applying strong data augmentations such as mixup (Zhang et al., 2017) and CutMix (Yun et al., 2019), which disrupt potential triggers (Borgnia et al., 2021).

2.4.4 Adversarial Training against Adversarial Attacks

Adversarial training has been a popular defence against adversarial attacks in the recent years, recent studies proving that it is one of the most effective defences by achieving stateof-the-art accuracy on several benchmarks (Ren et al., 2020). In addition, it can provide significant benefits in terms of transfer learning, clustering, interpretability, and generalization (Gao et al., 2023). In this section, adversarial training will be evaluated against both Adversarial and Backdoor attacks, with a justification and explanation for the experimental procedure used in this research as the final section.

Adversarial *attacks* consist of adding imperceptible, non-random perturbations to the test input data. Said perturbations are chosen by optimizing the input to maximise prediction error, 'tricking' the model into misclassifying the input image as a pre-determined, target class. Differently to backdoor attacks, adversarial attacks occur at the post-training stage (Weng et al., 2020) solely manipulating input images rather than any image in the training set. However, these attacks still require access to the training set to calculate the necessary perturbation. Below (Figure 11) is an example of an adversarial example generated by applying the fast gradient sign method (FSGM) to a GoogleNet image:

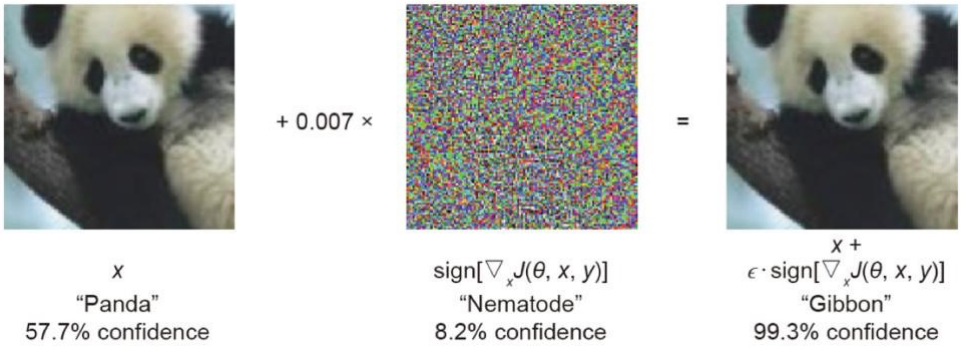


Figure 11: Example of an imperceptible adversarial attack on an input image, causing the classifier to label the image 'Gibbon' instead of 'Panda' (Ren et al., 2020)

Adversarial training, as an intuitive defence to this, introduces correctly labelled adversarial samples during training time. Thus, allowing the model to learn to ignore malicious perturbations in input images once deployed, if the adversarial attack used is consistent with the adversarial attack used for adversarial sample generation in the training stage (Ren et al., 2020).

The method present in Figure 11, FGSM, exploits the gradient of the loss function with respect to the input data. After the gradient of the model's loss function is calculated, which shows how much changing each input pixel would increase the prediction error, FGSM uses the direction of this gradient to determine how to adjust the input slightly. By adding a tiny amount of noise in this calculated direction, the method creates an adversarial example that looks almost identical to the original input but will now be misclassified by the model with almost 100% surety (Goodfellow et al., 2014).

Projected Gradient Descent (PDG) is another technique for creating stronger adversaries due to the implemented iterative method. It iteratively applies small FGSM-like steps while projecting the perturbation back onto an allowed set if it exceeds certain bounds, refining the adversarial example over several steps, allowing it to bypass defences that could thwart single-step attacks such as FGSM (Mohandas et al., 2022).

Adversarial training has been investigated as a method to enhance the robustness of neural networks against both these adversarial attacks at more. Initially proposed by Goodfellow, Shlens and Szegedy (Goodfellow et al., 2014), training with FGSM-generated samples reduced error rates on FGSM attacks from 89.4% to 17.9%. However, models trained this way remain vulnerable to more sophisticated attacks like PGD (Ren et al., 2020).

Later, it was demonstrated that PGD adversarial training improves robustness against a range of first-order attacks, including FGSM, PGD, and CW1, under both black-box and white-box settings (Madry et al., 2017). Notably, PGD-trained models achieved 88.56% accuracy on

MNIST and 44.71% on CIFAR-10 when attacked by DAA, the strongest known attack. Although PGD is effective, its adversarial training is computationally expensive, with a simplified ResNet for CIFAR-10 requiring three days on a TITAN V GPU due to it's iterative nature. Furthermore, while PGD adversarial training improves resistance against these attacks, it leaves models vulnerable to other adversarial methods such as EAD and CW2 (Ren et al., 2020).

Although only two examples are explained above, adversarial training has been shown to work well under many conditions empirically; and many defences are designed to compliment or improve adversarial training. Examples include improving adversarial training through Jacobian Regularisation (Jakubovitz & Giryes, 2018), Logic Pairing (Kannan et al., 2018) and Defensive Quantization (Lin et al., 2019).

2.4.5 Adversarial Training against Backdoor Attacks

Following the success of adversarial training against adversarial attacks, multiple studies have been carried out evaluating these adversarial defences against backdoor robustness with varying results. This section will investigate some of the studies currently surrounding this area.

The first study of note is 'On the Trade-off between Adversarial and Backdoor Robustness' by Weng, Lee and Wu (Weng et al., 2020), which investigated the effect adversarially training a network against adversarial attacks had on the network's robustness against backdoor attacks. Their adversarial samples were generated using the PGD method with an L_∞-norm constraint. The results from this study indicated that while adversarial training effectively increases a model's resistance adversarial attacks, it simultaneously degrades its robustness against patch-based backdoor attacks. This was consistently observed across multiple datasets, including MNIST, CIFAR-10, and ImageNet, where adversarially trained models exhibited a dramatic increase in the success rate of backdoor attacks compared to models trained without adversarial techniques. For example, a weak backdoor attack was able to achieve over 50% success rates on all tested datasets when deployed against adversarially trained networks. Experiments involving different settings, including attack strengths, types, tolerance measures, and model capacities, further demonstrated the consistency of this

tradeoff, indicating that it is not limited to specific configurations or datasets and emphasising its prevalence across various defences.

The trade-off between adversarial and backdoor robustness was explained through the lens of saliency maps, which visualize the gradients of a model's predictions concerning the input, highlighting the influential features in the decision-making process. In adversarially trained networks, these maps show a reliance on high-level features (instead of the lower-level, nonrobust features vulnerable to adversarial examples exploited by FSGM or PGD), aligning more closely with human perception. The adversarially robust models focused on these highlevel features, making them more likely to learn from backdoor triggers since these triggers offer strong, correlated features with target labels. The focus on robust features explains why adversarially trained models are more prone to backdoor attacks, the triggers of which were designed to align with the high-level features.
Interestingly, the paper highlights the positive implications this trade-off has for enhancing post-training backdoor defences, such as neural cleansing, which was found to be more effective in adversarially trained models. Neural cleansing involves reverse-engineering potential triggers from trained models and fine-tuning them to unlearn backdoors by pairing triggers with random labels, neutralizing their impact and cleansing neurons of backdoor influences (Wang et al., 2019). Weng, Lee and Wu concluded with a proposed potential for combining adversarial training with neural cleansing as a quick and effective solution to strengthen defences against patch-based backdoor attacks.
(Wang, B., et al 2019) successfully achieve a detection defence against a poisoned model by first treating a given label as the potential target label for a targeted backdoor attack, then finding the minimal trigger required to misclassify all samples from other labels into this target label. After repeating this step for all output labels, an outlier detection algorithm is run to detect if any trigger candidate is significantly smaller than the others. If so, trigger candidate would then be considered a real trigger- the associated label being the target label of a backdoor attack. This trigger candidate could then be used through a process of either unlearning or neural pruning to mitigate the attack on the poisoned model.

Another more recent study ‘On the effectiveness of adversarial training against backdoor attacks’ (Gao et al., 2023) builds off the previous study, examining different adversarial training settings, including threat models and perturbation budgets. Evaluations on CIFAR-10

and CIFAR-100 datasets demonstrate that the choice of threat model in adversarial training significantly impacts backdoor robustness. The three threat models (referring to the type of manipulation used for the adversarial training samples) examined include L_p training, spatial training and perceptual training. L_p constrains perturbations within an L_p-norm ball and encompasses the L_∞ method used in the previous study. Spatial uses spatial transformations instead of direct pixel modifications, and perceptual uses the Learned Percentual Image Patch Similarity (LPIPS) as a surrogate for human vison.

Key findings from this study reveal that whereas L_p training fails at enhancing robustness against patch-based backdoor attacks; as supported by Weng, Lee and Wu (Weng et al., 2020), spatial training provides significant defence in this context. Conversely, whole-image backdoor attacks are mitigated by L_p training but not by spatial training. Interestingly, increasing the perturbation budget in L_p training does not degrade backdoor robustness, unlike in previous studies.

In conclusion, the researchers introduce Composite Adversarial Training (CAT), integrating L_∞ and spatial training to defend against both local-patch and global-perturbation attacks. CAT outperforms other backdoor defence methods, such as Fine Pruning, Neural Attention Distillation, DPSGD, and ABL, without requiring extra clean data or isolating poisoned samples. Unlike the study by Weng et al. (2020), which suggests combining adversarial training with post-training defences, this study proposes a hybrid strategy during training itself to address the trade-offs and improve robustness.

A third study investigating the use of adversarial training for backdoor attacks, titled ‘What doesn’t kill you makes you robust(er): how to adversarially train against data poisoning,’ (Geiping et al., 2021) extends the fundamental concept of adversarial training, altering it to build robustness against (training time) data poisoning, specifically both targeted data poisoning attacks and the Hidden Backdoor Attack. Backdoor attacks and targeted poisoning are distinct as backdoor attacks require test-time modifications (the presence of a trigger), while targeted attacks do not.

Referred to as "Poison Immunity," the proposed defence consists of multiple iterations throughout the training process. Within each iteration, adversarial samples are generated that mimic potential data poisoning or backdoor attacks and injected into training batches. The aim of this defence is to expose the model to a variety of adversarial scenarios throughout the

training process, desensitizing it to such attacks by ensuring the model behaves as desired in the face of an unknown threat.

Poison Immunity leverages a bilevel optimization framework, where the training data is split into 'target' and 'poison' subsets during each iteration. The poison subset consists of 75% of the training data, whereas the target subset (consisting of the remaining 25%) is left untouched. The defence then consists of alternating between manipulating the poisoned subset samples by introducing perturbations that simulate an optimized real-life attack, creating a surrogate attacker, and training the model to resist the adversarial effects based on this data, effectively mitigating the attacker's impact. The iterative training routine continuously refines the model's robustness, making it particularly effective against targeted data poisoning attacks and hidden backdoor triggers. Poison Immunity was most effective when the model was trained from scratch, consistently lowering attack success rates across different threat models. However, the method also showed robustness when confronted with a poisoned model, fine-tuning it by re-training all layers and successfully unlearning the trigger.

2.4.6 Proposed Solution

Although the method of altered adversarial training (Geiping et al., 2021) was only tested against Hidden Backdoor triggers (Saha et al., 2019), 'unlearning' a trigger implemented using a Poison Backdoor (Gu et al., 2017) has proven effective when the trigger used for the adversarial training samples is the same or similar to the trigger used to implement the backdoor (Wang et al., 2019). Using a reverse engineered trigger (Figure 12) to fine tune a poisoned model, Wang et al. managed to achieve an attack success decrease from 99.9% to 0.57% (only 0.28% higher than when using the original trigger).

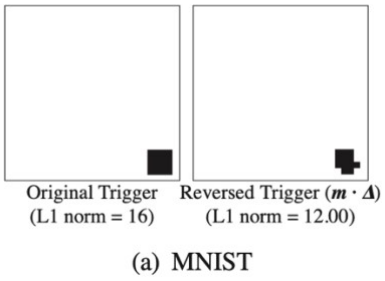


Figure 12: Original and reversed trigger (Wang et al., 2019)

Due to the clear success in the aforementioned studies, the solution proposed in this research will use the same fundamental idea of altered adversarial training, using clean-labelled images with a present trigger, but manually testing the effectiveness of slightly altered trigger patches on a Poison Backdoor trigger. By borrowing concepts from both adversarial training

against adversarial attacks and robustness training, using said concepts for a preventative defence, this research analyses how training alongside varying volumes of clean adversarial samples, as well as the variance in trigger formation, will impact a model's response to a known Poison Backdoor.

Differing from Geiping and his colleagues paper, this study will not go through the process of generating optimized adversarial attacks to use as a surrogate attacker. Instead, both the perceptible and non-perceptible trigger pattern is known to the defence. Therefore, adversarial samples specific to the backdoor can manually generated and used to explore any relationship between robustness and trigger similarity.

Theoretically, the aim is to desensitize the model to the present trigger, rendering the trigger an unreliable labelling tool for the model. The model should then disregard the backdoor trigger, instead of overfitting the undesirable behaviour to it, allowing it to effectively mitigate the backdoor.

3 Research Design

In developing the experimental process, multiple considerations were taken into account to ensure the experiment was robust, repeatable, and logically sound. In addition, it was important to be mindful of structuring the experiment's back-end so that it could be seamlessly incorporated with an HTML / CSS front-end to facilitate the experimental dashboard.

Collecting quantitative data rather than qualitative for this experiment was needed to objectively measure and analyse the impact of adversarial training using variables such as accuracy in the form of F1 scores as well as time and other data analytics tools that are expanded on in section 4.2. Whereas qualitive data is non-numeric, quantitative data enables precise quantification, statistical testing, and detailed analysis of different variables, ensuring that the findings are reliable, valid, and generalizable. This is crucial to a project wishing to assess the found results against the starting hypothesis- outlined in section 4.1.3.

3.1 Experiment purpose and aim

This research will integrate various clean adversarial samples into the original training set with the aim of mitigating the effect of a poison backdoor attack, implemented during the same training iteration. This experimental design requires the defender's knowledge of the trigger pattern used for the attack. The clean adversarial samples integrated in response to this attack will vary both on:

- Percentage of the training dataset they encompass
- Trigger perturbation present on the data

The purpose is to explore a potential 'clouding' effect of the backdoor attack using these clean adversarial samples, rendering the trigger an unreliable labelling tool for the model. When successful, the computational cost associated with the desensitization of the model to the backdoor trigger is then assessed.

This research will explore the possibility that multiple variations of the clean training sets' adversarial perturbation will succeed in clouding the trigger introduced by the backdoor attack (under the assumption that they are of similar nature, such as the same trigger pattern in a different location). Whereas is not directly relevant to real-world scenarios due to the white box nature, if successful, further research could be carried out investigating the level of similarity required between the backdoor trigger and the perturbations present in the adversarial training data. This could potentially reveal a set of 'generalisable' perturbations that would increase model robustness against a wider range of backdoors. Incorporating adversarial samples with said perturbations into training datasets would therefore be able to improve model robustness to backdoor through adversarial training, without any knowledge of the backdoor's existence or trigger.

3.2 Hypothesis:

a. Increasing the percentage of clean adversarial training data within the training set will improve the model's resilience against the backdoor attack when both the clean labelled adversarial training data and backdoor attack shares the same perturbation.

b. Varying perturbations on the adversarial training data will influence the effectiveness of this defence strategy, however it is expected that the clean labelled adversarial training data will positively impact the model's robustness to the backdoor attack assuming the perturbation on the adversarial training data and the poisoned data used for the backdoor implementation are of a similar nature.

c. Computational expense will increase in tandem with the increase in adversarial training samples added to the dataset

3.3 Ethical considerations

There were no ethical considerations relevant to this research as no surveys or other qualitative data was required, in addition all libraries and datasets used were publicly available. List of notable libraries and datasets below, full list in the [appendix 1.A][appendix2.A]:

DATASET	
MNIST dataset	Dataset of images used for training, testing and evaluating the network
Libraries	
Adversarial robustness toolkit	Used for implementing manipulations on the MNIST dataset images for implementing and training against network backdoors
torch	Used for neural network building and training
sklearn.metrics	Used to calculate f1 score of tested neural network
flask	Used for facilitating the website dashboard
matplotlib	Used for plotting of results graph

3.4 CNN Implementation

Figure 13 details the final CNN model used for this study, explained in section [2.2: Convolution Neural Network Structure]

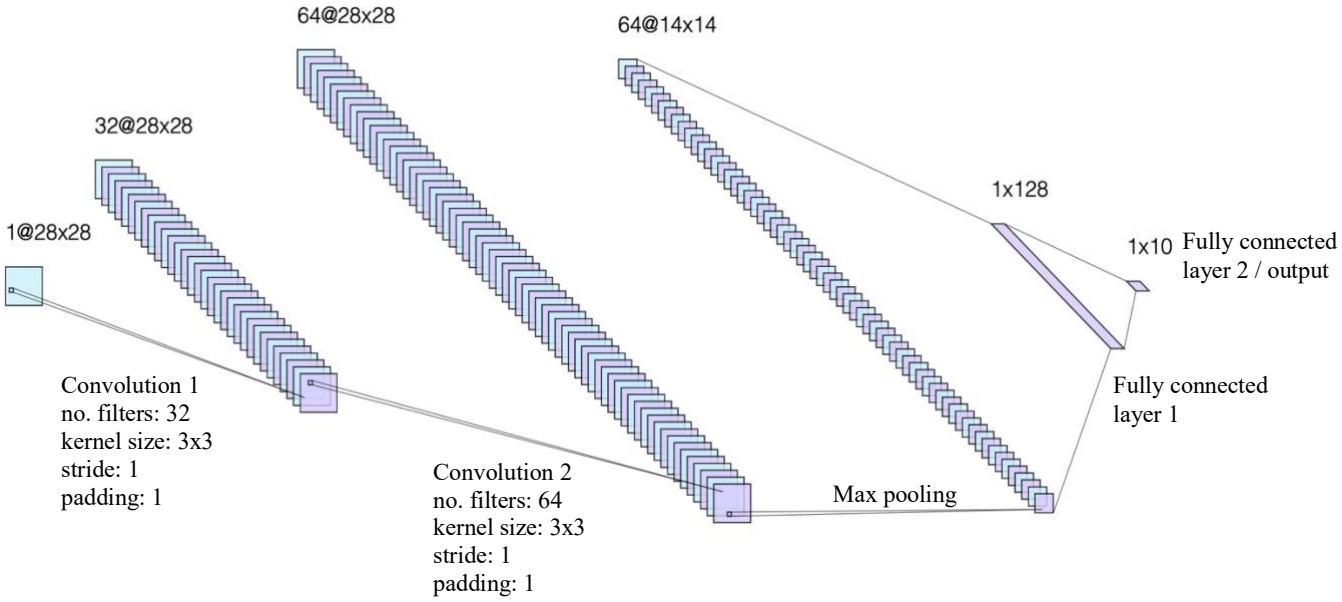


Figure 13: Diagrammatic representation of final CNN used in the experiment

When assessing the structure of the utilised CNN outlined above, it is important to note the randomisation of weights and mini batches within the model initialisation. As later explained in section [3.5.2.2: PyTorch], the library used for this project implementation was PyTorch. In Pytorch, when a layer is defined, the weights are initialised randomly by default, following the Kaiming (He) Initialisation method (He et al., 2015; Alahmari, S. et al., 2020). Both these factors are responsible for introducing variability in each instantiation of a model.

When Pytorch initializes the fully connected and convolutional layers, the randomised initial weights create different starting points for the training process on each run. This leads to different optimisation paths for each new model, introducing variability and resulting in what should be small variations in performance (Paszke et al., 2019).

The optimiser used is the Adam optimisation method, an advanced variant of Stochastic Gradient Descent (SGD). SGD and its variants work by randomly shuffling the

dataset and randomly selecting minibatches to compute the gradients for each epoch. This causes variations in the gradient updates, and therefore, in the training trajectory. The gradients for the randomly selected minibatches are also noisy, which helps the optimiser to escape small, suboptimal points (local minima) and continue searching for lower loss regions. Therefore, the optimiser can explore a broader range of the loss landscape, finding more robust, generalised solutions that are less likely to overfit the training data (Kingma & Ba, 2015). Due to the stochastic nature of minibatch selection and gradient updates, the optimisation process converges to different local minima and saddle points, resulting in different model performances across runs.

When discussing performance differences between instantiated models, it is crucial to remember said differences should be minimal. The variety of optimisation paths and training trajectories do not change model behaviour significantly for most instances, and the implemented model is well designed for its task. However, outlying models with significantly better or worse performance results are still possible. Therefore, the stochastic nature of both the initialization and training process is necessary to ensure generalised and robust results. In addition, as later expanded on in the Experimental Design, consistent results across multiple runs, of multiple models with varying shuffled training datasets indicate repeatable and reliable findings with solid foundation, rather than findings tailored to a single instance, which has the small yet possible potential to be irrelevant to most models created.

3.5 Experimental Setup and Implementation

This study employs multiple controlled experimental setups, wherein the independent variable is the proportion of adversarial training data (0%, 25%, 50%, 75%, 100%), and the dependent variable is the model's F1 score on both clean and backdoor-injected test data. This design is intended to isolate the impact of adversarial training data on model performance. The varying manipulations applied to the MNIST dataset for the defending and attacking datasets will be detailed in separate planned experiments, as defined and explained in Section 4.1.5.

3.5.1 Implementation

The experiment itself is run using the main run function present in the initial_network file [appendix 2.H]:

```
run(numberOfExperiments, adversarialPercentage, testName, trainName, experimentRepeats, fileName,
                                                    epochs=3)
```

Through this function, the entire experiment carried out on a singular chosen adversarial data percentage is executed, allowing the modifications of all defined parameters, explained as follows:

Parameter	**Purpose**
adversarialPercentage	Defines the percentage of the adversarial training dataset to be used for training model robustness during experiment run.
fileName	The name of the file saved to disk, where results from the experiment can be found
numberOfExperiments	The number of experiments run on the defined adversarial percentage. This differs from experimentRepeats as the MNIST dataset is reshuffled before all clean and adversarial datasets are cut from it. This repetition is necessary to ensure repeatability; allowing the identification and disregard of unexpected behaviours caused by uneven divisions of images in outlying subsets.
experimentRepeats	How many times a new trained model is created and assessed on the defined test dataset. This repetition is necessary to ensure repeatability; allowing any unexpected behaviours within outlying instantiated models discussed in section 3.4 to be recognised and ignored.

testName	The name of the perturbation that will be utilised for the backdoor implementation and final test subsets. The backdoor set will have the undesired 'poison' labels, implementing the backdoor. The final test set will be clean labelled for model performance assessment.
trainName	Name of the perturbation utilised for the adversarial training subset, said subset will be clean labelled and used for corrective training and enhancing model robustness against the implemented backdoor.
epochs=3	The number of training iterations the model goes through whenever trained.

As a standard, the values experimentRepeats, numberOfExperiments, and epochs were all given a value of 3. This is to ensure enough repetition to allow the experiment to be reliable and repeatable.

Within each numberOfExperiments iteration, first the initial dataset was gathered. Two copies of this dataset are created:

1. One copy of the entire MNIST dataset gets shuffled and split into the DataClass object initialDataset.

   This class has three core attributes; trainData, evalData and testData. The MNIST dataset is organised so that a random 60% of all the images fall within the trainData attribute and the remaining 40% are split evenly between the other two.
2. A second copy is split into the aforementioned percentages and returned as individual subsets (reasoning for this decision elaborated on in the reflection section) and individually passed to the generatePoisonAdversarialPy function (appendix [1.a]) for the manipulation defined by testName and trainName.

Name of Dataset		Percentage of total dataset	No. images in dataset	No. images used in dataset
Total dataset		100%	140 000	
Clean dataset		50%	70 000	
	Training data	30%	42 000	42 000
	Evaluation data	10%	14 000	14 000
	Testing data	10%	14 000	14 000
Adversarial dataset		50%	70 000	
	Training data	30%	42 000	$0 \leq x \leq 42\,000$
	Backdoor data	10%	14 000	14 000
	Testing data	10%	14 000	14 000

Figure 14: Explanation of full makeup of datasets used in the experiment

Figure 15 is a diagrammatic representation of the process of manipulating and dividing the MNIST dataset to run the experiment:

C21011405 – Evaluating Adapted Adversarial Training as a Preventative Defenc

e against Backdoor Attacks on CNNs

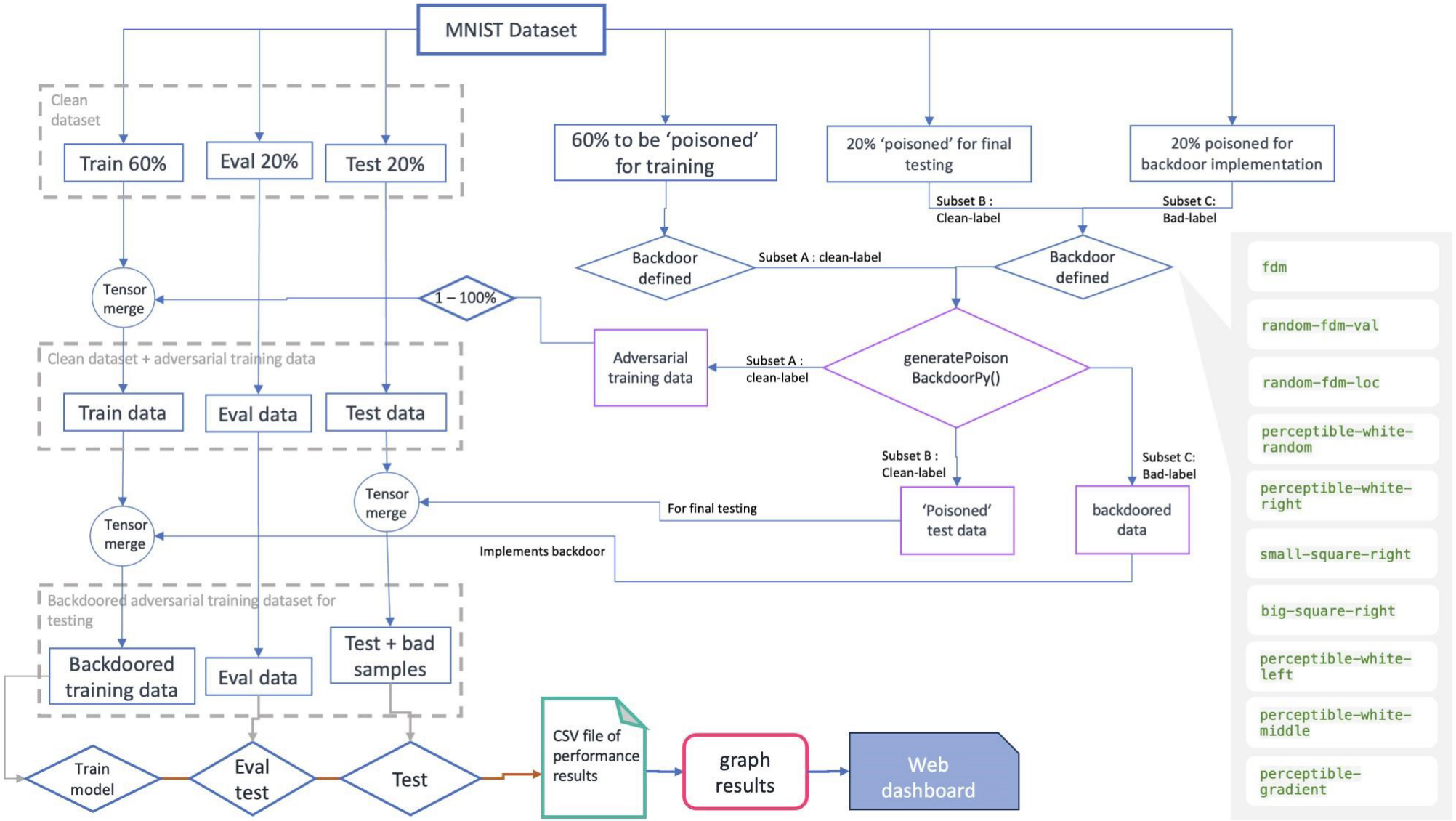


Figure 15: diagrammatic representation of the manipulations on the MNIST dataset required to acquire the datasets used in this experiment

3.5.2 Software

3.5.2.1 Adversarial Robustness Toolbox

To generate the necessary adversarial samples to generate the samples that will be used in both implementing the backdoor attack and for adversarial training, the adversarial robustness toolbox (ART) will be used. This decision is supported by the fact that ART boasts a dedicated and seamless integration with the chosen framework (PyTorch). This comprehensive support for PyTorch models, including a native PyTorch
classifier wrapper and backdoor attack implementation, makes it particularly well-suited for this project (IBM et al., 2024).

3.5.2.2 PyTorch

This decision to use PyTorch rather than TensorFlow in this project was made for several reasons. Pytorch is easy to use for Python programmers and has an intuitive Application Programming Interface (API) which makes it possible to write code and see the results straight away which simplifies debugging. It uses a dynamic computation graph, which allows for more flexibility and makes it easier to write and modify CNN models which is useful when experimenting with different tests. The strong community and extensive documentation available was also a prime consideration and the help available to troubleshoot ideas was invaluable during this project. In addition, PyTorch has good debugging capabilities because standard Python debugging tools like `pdb` can be used making it easier to identify and fix errors in the model or training code.
PyTorch was chosen over Tensor Flow because with TensorFlow debugging can be a lot more challenging. This is because Tensor Flow operates using a static computation graph so the entire model must be defined before running it and if problems occur it is not clear where they lie on the graph. This added complexity made TensorFlow less appealing for this project.

3.5.2.3 MNIST Dataset

The reason the MNIST dataset is being used for this research instead of a more complex one, is this project is specifically testing the model's ability to identify and handle poisoned samples, investigating the associated F1 score. The simplicity of the MNIST dataset, and therefore the expected high accuracy scores, allows for the isolation of solely on this test purpose. Otherwise, if lower scores were expected at the beginning of the training process, it's possible that a larger portion of potential improvements could be mistakenly attributed to identifying adversarial data than would be accurate.
In other words, a model trained on the MNIST dataset serves to eliminate potential confusion about whether improvements are due to the model's genuine learning due to the increased sample sizes or its ability to handle adversarial scenarios. This way, the impact of specifically adversarial data can be isolated and analysed.

3.5.3 Data Analysis tools

For the research carried out in this paper, quantitative data will be gathered as the study consists of iterations of a variable (amount of adversarial data) and measurable outcomes (model performance). These measurements are required to collect data on model performance metrics under different various conditions, varying the ratio of adversarial examples in the training set. Using a quantitative approach allows for the objective comparison of experimental findings, as well as replicability due to the well-defined metrics and experimental procedures. Specifically, the effects of incrementally increasing adversarial training data will be quantified in terms of:

3.5.3.1 Model performance

Model performance will be assessed in the form of F1 scores against a separate test dataset previously unseen by the model. F1 score analysis is particularly useful when accounting for both false negatives and false positives, as it balances the concepts of precision and recall, penalizing extreme values. Below are explanations of both concepts in regard to the MNIST dataset that will be used:

Precision: High precision indicates low false positive rate.
Precision refers to the ratio of true positives (correct predictions for a particular class) to the number of predicted positives for that class (the sum of true positives and false positives for that class), where class is defined any of the classes in range 0 - 9. To calculate the precision across all classes, each class's precision will be computed individually, before being averaged to provide an overall precision metric.

Recall: High recall indicates a low false negative rate.
Recall refers to the ratio of true positives (correct predictions for a particular class) to the total number of relevant class instances observed in the dataset. Again, to calculate recall across all classes, each class is computed individually and averaged to achieve an overall recall metric.

There are two average calculations available to use for these metrics within the F1 analysis: weighted and macro metrics. The macro metric doesn't account for class frequency in the dataset, calculated by summing the precision or recall of each class before dividing by the number of classes.
The weighted average, however, does account for class imbalance, accounting for the number of true instances of each class. This value then weights the precision or recall ($PorR$) of each class by its frequency in the dataset:

$$Weighted\ Average\ PorR = \frac{\sum^{N}_{i=1}(PorR_i \times Number\ of\ Class\ Instances)}{Total\ Number\ of\ Class\ Instances}$$

Weighted averages are particularly useful when class frequencies are uneven, as they prevent more populous classes from disproportionately influencing the metric. The MNIST dataset in its entirety is a balanced dataset. However, since the experiment consists of multiple instances of random shuffling and indexing across various stages, the weighted average is used for this research to account for variations within the relevant subsets. Despite the fact said variations are likely to be minor, ensuring any imbalance is addressed helps to guarantee an accurate reflection of the model's overall ability to correctly identify all classes relative to their true frequencies.

3.5.3.2 Computational expense

The computational expense of the adversarial training will be assessed in regard to the time cost, processing power required (CPU) as well as the amount of Random Access Memory (RAM) used. This variety of techniques is intended to provide a comprehensive view of the feasibility and real-world considerations necessary to make informed recommendations regarding robust CNN deployment.

Time Cost: Time efficiency is a crucial consideration in real-world applications, as the time taken for a model to be trained directly impacts the speed at which the model can be developed, improved, and deployed. Faster training times allows for a greater number of training iterations, resulting in more efficient refinement of models. This is a critical aspect of model practicality, not only in regards to the limited resource of time itself, but also in terms of financial and computational resources. Training deep learning models (especially CNNs) can be computationally and financially expensive (Schmidhuber, 2015; Goodfellow et al., 2016; Gu et al., 2017). Two attributes which can go hand in hand, especially if the organization utilises cloud computing environments, where resources are billed by usage (Gu et al., 2017). Managing these costs is an integral priority of any organisation, regardless of size.

It's important to assess and discuss this metric as the appropriate balance between accuracy and speed will vary based on the company's priorities. For example, when considering medical imaging, efficient deployment of improved models has been shown to have a significantly positive impact on patient outcomes (Litjens et al., 2017; Esteva et al., 2017). The critical nature of the field, however, calls for careful handling due to attempting to minimise trade-off in both directions.

The time cost we will be measuring in this experiment is solely the time of training iterations, rather than adversarial data generation and testing. This is because training iterations consume significantly more computational resources than the other phases and are the primary area where optimisation techniques can significantly reduce time and resource demands.

CPU & RAM usage: Like time costs, analysing the expected increase in processing power and RAM required for adversarial training is an important consideration when assessing the impact said training will have on a system. This is for multiple reasons:

again, similar to time costs, if the training loop requires excessive processing power or RAM usage, this can become economically taxing- especially when organisations manage their training in the cloud, as many do. In addition, high CPU/RAM usage often correlates with higher energy consumption (Gu et al., 2017). Assessing and reporting on these necessary burdens can therefore also help in estimating the energy footprint of training operations, crucial for managing environmental impact.

Measuring the CPU usage for this experiment will be done as follows, and can be viewed at [Appendix 2.D]:

1. Taking an initial percentage measurement of the CPUs currently in use at the start of each training epoch
2. Taking a final percentage measurement of the CPUs currently in use at the start of each training epoch
3. CPU usage per epoch then equals:

$$CPU\ usage\ per\ epoch = \frac{Final\ CPU\ \%\ usage - Initial\ CPU\ \%\ usage}{Number\ of\ cores\ on\ the\ experiment\ device}$$

   Dividing by the number of cores on the experiment device helps standardize the results, keeping the percentage value in the expected 0% - 100% range.
4. $CPU\ usage\ per\ epoch$ is then added to a list; which holds the averaged CPU usage of each training epoch of the training loop. At the end of the training function, the average of this list is calculated to obtain the average CPU use over all epochs, which will be returned to the $run()$ function as the CPU % usage for that experiment iteration.

As the experiment aims to isolate the CPU usage of the training process itself, the current CPU usage of the device must be accounted for before the training itself begins. This allows for the comparison of CPU usage before and after training, helping to minimise the effect other activity on the device. In addition to this, the experiments are

run overnight when there is limited device activity, further isolating the CPU usage of the experiment.

3.6 Experimental Walkthrough

To accurately and completely assess the effect of training a neural network on various sets of adversarial data, multiple test cases were run. Test cases were added as the need for further exploration became apparent, to fully understand and discuss the found results especially regarding the secondary hypothesis. Due to the nature of the multistage research, the most comprehensive method of communicating the experimental evolution and the proactive discussion of the found results is in a chronological manner. This section will detail the Stages of the experiment, highlighting the aims, relevant tests, results and conclusions of each Stage.

3.6.1 Test: Clean CNN model on MNIST dataset

Before starting the test cases for the experiment, it is crucial to check that the coded implementation of the CNN model (section [3.4: CNN Implementation]) [appendix 2.B] works as expected. Tests need to be run to ensure that the model can be created and trained on clean training data, then assessed on clean test data to achieve a high F1 score.

A lower F1 result could imply that:

1. There is a logical error in one of the: initialisation, train, test functions [Appendix 2.B , 2.G , 2.E] or in the data retrieval function [Appendix 1.B], so they are not working as expected. This would be the most likely option as without testing it is difficult to ensure all the functions are logically sound.
2. The MNIST dataset has been divided up so that the clean training data is insufficient in training the model for the clean test data, possibly due to unexpectedly weighted subsets. This possibility has been minimised as the MNIST dataset is shuffled during the `generateDataInitial()` function [Appendix 1.C], before being divided into train, test and

eval subsets. This ensures each run is trained and tested on a randomised subset of MNIST.

3. The structure of the CNN is not suitable for the task it is being provided with. This possibility has been minimised through the careful researching and planning shown in section [3.2: Hypothosis].

The clean model was evaluated as following:

```
model = MNISTCNN()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# evaluate clean model
cleanExperimentDataset = DataClass(initialDataset.evalData, initialDataset.testData, initialDataset.trainData, adversarialPercentage)
train_eval_loader = DataLoader(cleanExperimentDataset.trainData, batch_size=64, shuffle=True, collate_fn=custom_collate)
test_eval_loader = DataLoader(cleanExperimentDataset.evalData, batch_size=64, shuffle=True, collate_fn=custom_collate)
Ecpu, Eram, EtimeTaken = train(model, train_eval_loader, optimizer, criterion, epochs)
Ef1 = test(model, test_eval_loader)
clean_result = writeResult(testName, Ef1, adversarialPercentage, Ecpu, Eram, EtimeTaken, trainName)
writeCSV(clean_result, 'clean_test')
print('done clean test')
break
```

Figure 16: Code used to test clean model on clean data

Results

As shown, the clean model structure responds with an average F1 score of 0.986, this is high and proves that the clean model behaves as expected.

0.982993224
0.985865638
0.983709167
0.988002412
0.985975044
0.988004299
0.98814372
0.985158917
0.984860994
0.985342827
0.985361955

Figure 17: Clean model test results on clean test set

3.6.2 Poison Backdoor Implementation tests

In addition to testing the model, the backdoor manipulation must also be tested prior to beginning the defined experiment. This is crucial to isolate any unexpected behaviours in the test cases as relevant to the test case specifically.

1. Visually, as the backdoor trigger is perceptible, it should appear as expected
2. The implemented backdoor should have a significant negative impact on the models F1 score when presented with test samples containing the trigger

3.6.2.1 Visualisation test

As the function to apply the backdoor is looped, its safe to assume that the exact same perturbation is happening to all MNIST that travel through that loop. Therefore, visualising and checking a small sample of the images by eye is satisfactory.

The images were visualised using matplotlib.pyplot , utilizing the visualizeImage function [Appendix 2.F].

The manipulation present in this test was the perceptible-white-right function [Appendix 0], expected to place a 3x3 white square at the bottom right corner of the image.

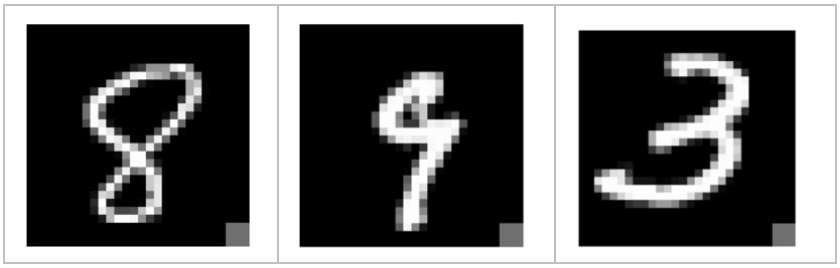


Figure 18: Visualisation of MNIST images manipulated using the perceptible-white-right attack

As shown, in Figure 18 the location and shape of the trigger is consistent with the expectations. However, the square's colouring is grey instead of white. This could be changed but is not an issue as the trigger pattern is consistent, and the colouring has made little difference to the perceptibility of the trigger. Therefore, the behaviour of the model should not be affected. It will be important in future stages to visualise a sample of the images manipulated by the used backdoor to ensure that the triggers are consistent with expectations.

3.6.2.2 Poisoned Model classification behaviour test

This test could be achieved utilising the `run()` function [Appendix 2.H] by setting the `adversarialPercentage` value to zero. As shown, the models' performance dropped by just under 0.5 when half the testing dataset contained the backdoor trigger (Figure 14). This confirms that the backdoor was effective on the poisoned testing datapoints, whilst still correctly classifying the remaining half of the clean samples.

0.496207580
0.494778152
0.495824513
0.49541902
0.496023041
0.494502183
0.494565299
0.493051667
0.49472828

Figure 19: Poison model test results on mixed test dataset

3.6.3 Stage one: Training with Trigger Pattern in Alternate Location to Poison Backdoor Trigger

In order to test both hypothesis [a.] and [b.], defined in section 3.2, the two types of backdoor manipulation used for the images were perceptible-white-right and perceptible-white-left [Appendix 0].

3.6.3.1 Adversarial perturbation Visualisation

Testing the visualisation of perceptible-white-left manipulation:

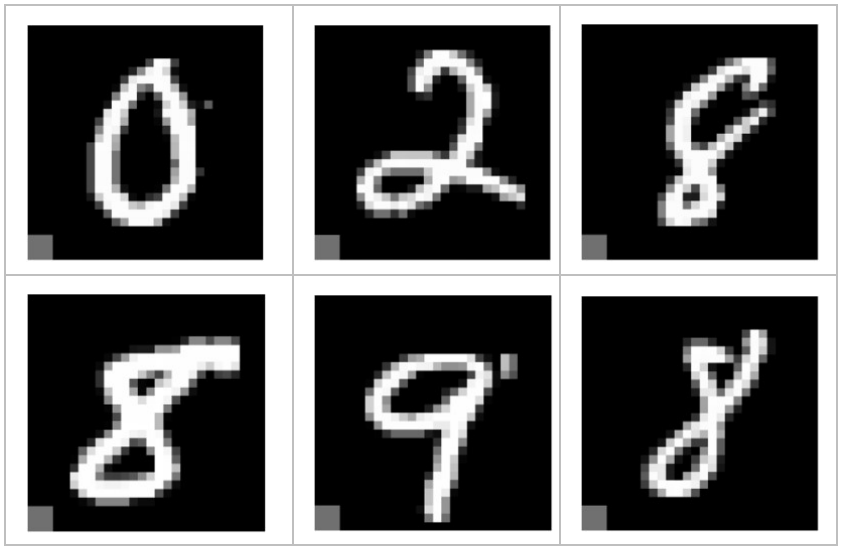


Figure 20: Confirms that the sole change to the trigger is its location

3.6.3.2 Test Cases

Test #	Training set	Backdoor type	Test Purpose
1	perceptiblewhite-right	perceptible-whiteright	Test hypothesis [a.] and if proven true, assess improvement curve of results in relation to data analytics methods defined in 3.5.3.2

2	perceptiblewhite-left	perceptible-whiteright	Test hypothesis [b.] using a similar but different manipulation technique, in this case, the perturbation technique is the same in all manners except location, as the adversarial training data has the pattern located on the left hand corner instead of the right. If proven true, assess improvement curve of results in relation to data analytics methods defined in 3.5.3.2

3.6.3.3 Results and discussion

TEST CASE ONE RESULTS:

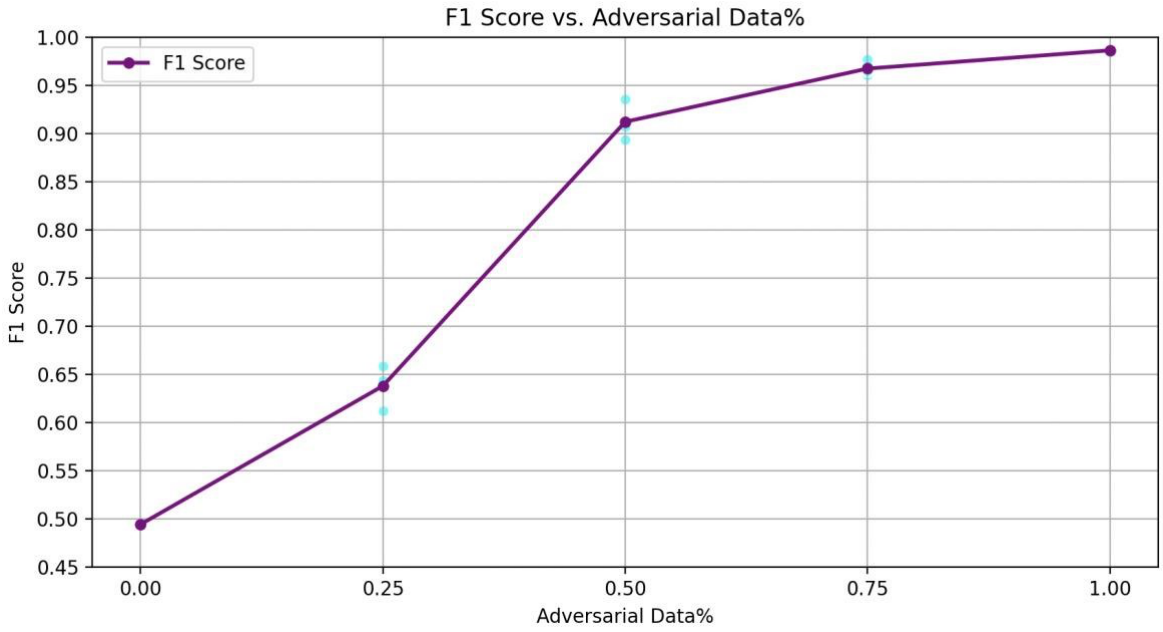


Figure 21: Test case 1, F1 score graphed

The drop in F1 score when the model has been poisoned with zero adversarial countermeasures shows that the backdoor attack is successful, creating a strong negative effect on the model's performance.

As shown in the results present in Figure 21, using clean versions the exact same trigger present for implementing the backdoor attack is successful at preventing against the attack

when the percentage of the adversarial data used is ~68%. Meaning that the clean-labelled adversarial training samples used account for 28,560 samples within of the training set, whereas the poisoned data used for implementing the backdoor account for 14,000 (42,000 of the samples are fully clean). Once the adversarial training data present in the training set reaches this number, the model achieves an F1 score of 95%, meaning the attack has been minimised by 93%; from producing a ~50% effect to one closer to 3.5%.

TEST CASE TWO RESULTS:

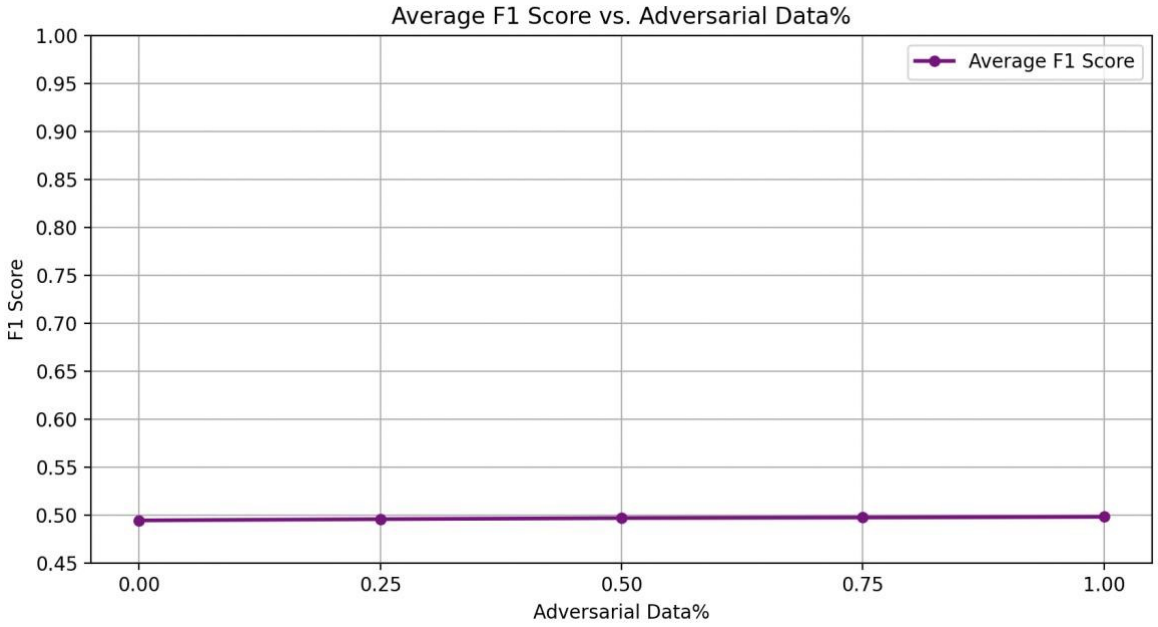


Figure 22: Test case 2, F1 score graphed

The results achieved by test case two were unsuccessful, training the model on the additional training dataset with the trigger at a different location appeared to have negligible effect on the success of the backdoor attack. Although a very slight increase in F1 score can be perceived, it is assumed this is due to the additional MNIST images being used for training- therefore correctly classifying a slightly larger amount of clean samples, rather than a response to the backdoor itself. This is likely as the model's accuracy when completely clean was not 100% (Figure 17). Regardless, the increase in F1 is negligible Figure 22, meaning that this could not be considered a notable decrease in attack efficiency in either scenario.

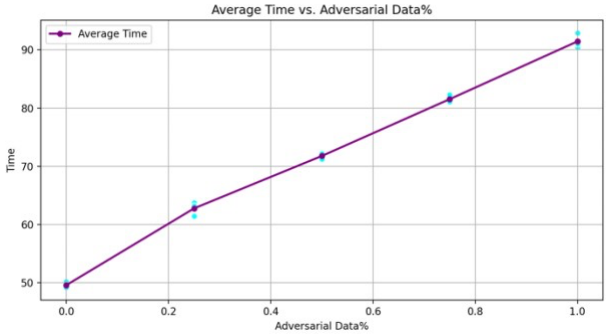
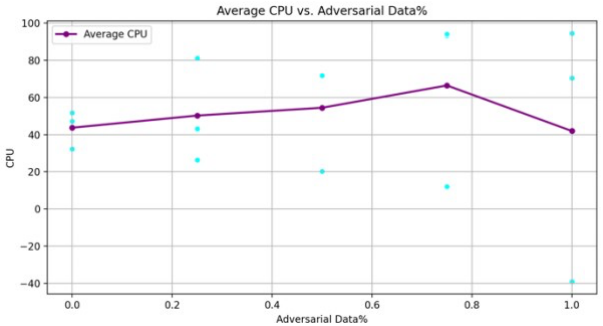
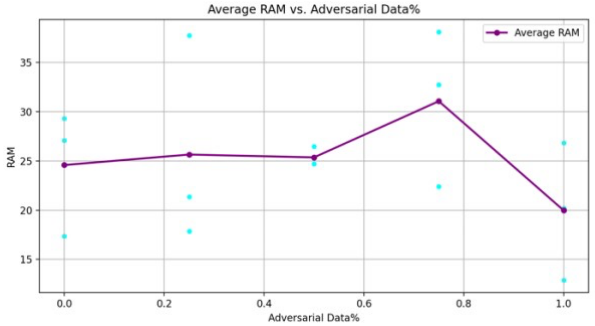
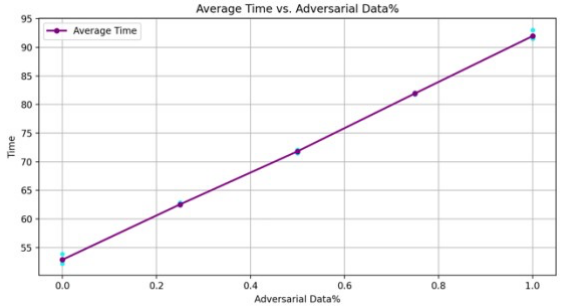
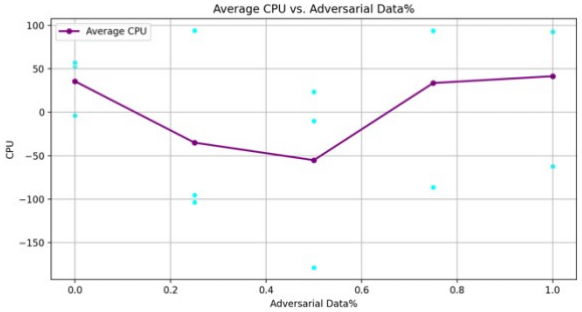
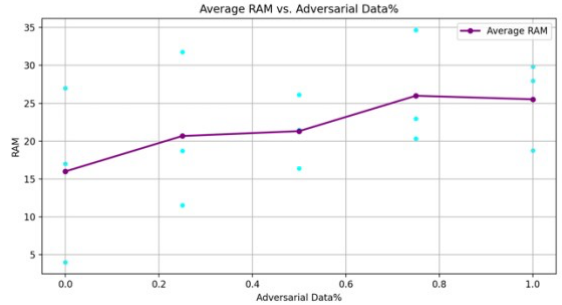
#1	 *Figure 23: Test case 1, average time taken per training iteration*	 *Figure 24: Test case 1, average CPU usage per training iteration*	 *Figure 25: Test case 1, average RAM usage per training iteration*
#2	 *Figure 26: Test case 2, average time taken per training iteration*	 *Figure 27: Test case 2, average CPU usage per training iteration*	 *Figure 28: Test case 2, average RAM usage per training iteration*

Figure 29: table of additional data analysis results for test case 1 and test case 2

DATA ANALYSIS DISCUSSION:

As shown in Figure 29, the CPU and RAM measurements collected in this experiment are inconclusive. The data shows some inconsistencies between test cases and a noticeable spread in data points. Inconsistencies between test cases are unexpected in this case as the measurements for computational expense are taken during the training iterations. Both triggers implemented in test case 1 and test case 2 have the same type of effect on the MNIST image, as they are identical perceptible perturbations in different locations (i.e. neither add additional noise). Additionally, there were a few instances where CPU usage readings were unexpectedly negative, suggesting areas that need refinement in data collection.

Currently, CPU and RAM usage are captured as snapshots at specific points in the process. However, both CPU and RAM usage is often volatile. Taking only two snapshots may not accurately reflect the average CPU usage during the entire epoch. A more accurate representation could be achieved by implementing continuous monitoring or calculating usage over time. This could help account for the natural volatility of these metrics, as reflected in the variations seen in the data.

The average time taken per training iteration, unsurprisingly has a linear relationship to the amount of data samples present in the training dataset. This could be important to note if a significant amount of adversarial samples would need to be incorporated in the dataset to achieve a robust model.

```
def train(…):
        cpu_use = []

        isolate process
        process = psutil.Process(os.getpid())

        num_cores = psutil.cpu_count()
        for epoch in range(num_epochs):
                initial_cpu = process.cpu_percent(interval=None)
                < training epoch >
```

Algorithm 1: Initial method of gathering CPU usage data

Due to the challenges in obtaining accurate CPU usage metrics in Stage One, the training function was modified in [section 3.6.6: Stage Four] to achieve more representative results [Appendix 2.G]. Initially, CPU usage data was gathered by calling

```
process.cpu_percent(interval=None)
```

which returns a CPU usage percentage since the last call. However, the first call in each epoch returned 0.0, as it captures usage from the start of the process. Subsequent calls provided CPU usage since the previous call, leading to calculations such as:

```
cpu_use.append((final_cpu - initial_cpu) / num_cores)
```

This approach occasionally resulted in negative values, which, while unexpected, can occur due to the volatility of CPU measurements. These fluctuations highlight the complexity of accurately capturing CPU usage in dynamic environments, suggesting the need for alternative methods or more stable measurement intervals. Although, due to time constraints this experiment was unable to be re-run, a more reliable measurement of CPU usage can be found in [section 3.6.6.1: Stage Four Data Analysis correction].

3.6.4 Stage two: Randomising Location of Trigger Pattern in Adversarial Training Data

The results from Stage One indicate that the model did not become desensitized to the presence of the trigger when the trigger was placed in a single, different location. Consequently, it can be inferred that, rather than learning that the presence of the 3x3 square was an unreliable labelling tool, the model overfitted to the 'fix' being the specific location of the trigger in the adversarial training set. This overfitting resulted in the left-hand square having no impact on the right-hand square, as the model perceived them as entirely distinct triggers due to their locational differences.

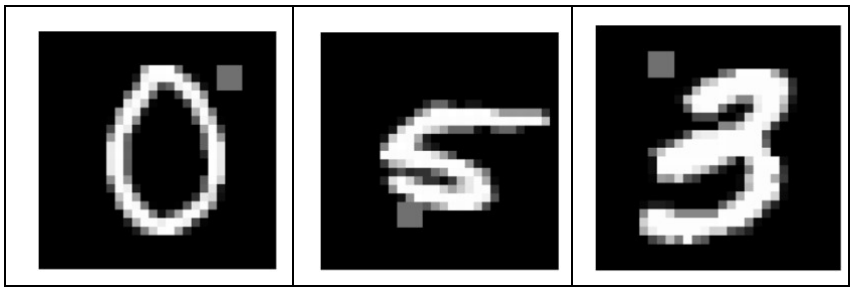
Therefore, in Stage Two an attempt is made to further 'cloud' the models understanding of the trigger via an adversarial training set that is manipulated with the same 3x3 square, however using randomised (x, y) values (function available in [Appendix 0]). This is a further attempting to encourage the model to view the square as a dynamic pattern, in order to prevent overfitting during both the adversarial training and the backdoor implementation.

Below details the testing methods used to ensure the integrity of the randomisation function, as well as the results and findings from this stage in the experiment.

3.6.4.1 Backdoor implementation testing and visualisation

After implementing the backdoor attack, the adversarial samples were visualised to ensure:

1. The randomisation function was working as expected
2. The trigger pattern was consistent with the trigger pattern present in section [3.6.2.1: Visulisation Test]



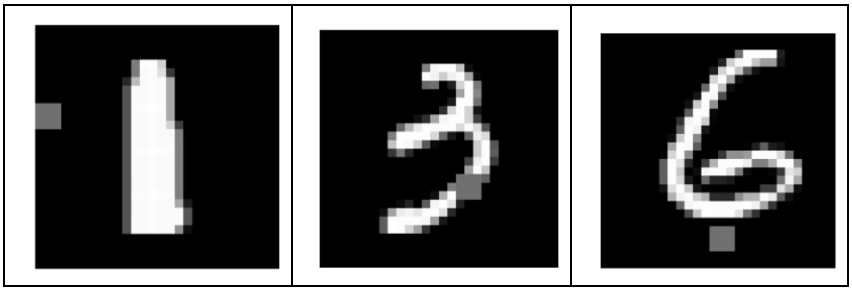


Figure 30: Visualisation of MNIST images manipulated using perceptible-white-random (Appendix[1.a])

To ensure that the model did not overfit to a specific location, it was crucial to analyse the heatmap of the square's location Figure 31, function available in [Appendix 1.E]. This analysis confirmed that the square was applied uniformly across the entire image, successfully achieving a randomised pattern.

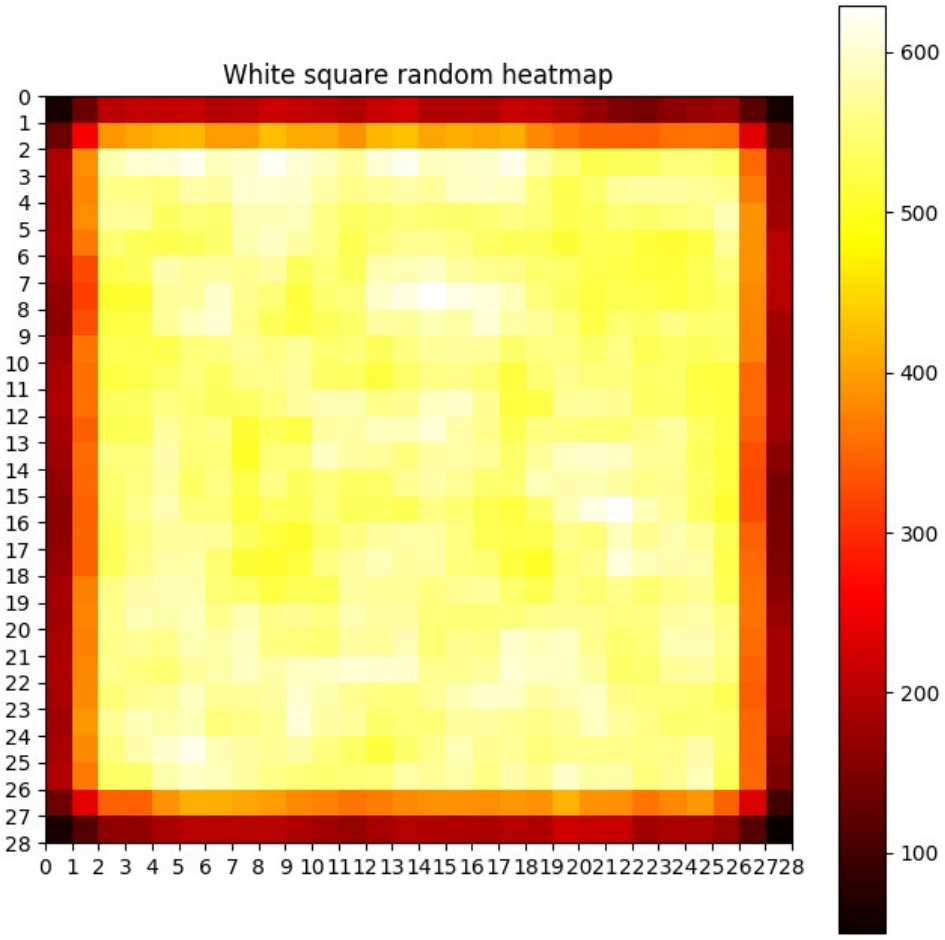


Figure 31: Heatmap of the randomised trigger square

The heatmap shown in Figure 31 clearly cools approaching the corners and the edges of the map. This is expected due to the concentration of available ‘overlap’ points these coordinates

have. However, it is worth noting that the backdoor manipulation being used currently puts the location of the trigger in the bottom right corner, not often visited by the randomised square.

For this reason, during this Stage, two test cases will be run. One using the original backdoor trigger, and another utilizing a backdoor trigger that is located more central to the image:

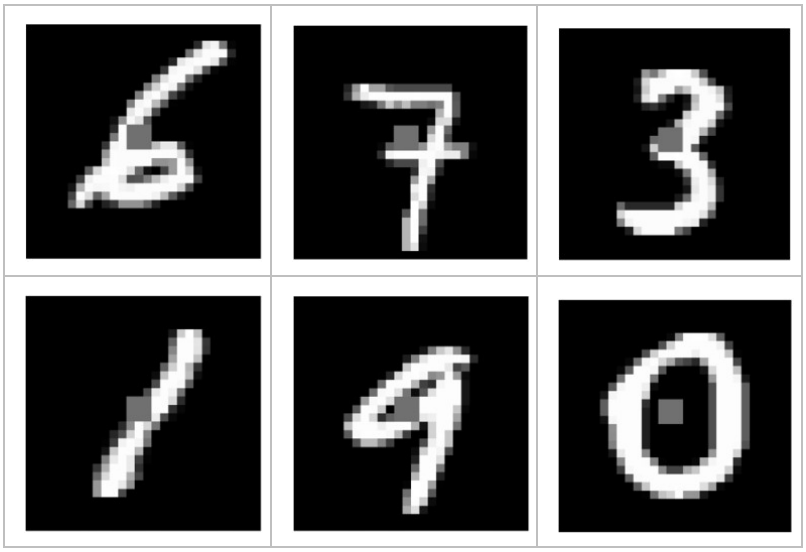


Figure 32: MNIST images manipulated by perceptible-white-middle (appendix [1.a])

Since the trigger is central to the image, it overlaps with the MNIST numbers present in the dataset. There is a possibility that this overlap will make it more difficult for the model to recognise the backdoor effectively, however this was proven to not be an issue through testing of the clean model on such examples.

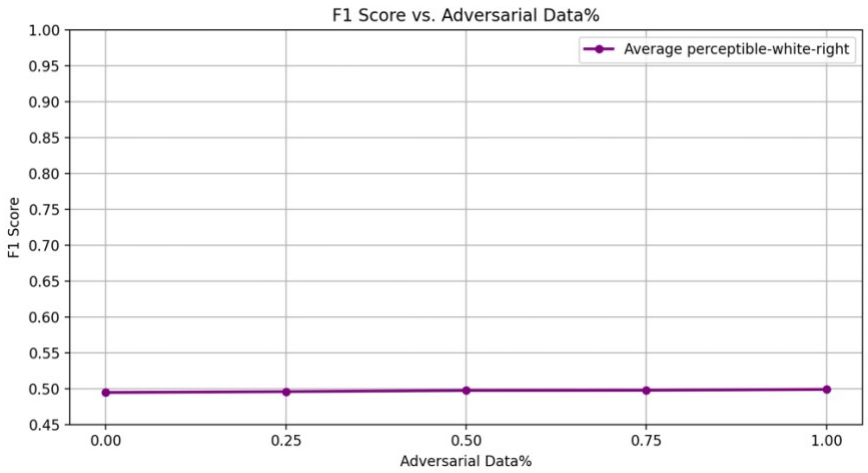
3.6.4.2 Test Cases

Test #	Training set	Backdoor type	Test Purpose

3	perceptiblewhite-random	perceptible-whiteright	Test hypothesis [b.] using similar but different manipulation technique, in that the location of the shared trigger will be randomised. The backdoor trigger will remain in the right hand corner.
4	perceptiblewhite-random	perceptible-whitemiddle	Same as test case 3, however the backdoor trigger pattern will be centred in the middle of the MNIST image. This avoids and/or highlights issues regarding a possible concentration issue where the corners of the heatmap were less visited by the randomised square in the training data.

3.6.4.3 Results and Discussion

Test case 3:



Test case 4:

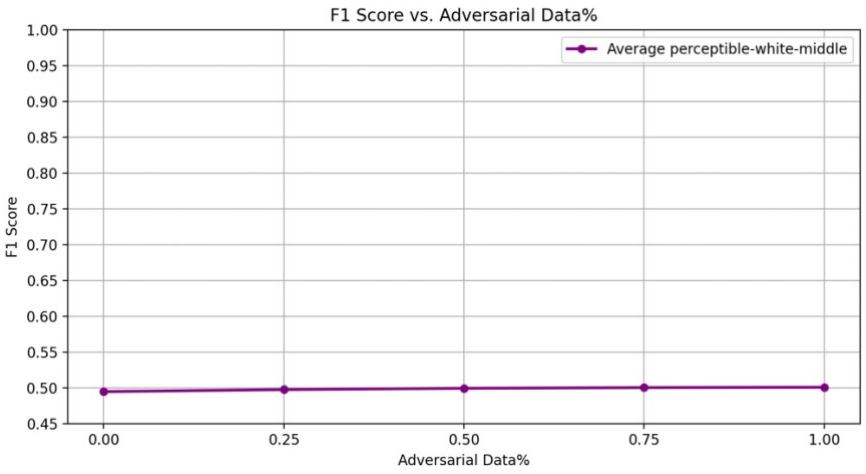


Figure 33: F1 Scores of test case 3 and 4 from stage two

The results (Figure 33) show that the model is immune from being clouded and desensitized from the trigger by varying the trigger's location randomly in the clean adversarial samples introduced during training time, even if the trigger used is identical to the trigger used at implementation. At this point, its concluded that the model cannot be trained to resist a perceptible Poison Backdoor attack (Gu et al., 2017) where the trigger shape is known but the location of said trigger is not by randomising the location of the trigger in clean adversarial training samples.

Data analysis will not be done when the adversarial training data was unsuccessful in mitigating the attack, as there is no retrievable point of diminishing returns without an increased model robustness.

3.6.5 Stage Three: Altering Shape of Trigger in One Location

As the locational variance of the trigger proved unable to prevent the effect of the backdoor attack, Stage Three will explore the formational difference of the trigger and if slight variations in the trigger's shape or colour are still able to positively influence the model at training time.

3.6.5.1 Implementation testing by visualisation

Before attempting the test cases, the following tests need to be carried out:

1. Each trigger manipulation should be visualised to ensure the expected trigger is being tested against the backdoor.

Test Case 5	Backdoor trigger	Test Case 6	Test Case 7

Figure 34: Backdoor trigger implemented by small-square-right

Figure 35: Backdoor trigger implemented by perceptible-white-right

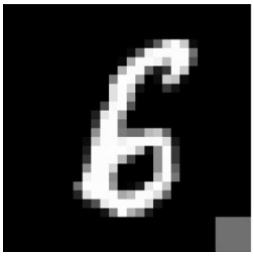


Figure 36: Backdoor trigger implemented by big-square-right

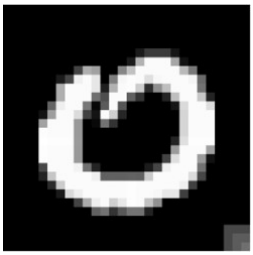


Figure 37: Backdoor trigger implemented by perceptible-gradient

Figure 38: Comparison of each backdoor used in Stage Three

Figure 38 clearly shows a comparison of the three adversarial triggers used in this Stage, as well as a side-by-side comparison with the poison backdoor trigger used to implement the attack. All are as expected.

3.6.5.2 Test Cases

Test #	Training set	Backdoor type	Test Purpose
5	small-square-right	perceptible-white-right	Trigger used for generating the clean adversarial training samples is 2x2 pixels instead of 3x3
6	big-square-right	perceptible-white-right	Trigger used for generating the clean adversarial training samples is 4x4 pixels instead of 3x3
7	perceptible-gradient	perceptible-white-right	Trigger used for generating adversarial samples is the same size as original trigger (3x3); however, the colouring of the trigger is a gradient instead of a solid colour.

3.6.5.3 Results and discussion

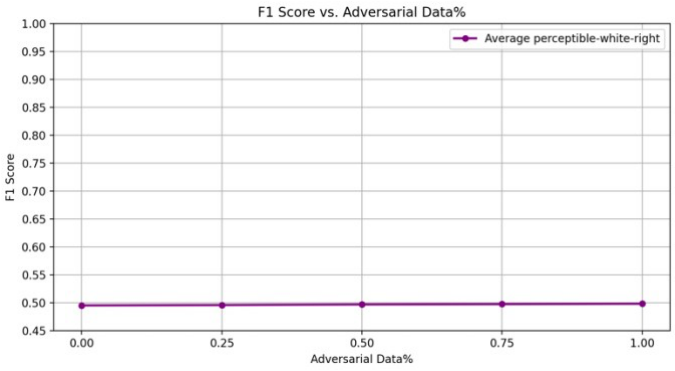


Figure 39: Test case 5, F1 Scores graphed

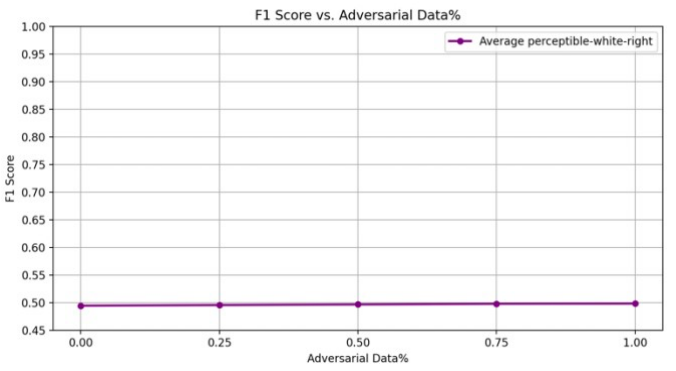


Figure 40: Test case 6, F1 Scores graphed

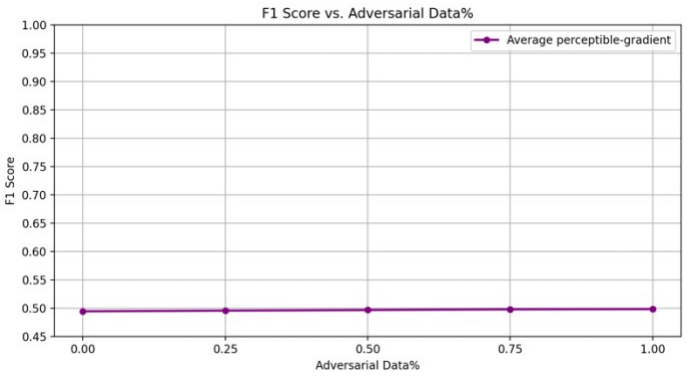


Figure 41: Test Case 7, F1 Scores graphed

As the results show, (Figure 39, 40, 41) even when the trigger used in the clean adversarial training samples encompassed the entirety of the trigger used for the backdoor (as was the case in test 6 (Figure 40), the training still had no effect on the model's ability to mitigate the backdoor. This is surprising since, as mentioned in section [2.4.6: Proposed solution], Wang et al. (2019) successfully utilized a reversed engineered trigger (that varied slightly from the original) to mitigate an attack in the post training stage of a model poisoned using a perceptible Poison Backdoor attack. In addition, Poison Immunity (Geiping et al., 2021) was most effective against Hidden Backdoor Attacks when utilized at this training stage, rather than during the post training stage. Learning from both these studies appears to suggest that a slight deviation in the trigger, used for clean adversarial samples in the training stage could

strongly encourage the model to ignore the trigger used in the backdoor. This, however, is inconsistent with the acquired results, suggesting either:

1. Method used for acquiring the reverse-engineered trigger (Wang et al., 2019) creates a trigger the model sees as more similar to the ones tested in this study, resulting in one which can be used to effectively mitigate the backdoor better than the manual recreations presented here. More on this point in the final discussion.

2. The Poison Backdoor Attack used in this research against this model is too strong, and incompatible with this type of robustness training.

3.6.6 Stage Four: Testing Solution on Imperceptible Poison Backdoor

Due to the unsuccessful nature of the results in previous stages, the question explored in Stage Four is whether the perceptible backdoor attack is too effective on a model specifically crafted to be highly sensitive to high-level features. As shown in section [2.2: Convolutional Neural Networks] the model used for this research has been specifically designed to be effective on the MNIST dataset. Because of this, during max-pooling the salient features focused on by the model are features such as edges, strokes, and the general shapes as these are the key features that allow the model to differentiate the digits. In the previous stages of this experiment, both the backdoor implemented, and any triggers used in the clean adversarial training data were perceptible squares, in other words, high-level features. From the previous results it can be gathered that no alterations to the trigger implemented for the backdoor would succeed in mitigating (or 'clouding' the attack). This suggests that the model treated every trigger presented to it as completely distinct features, possibly due to its strong ability to recognise high-level features.

In this Stage Four, three new test cases are proposed that implement an imperceptible Poison Backdoor (Gu et al., 2017), created using a subtle frequency domain manipulation (FDM), in place of the perceptible 3x3 square. The frequency domain backdoor manipulation has been proven as an effective method to fool both human and DNNs during training process (Liu et al., 2023) and was achieved using NumPy's discreet Fourier Transform function, which uses

Fast Fourier Transform (FFT). To implement the backdoor fdm [appendix 1.D], the 2D transform is computed over the entire image, converting it to the frequency domain before adding a slight modification $\alpha = 0.5$ at the fixed frequency coordinate (10,10). The sample altered is then converted back to the spatial domain, using the inverse FFT before being clipped to ensure all pixel values remain in the valid greyscale range. The resulting image is one with an imperceptible backdoor, one that exactly mimics the effect of the previously used backdoors in terms of its initial effect on the model, causing it to achieve the same F1 score as its perceptible counterpart.

To mitigate the imperceptible backdoor, two randomisation functions [appendix 1.D] will be attempted:

1. random-fdm-val : randomises the slight modification α to be used at the fixed frequency coordinate between $-0.5 < \alpha < 1.5$
2. random-fdm-loc : randomises the fixed frequency coordinate (x, y) manipulated by $\alpha = 0.5$ where $0 \leq x \leq 28$ and $0 \leq y \leq 28$

3.6.6.1 Data Analysis Correction

The initial approach to measuring CPU usage shown in [section 3.6.3.3: Section 1 Results] faced challenges due to the volatility and unpredictability of CPU usage data when using `process.cpu_percent(interval=None)`. This method returned `0.0` on the first call and measured CPU usage since the last call on subsequent invocations, leading to inconsistencies and occasional negative values in the recorded data.
To address these issues and achieve more accurate and representative CPU usage measurements, the `train` function was revised as follows:

```
def train(…):
        cpu_use = []

        isolate process
        process = psutil.Process(os.getpid())

        num_cores = psutil.cpu_count()
        for epoch in range(num_epochs):
                < training epoch >          final_cpu =
        process.cpu_percent(interval=0.1)
        cpu_use.append(final_cpu / num_cores)
        avg_cpu_increase = statistics.mean(cpu_use)  return
        avg_cpu_increase
```

3.6.6.2 Implementation testing and visualisation

Before attempting the test cases, it's crucial that each aspect of this new backdoor is set to work as expected.

1. The backdoor must successfully poison the model, producing a low F1 score when tested with no adversarial training
2. The randomisation of both α and (x, y) must be confirmed as an even distribution within the given range
3. Both the backdoor and the two adversarial trigger functions must produce a backdoor imperceptible to the human eye

3.6.6.2.1 Backdoor Effectiveness Test

To measure the effectiveness of the backdoor, the poisoned model is tested against the test dataset (including clean labelled backdoored samples). As shown in Figure 42, the backdoored model structure responds with an average F1 score of 0.495. This is significantly lower than the clean model's F1score and proves that the backdoored model behaves as expected, thereby confirming the effectiveness of the backdoor.

0.495710508
0.494264927
0.495523975
0.494281120
0.495327269
0.495740396
0.493296672
0.496015666
0.495659323
0.496197726
0.49577439

Figure 42: F1 Scores of CNN poisoned with the imperceptible FDM backdoor

3.6.6.2.2 Randomisation Tests

random-fdm-loc:

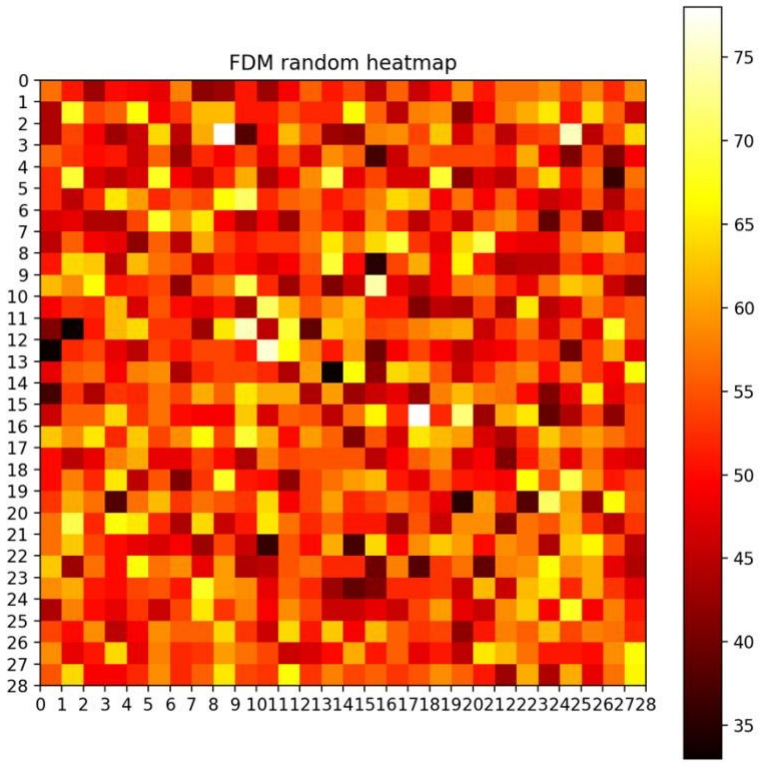


Figure 43: Heatmap of the randomised (x, y) values for the +0.5 FDM

Similar to testing done in stage two, Figure 43 represents a heatmap illustrating the frequency distribution of the coordinate pairs (x, y) selected by the random-fdm-loc function for FDM for the clean adversarial samples. The heatmap reveals that the manipulations are applied with a largely uniform distribution across the entire image, indicating that the randomisation does not exhibit any significant bias, as desired.

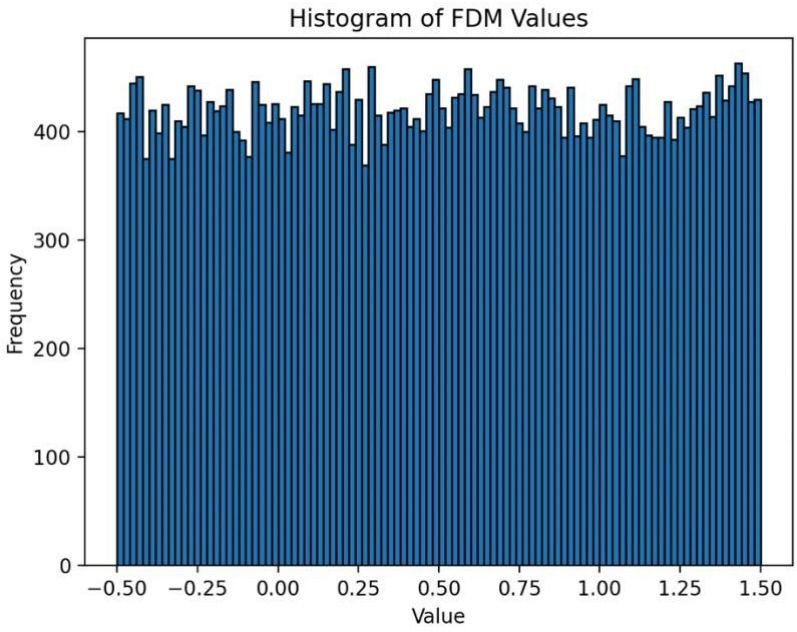


Figure 44: Histogram of the randomised a values

random-fdm-val :

To test accurate randomisation of α within the random-fdm-val function, a histogram was used.

This histography Figure 44 clearly shows that the randomisations within the function are mostly uniformly distributed, with slight and expected variation.

3.6.6.2.3 Visualisation Tests

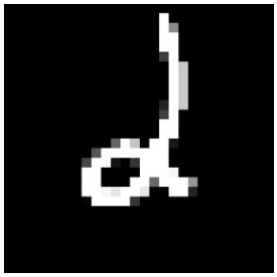
fdm	random-fdm-val	random-fdm-loc
		

Figure 45: Table visualising the different FDMs used in this stage

As shown in Figure 45, all FDMs used both for the backdoor attack and adversarial training manipulations are imperceptible to the human eye.

3.6.6.3 Test Cases

All test case implementations available in [appendix 0 , 1.D]

Test #	Training set	Backdoor type	Test Purpose
8	fdm	fdm	Testing hypothesis [a.] with an imperceptible backdoor, FDM = α and $(x,y) = (10,10)$
9	random-fdm-val	fdm	Testing hypothesis [b.] , trigger used for generating the clean adversarial training samples has varying α values
10	random-fdm-loc	fdm	Testing hypothesis [b.] , trigger used for generating the clean adversarial training samples has varying (x, y) values

3.6.6.4 Results

Shown below in [figure 46 47 48], all three test cases were successful in mitigating the backdoor with the same effectiveness.

Results will be discussed in [section 3.7: Final Results and Discussion].

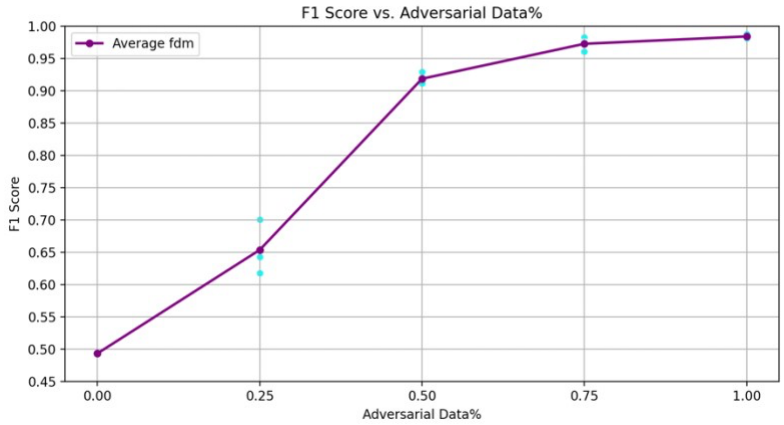


Figure 46: Test case 8, F1 Scores graphed

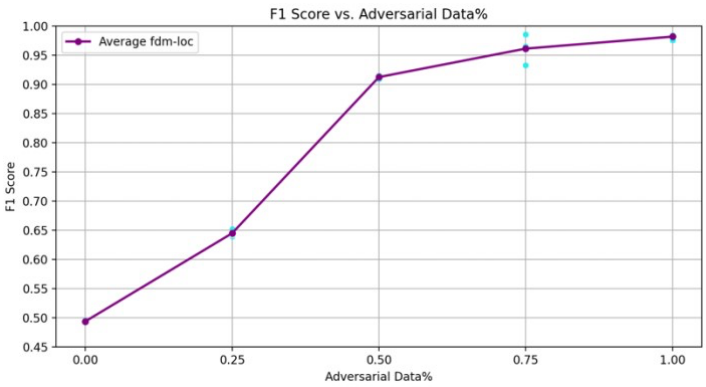


Figure 47: Test case 10, F1 scores graphed

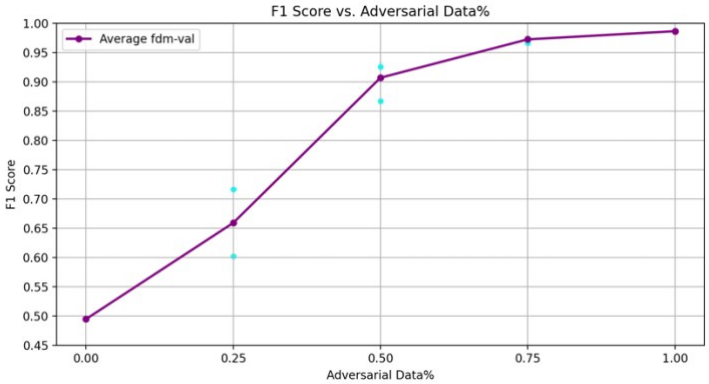
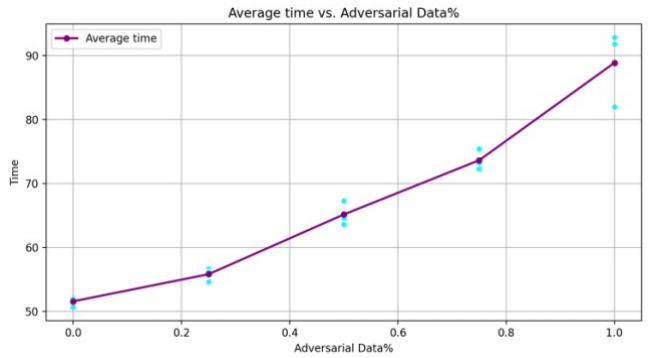
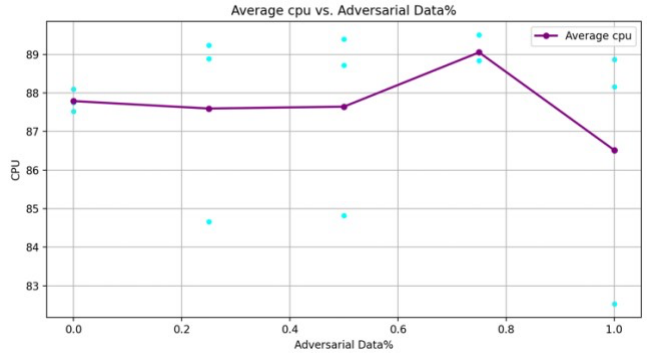
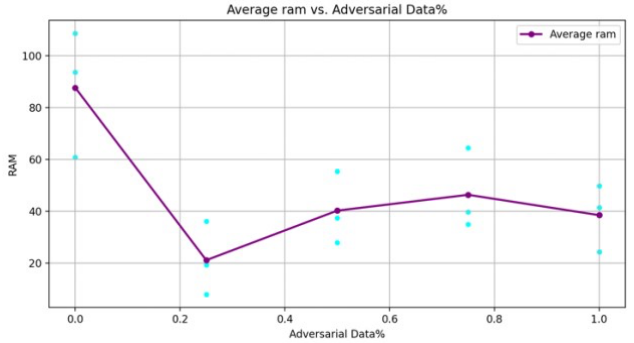
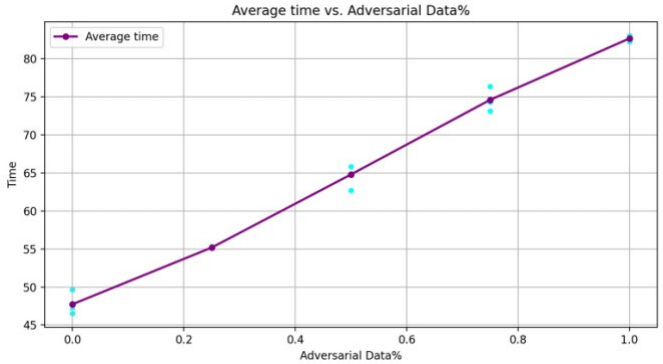
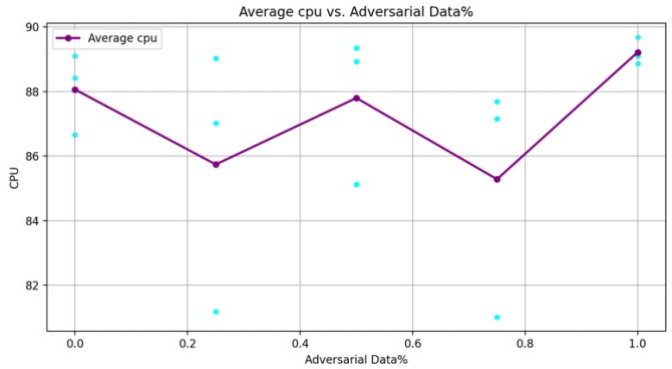
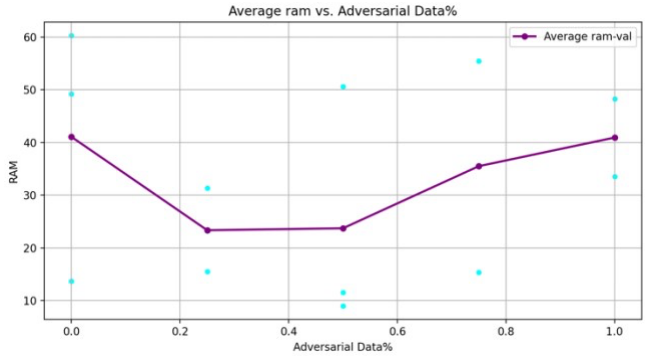


Figure 48: Test case 9, F1 Scores graphed

Test #	TIME	CPU	RAM
8			
9			

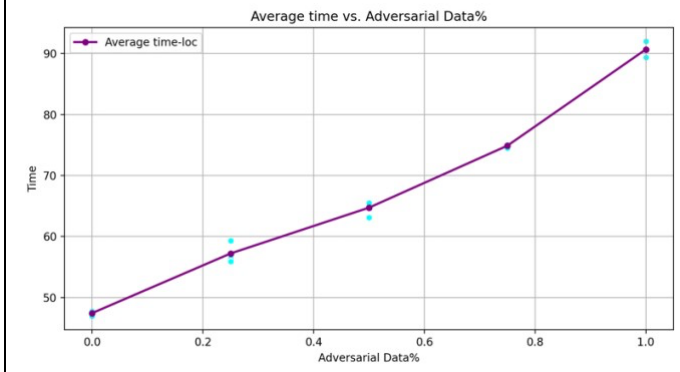
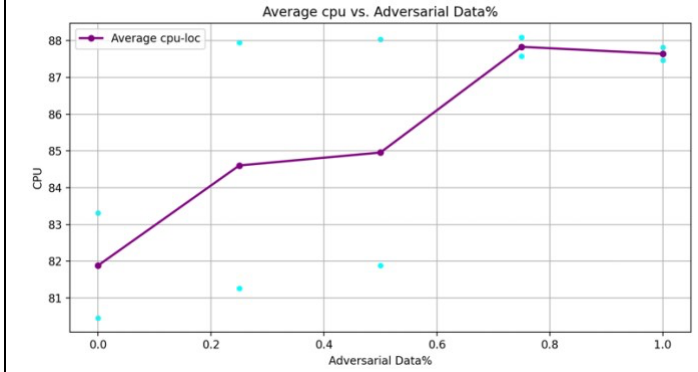
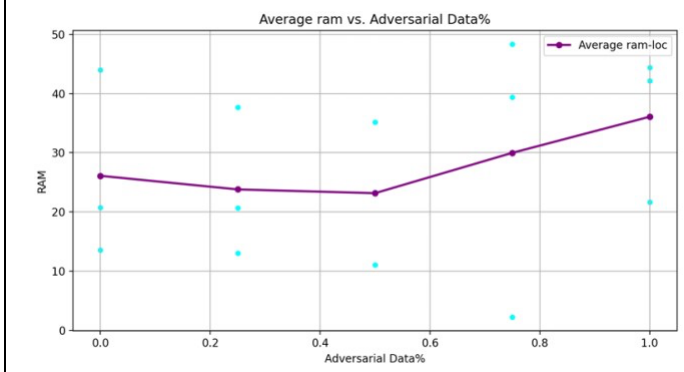
10			

Figure 49: Table detailing the Data Analysis results of Stage Four

3.6.7 Final results and Discussion

As shown in Stage Four, when the clean adversarial training data used triggers generated with random variations in the FDM parameters (α or (x,y)), it was successful in mitigating the fixed FDM attack. The success of all test cases (8, 9 and 10) to significantly improve the F1 score also shows no (or negligible) drop in the defences' effectiveness despite the adversarial training trigger's variance in value or location in test cases 9 and 10 respectively. This demonstrates that the model was successfully desensitized to the imperceptible trigger, which was subtle and reliant on low-level features.

The data analysis results are consistent with Stage One, in that CPU and RAM usage still prove inconclusive results in terms of trends- however the CPU values are no longer negative. Time is as expected.

4 Discussion

4.1 Discussion of Results

This research demonstrates that when the clean adversarial training data used triggers generated with random variations in the FDM parameters (α or (x, y)) in Stage 4, it was successful in mitigating the fixed FDM attack.There is scope for further work to be done, because, despite several attempts, these results could not be achieved when the backdoor trigger used to poison the model was perceptible and reliant on high level features. These findings (from Stages One, Two and Three) prompted a theory that the model was too adept at recognizing high-level features to generalize the varying clean triggers presented to it during training. The new results gathered in Stage Four support this hypothesis, suggesting that the model's sensitivity to high-level features like the shape and colour of the trigger was too strong to be mitigated by slight variations in those features during adversarial training. This was because the model viewed even slight changes in the perceptible trigger patterns as distinct patters. Low level features, however, which the model had not been built for or trained on, were less easily differentiated by the model. The lack of distinction between the triggers introduced within the clean adversarial training data and the poison backdoor data

meant the model was able to generalize the ‘fix’ trigger, using it to ‘patch’ and unlearn the poison one.

Following these results, further investigation into the Neural Cleanse experiment (Wang et al., 2019) was deemed necessary, as it appeared to achieve outcomes via variations of perceptible trigger unlearning that were not observed in this study; specifically, how their non-original trigger was able to mitigate the backdoor.

From deeper research, it was gathered that in order to reverse engineer the trigger used: for a given label, the algorithm treats it as potential target label of a targeted backdoor attack. The optimization scheme then finds the minimal trigger required to classify all samples from other labels into this target label. In the vision domain, the trigger found defines the smallest collection of pixels and their associated colour intensities to cause misclassification. The technique is then to do this for all triggers, before running an outlier detection algorithm to detect if any trigger candidate is significantly smaller than other candidates. A significant outlier would effectively represents a trigger.

Being reverse engineered, the trigger had been actively recognized as a key labelling feature before the attempted unlearning, instead of trying to manually engineer one to be recognized by the model as similar to the original. Despite the manual triggers being similar in shape, size, and color, the process of engineering them involved a significant, although initially underestimated, degree of uncertainty, unlike the systematic reverse engineering approach used in Neural Cleanse. This highlights the challenge of anticipating the model's learned representation of the trigger, which may diverge from the original design in unexpected ways. Wang et al. (2020) highlights that: ‘when the model is initially training to recognize the trigger, it may not learn the exact shape and colour of the trigger. This means the most “effective” way to trigger the backdoor is not the original injected trigger, but a slightly different form’. On one hand, this suggests that small variations from the original poison trigger would be accepted as a mitigating generalization for the attack; however, on the other hand, it could also imply that the manual alterations made to the attack trigger involved a larger magnitude of change than initially anticipated.

This brings the discussion to a new possibility regarding the perturbation budget allowance from the attack trigger. A perturbation budget defines the allowable changes to an input before it is no longer recognized as the same class, usually in the case of classified images, however, also applicable to backdoor triggers. In the case of perceptible triggers, the perturbation budget may have been exceeded, leading the model to treat even slight variations as entirely new patterns, rather than as minor deviations from the original trigger. Changes in Frequency Domain Manipulations have a much smaller perturbation change, possibly creating a situation that wasn't to do with whether or not the trigger was perceptible or imperceptible, but in the perturbation budget from the original trigger.

4.2 Reflection on process and Alterations from Initial Plan

When I wrote my initial plan back in January, I had a lot of interest and very little knowledge on the subject area I was just beginning to learn about. I'm glad to say that I feel like I've learnt a lot through my in depth exploration of this field, even though it's clear I've hardly scratched the surface. As a result of this however, my project aims have shifted somewhat significantly from what was outlined in the initial plan. The aims of my initial plan focused heavily on the point of diminishing returns, as I planned to increase the amount of various types of adversarial samples in the dataset well beyond 100%, observing the inevitable joint increase in the percentage of adversarial samples and F1 score until it hit a plateau, or the computational costs became too expensive to justify the means.

This did not go to plan, instead, the perceptible backdoor I had implemented was stubborn, refusing both the perceptile-white-left backdoor and most surprisingly, the perceptible-white-random backdoor. From there, I had two options, either I continue with the sole dataset that had given the 'expected' results (when both the attacking and defensive triggers are identical), and focus on analyzing the trends in computational expense as a dataset grows larger. Or, I focus on the adversarial training portion of the research and attempt to actually mitigate an implemented backdoor. So, I revisted my research, eventually discovering both the the Neural Cleanse paper (Wang et al., 2020) and Geiping et als' (2021) 'What doesn't kill you makes you robuste(er)'. I read of visually similar but not identical triggers being utilized for

unlearning a trigger implemented with the Poison Backdoor attack, and Geiping and his colleauges' success in using adapting adversarial training for the Hidden Backdoor attack. So I set up Stage Three.

After the results from stage three came back inconclusive, I decided to turn my attention onto the model itself. I knew I had thought into ensuring that it was a good fit for the MNIST dataset, which I realised (with the nature of the trigger and the MNIST images itself) could be working against me. Therefore, I created an imperceptible backdoor and attempted this instead. Thankfully it worked, However there's much I would change if I was to go back and do this project from the start.

The challenges I faced in this project were substantial, but managable, in both my coding implementations and (as discussed) the theoretical side of understanding the inner workings of a CNN and reshaping my assumptions in the face of new results.

One of my the struggles I overcame within the coding was based in my previous understanding of the various variable types I was using when manipulating subsets of the MNIST database.

For example, when I planned my experiment I originally created the DataClass with the intention that the it would be used to handle all dataset related manipulations and organisation. Although it does handle most of it in the final versions, the initial idea was that in the following line:

```
initialDataset, adTrain_set, adTest_set, backdoor_set = DataClass.generateDataInitial()
```

would have been:

```
initialDataset, adversarialDataset = DataClass.generateDataInitial()
```

The thinking was that the different attributes of adversarialDataset (testData, evalData – used as the backdoor data , trainData) could then called on and manipulated according to the defined test and train manipulations passed at run time. Unfortunately, I underestimated the difficultly of manipulating images of class attributes of type Subset. Although the attributes of type subset could be passed to the dataloaders without issue, the direct manipulation of images necessary for the backdoor manipulation required multiple stages of variable type modification that became too complex to justify. The solution was to pass the subsets directly to the main code, and manipulate them as separate entities.

Despite the struggles, I'm proud of the knowledge I've gained about Convolutional Neural Networks (CNNs) and the complex world of adversarial attacks and defenses. When I first began this research, I had only a basic understanding of CNNs, but through this journey, I've developed a deep appreciation for the intricate mechanisms that make these models both powerful and vulnerable. Exploring adversarial attacks has shown me how even the most robust systems can be exploited with carefully crafted inputs, while learning about various defense strategies has equipped me with the tools to counteract these threats. This experience has not only expanded my technical skills but also deepened my understanding of the ongoing challenges in securing AI systems. The opportunity to engage with such cutting-edge concepts and apply them to real-world problems has been incredibly fulfilling, and I'm proud of the progress I've made in mastering these complex topics.

5 Conclusion

The study was conducted in several stages, each designed to explore the effectiveness of adversarial training in mitigating backdoor attacks. Initially, perceptible triggers were used to assess how variations in location, size, or appearance during training affected the model's vulnerability to these attacks. Despite different manipulations, the backdoor attack persisted indicating an overfitting to specific trigger features. So, in subsequent stages, the research developed and the focus shifted to exploring imperceptible backdoors created using frequency domain manipulation (FDM).

The efficacy of both perceptible and imperceptible Poison Backdoor attacks and defence mechanisms on a CNN trained on the MNIST dataset, was explored. The focus was originally on perceptible triggers, however grew to include imperceptible triggers as the scope of the research widened due to unexpected results. The key objective was to determine whether variations in trigger location, size, or appearance used during adversarial training could successfully mitigate backdoor attacks, which were intentionally introduced to the model during training. The results across the different stages of the study indicate that while the perceptible defensive triggers consistently failed to mitigate the perceptible backdoor, the frequency domain manipulation (FDM) backdoor—an imperceptible attack—was successfully mitigated using randomised triggers in the frequency domain.

6 References

Abadi, M., Chu, A., Goodfellow, I., McMahan, H.B., Mironov, I., Talwar, K. and Zhang, L. (2016) Deep Learning with Differential Privacy, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* pp. 308-318. doi:10.1145/2976749.2978318.

Ahmed, I., Jeon, G., and Piccialli, F. (2022) From Artificial Intelligence to Explainable Artificial Intelligence in Industry 4.0: A Survey on What, How, and Where, *IEEE Transactions on Industrial Informatics,* 18(8), pp. 5031-5042. doi:10.1109/TII.2022.3146552.

Alahmari, S.S., Goldgof, D.B., Mouton, P.R. and Hall, L.O. (2020) Challenges for the Repeatability of Deep Learning Models, *IEEE Access,* 8, pp. 211860-211868. doi:10.1109/ACCESS.2020.3039833

Albawi, S., Mohammed, T.A. and Al-Zawi, S. (2017) Understanding of a convolutional neural network. *2017 International Conference on Engineering and Technology (ICET)*, Antalya, Turkey, pp. 1-6. doi:10.1109/ICEngTechnol.2017.8308186.

Bezirganyan, G.V. and Sergoyan, H.T. (2022) A Brief Comparison Between White Box, Targeted Adversarial Attacks in Deep Neural Networks, *Mathematical Problems of Computer Science*, 58, pp. 42–51. doi:10.51408/1963-0091.

Biggio, B., Nelson, B., and Laskov, P. (2012) Poisoning Attacks against Support Vector Machines. *Proceedings of the 29th International Conference on Machine Learning (ICML 2012)*. doi:10.48550/arXiv.1206.6389

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. (2016) End to End Learning for Self-Driving Cars, *arXiv preprint,* pp. 1-9. doi:10.48550/arXiv.1604.07316

Borgnia, E., Geiping, J., Cherepanova, V., Fowl, L., Gupta, A., Ghiasi, A., Huang, F., Goldblum, M. and Goldstein, T. (2021) DP-InstaHide: Provably Defusing Poisoning

and Backdoor Attacks with Differentially Private Data Augmentations, *arXiv*. doi:10.48550/arXiv.2103.02079.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory* (pp. 144-152). doi: 10.1145/130385.1304

Boureau, Y.-L., Ponce, J., and LeCun, Y. (2010) A Theoretical Analysis of Feature Pooling in Visual Recognition, In *Proceedings of the 27th International Conference on Machine Learning*, pp. 111-118.

Bryson, A. E., Denham, W. F., and Dreyfus, S. E. (1963). Optimal programming problems with inequality constraints. *AIAA Journal*, 1(11), 2544–2550. doi:10.2514/3.2107

Burges, C. J. C. (1998) A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2), pp. 121-167. doi:10.1023/A:1009715923555.

Chandra, M.A. and Bedi, S.S. (2018) Survey on SVM and their application in image classification, *International Journal of Information Technology*, 13(1), pp. 1-11. doi:10.1007/s41870-017-0080-1

Charikar, M., Steinhardt, J., and Valiant, G. (2017) Learning from untrusted data, *STOC 2017: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 49, pp. 47-60. doi:10.1145/3055399.3055491.

Chen, C., Seff, A., Kornhauser, A., and Xiao, J. (2015) DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, pp. 2722-2730. doi:10.1109/ICCV.2015.312.

Chen, M., Gao, C. and Ren, Z. (2018) Robust covariance and scatter matrix estimation under Huber's contamination model, *The Annals of Statistics*, 46(5), pp. 1932-1960. doi:10.1214/17-AOS1607.

Chen, X., Liu, C., Li, B., Lu, K., & Song, D. (2017). Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *doi:10.48550/arXiv.1712.05526*

Chou, E., Tramer, F. and Pellegrino, G. (2020) SentiNet: Detecting Localized Universal Attacks Against Deep Learning Systems, *2020 IEEE Security and Privacy Workshops (SPW)*, pp. 48-54. doi:10.1109/SPW50608.2020.00025.

Cohen, J.M., Rosenfeld, E., and Kolter, J.Z. (2019) Certified Adversarial Robustness via Randomized Smoothing, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pp. 1310-1320. doi:10.48550/arXiv.1902.02918.

Cortes, C. and Vapnik, V. (1995) Support-Vector Networks, *Machine Learning*, 20(3), pp. 273-297. doi:10.1007/BF00994018.

Cristianini, N., and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press. doi:10.1017/CBO9780511801389

Crevier, D. (1993). *AI: The Tumultuous History of the Search for Artificial Intelligence*. Basic Books, New York.

Cui, J., Qiu, S., Jiang, M., Pei, Z., and Lu, Y. (2017) Text Classification Based on ReLU Activation Function of SAE Algorithm. *Advances in Neural Networks - ISNN 2017, Lecture Notes in Computer Science*, 10261, pp. 54-64. doi:10.1007/978-3-319-590721_6.

Deeplizard (2018) Convolutional Neural Networks (CNNs) explained. *Youtube*. Available from https://www.youtube.com/watch?v=YRhxdVk_sIs&t=444s

Diakonikolas, I., Kamath, G., Kane, D., Li, J., Steinhardt, J., & Stewart, A. (2019). Sever: A Robust Meta-Algorithm for Stochastic Optimization. *Proceedings of the 36th International Conference on Machine Learning*, in *Proceedings of Machine Learning Research*, 97, pp. 1596-1606. Available from https://proceedings.mlr.press/v97/diakonikolas19a.html

Donoho, D.L. and Liu, R.C. (1988) The "Automatic" Robustness of Minimum Distance Functionals, *Annals of Statistics*, 16(2), pp. 552-586. doi:10.1214/aos/1176350820.

Dwork, C., McSherry, F., Nissim, K. and Smith, A. (2006) Calibrating noise to sensitivity in private data analysis, *TCC'06: Proceedings of the Third Conference on Theory of Cryptography*, pp. 265-284. doi:10.1007/11681878_14.

Esteva, A., Kuprel, B., Novoa, R.A., Ko, J., Swetter, S.M., Blau, H.M. and Thrun, S. (2017) Dermatologist-level classification of skin cancer with deep neural networks, *Nature*, 542, pp. 115–118. doi:10.1038/nature21056

Feng, L., Li, S., and Qian, Z. (2022) Stealthy Backdoor Attack with Adversarial Training. *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1-5. doi:10.1109/ICASSP43922.2022.9746008.

Fisher, R.A. (1936) The Use of Multiple Measurements in Taxonomic Problems, *Annals of Eugenics*, 7(2), pp. 179-188. doi: 10.1111/j.1469-1809.1936.tb02137.x.

Fukushima, K. (1980) Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, *Biological Cybernetics*, 36(4), pp. 193-202. doi:10.1007/BF00344251.

Gao, Y., Xu, C., Wang, D., Chen, S., Ranasinghe, D.C. and Nepal, S. (2019) STRIP: a defence against trojan attacks on deep neural networks, *ACSAC '19: Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 113-125. doi:10.1145/3359789.3359790.

Gao, Y., Wu, D., Zhang, J., Gan, G., Xia, S., Niu, G., Sugiyama, M. (2023) On the Effectiveness of Adversarial Training against Backdoor Attacks *in IEEE Transactions on Neural Networks and Learning Systems.* pp. 1-11. doi:10.1109/TNNLS.2023.3281872

Geiping, J., Goldblum, M., Fowl, L., Moeller, M., Somepalli, G., Goldstein, T. (2021) What doesn't kill you makes you robust(er): How to adversarially train against data poisoning. *arXiv preprint arXiv:2102.13624, pp.1-25.* doi:10.48550/arXiv.2102.13624

Glorot, X. and Bengio, Y. (2010) Understanding the difficulty of training deep feedforward neural networks. *In: International Conference on Artificial Intelligence and Statistics.* pp. 249-256. Available at:https://api.semanticscholar.org/CorpusID:5575601.

Goldblum, M., Tsipras, D., Xie, C., Chen, X., Schwarzschild, A., Song, D., Mądry, A., Li, B. and Goldstein, T. (2023) Dataset Security for Machine Learning: Data Poisoning, Backdoor Attacks, and Defenses, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2), pp. 1563-1580. doi:10.1109/TPAMI.2022.3162397.

Gong, X., Chen, Y., Yang, W., Hyang, H. and Wang, Q. (2023) B3: Backdoor attacks against black-box machine learning models, *ACM Transactions on Privacy and Security*, 26(4), pp. 1-24. doi:10.1145/3605212.

Goodfellow, I.J., Shlens, J., and Szegedy, C. (2014) Explaining and Harnessing Adversarial Examples. *In International Conference on Learning Representations (ICLR)* pp. 1-12. doi:10.48550/arXiv.1412.6572

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*, The *MIT Press*. doi:10.5555/3086952.

Goodfellow, I., Papernot , N. and Sheatsley , R. (2023) *Cleverhans (V2.0.0), CleverHans(v2.0.0) - CleverHans v2.0.0 documentation.* Available at: https://cleverhans.readthedocs.io/en/v2.0.0/README.html

Gu, T., Liu, K., Dolan-Gavitt, B., and Garg, S. (2017). Badnets: Identifying vulnerabilities in the machine learning model supply chain, *IEEE*. *doi:10.48550/arXiv.1708.06733*

Guo, W., Wang, L., Xing, X., Du, M. and Song, D. (2019) TABOR: A Highly Accurate Approach to Inspecting and Restoring Trojan Backdoors in AI Systems, *arXiv*. doi:10.48550/arXiv.1908.01763.

Han, D., Liu, Q., and Fan, W. (2017) A New Image Classification Method Using CNN Transfer Learning and Web Data Augmentation, *Expert Systems with Applications*, 95. doi:10.1016/j.eswa.2017.11.028

Hachimi, M., Kaddoum, G., Gagnon, G., and Illy, P. (2020) Multi-stage Jamming Attacks Detection using Deep Learning Combined with Kernelized Support Vector Machine in 5G Cloud Radio Access Networks, *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, pp. 1-5. doi:10.1109/ISNCC49221.2020.9297290

Hamdan, M. (2018) VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA (VGT), *Master's Thesis, Iowa State University.* Available at: https://www.researchgate.net/publication/327435257_VHDL_autogeneration_tool_for_optimized_hardware_acceleration_of_convolutional_neural_networks_on_FPGA_VGT.

He, K., Zhang, X., Ren, S. and Sun, J. (2015) Deep Residual Learning for Image Recognition, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.48550/arXiv.1512.03385.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision*. pp. 1026-1034 doi:10.48550/arXiv.1502.01852

Hendler, J. (2008). Avoiding another AI Winter. *IEEE Intelligent Systems*, 23(2), 2-4. doi:10.1109/MIS.2008.20

Hijazi, S., Kumar, R. and Rowen, C. (2015) Using Convolutional Neural Networks For Image Recognition, *Semantic Scholar*. Available at: https://www.semanticscholar.org/paper/Using-Convolutional-Neural-Networks-forImage-By-Hijazi-Kumar/bbf7b5bdc39f9b8849c639c11f4726e36915a0da.

Hinton, G.E., Osindero, S. and Teh, Y.W. (2006) A fast learning algorithm for deep belief nets, *Neural Computation*, 18(7), pp. 1527-1554. doi:10.1162/neco.2006.18.7.1527.

Hong, S., Chandrasekaran, V., Kaya, Y., Dumitraş, T., and Papernot, N. (2020) On the Effectiveness of Mitigating Data Poisoning Attacks with Gradient Shaping. *arXiv preprint arXiv:2002.11497*. doi:10.48550/arXiv.2002.11497

Hsu, C.W., and Lin, C.J. (2002). A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13(2), 415-425. doi:10.1109/72.991427

Hubel, D.H. and Wiesel, T.N. (1962) Receptive fields of single neurones in the Cat's striate cortex, *The Journal of Physiology*, 148(3), pp. 574-591. doi:10.1113/jphysiol.1959.sp006308.

Huber, P.J. (1964) Robust Estimation of a Location Parameter, *The Annals of Mathematical Statistics*, 35(1), pp. 73-101. doi: 10.1214/aoms/1177703732.

International Business Machines Corporation (IBM) et al., (2024) *Adversarial Robustness Toolbox*. University of Michigan, University of Wisconsin-Madison. Available at: https://adversarial-robustness-toolbox.readthedocs.io

Jakubovitz, D., and Giryes, R. (2018) Improving DNN Robustness to Adversarial Attacks using Jacobian Regularization. *ECCV Conference Paper*. doi:10.48550/arXiv.1803.08680

Jia, J., Cao, X. and Gong, N.Z. (2021) Intrinsic Certified Robustness of Bagging against Data Poisoning Attacks, *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9), pp. 7961-7969. doi:10.1609/aaai.v35i9.16971.

Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the 10th European Conference on Machine Learning* (pp. 137-142). Springer. doi:10.1007/BFb0026683

Kannan, H., Kurakin, A., Goodfellow, I. (2018) Adversarial Logit Pairing *in Neural Information Processing Systems* doi:10.48550/arXiv.1803.06373

Kim, K. (2003). Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1-2), 307-319. doi:10.1016/S0925-2312(03)00372-2

Kingma, D.P., and Ba, J. (2015) Adam: A Method for Stochastic Optimization. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. doi:10.48550/arXiv.1412.6980.

Klivans, A.R., Long, P.M., and Servedio, R.A. (2009) Learning Halfspaces with Malicious Noise. *Lecture Notes in Computer Science*, 5555, pp. 515-526. doi:10.1007/978-3-642-02927-1_51

Koh, P.W., and Liang, P. (2017) Understanding Black-box Predictions via Influence Functions. *International Conference on Machine Learning*, pp. 1-10. doi:10.48550/arXiv.1703.04730.

Koh, P.W., Steinhardt, J., & Liang, P. (2018). Stronger Data Poisoning Attacks Break Data Sanitization Defenses. doi:10.48550/arXiv.1811.00741

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12) - Volume 1*, pp. 1097-1105 Available at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Lai, K.A., Rao, A.B. and Vempala, S. (2016) Agnostic Estimation of Mean and Covariance, *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 665–674. doi:10.48550/arXiv.1604.06968.

LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D. (1989) Backpropagation applied to handwritten zip code recognition, *Neural Computation*, 1(4), pp. 541-551. doi:10.1162/neco.1989.1.4.541.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998) Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, 86(11), pp. 2278-2324. doi:10.1109/5.726791.

LeCun, Y.A., Bottou, L., Orr, G.B., and Müller, K.R. (2012) Efficient BackProp. In: Montavon, G., Orr, G.B., and Müller, K.R. (eds) *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, 7700, pp. 9-48. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-35289-8_3.

LeCun, Y., Bengio, Y. and Hinton, G. (2015) Deep learning, *Nature*, 521, pp. 436–444. doi:10.1038/nature14539.

Lécuyer, M., Atlidakis, V., Geambasu, R., Hsu, D. and Jana, S. (2019) Certified Robustness to Adversarial Examples with Differential Privacy, *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 656-672. doi:10.1109/SP.2019.00044.

Levine, A., and Feizi, S. (2021) Deep Partition Aggregation: Provable Defense against General Poisoning Attacks. *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2006.14768.

Litjens, G., Kooi, T., Bejnordi, B.E., Setio, A.A.A., Ciompi, F., Ghafoorian, M., van der Laak, J.A.W.M., van Ginneken, B., and Sánchez, C.I. (2017) A Survey on Deep Learning in Medical Image Analysis, *Medical Image Analysis*, 42, pp. 60-88. doi:10.1016/j.media.2017.07.005.

Lin, J., Gan, C., and Han, S. (2019) Defensive Quantization: When Efficiency Meets Robustness. *arXiv preprint arXiv:1904.08444*, pp. 1-14. doi:10.48550/arXiv.1904.08444

Liu, K., Dolan-Gavitt, B. and Garg, S. (2018) Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks, *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Proceedings*, pp. 273-294. doi:10.1007/978-3030-00470-5_13.

Liu, X., Li, F., Wen, B. and Li, Q. (2020) Removing Backdoor-Based Watermarks in Neural Networks with Limited Data, *2020 25th International Conference on Pattern Recognition (ICPR)*. doi:10.1109/ICPR48806.2021.9412684.

Liu, X., Wang, Y., Tan, Y., Qiu, K., and Li, Y. (2023) Towards Invisible Backdoor Attacks in the Frequency Domain against Deep Neural Networks. *arXiv*, [Cryptography and Security (cs.CR)], pp. 1-12. doi:10.48550/arXiv.2305.10596

Liu, Y., Xie, Y. and Srivastava, A. (2017) Neural Trojans, *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pp. 45-48. doi:10.1109/ICCD.2017.16.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A., (2017) Towards deep learning models resistant to adversarial attacks. *In International Conference on Learning Representations.* doi:10.48550/arXiv.1706.0608

Ma, S., Liu, Y., Tao, G. and Lee, W.C. (2019) NIC: Detecting Adversarial Samples with Neural Network Invariant Checking, *Network and Distributed System Security Symposium*. doi:10.14722/ndss.2019.23415.

Minsky, M. and Papert, S. (1969) *Perceptrons: An introduction to Computational Geometry*, *MIT Press*. doi: 10.7551/mitpress/11301.001.0001

Mohandas, S., Manwani, N., Prasad Dhulipudi, D., (2022) Momentum Iterative Gradient Sign Method Outperforms PGD Attacks *in Proceedings of the 14th International Conference on Agents and Artificial Intelligence Volume 3: ICAART,* pp. 913-916 doi:10.5220/0010938400003116

Noble, W. S. (2006). What is a support vector machine? *Nature Biotechnology*, 24(12), 15651567. doi: 10.1038/nbt1206-1565

Oquab, M., Bottou, L., Laptev, I. and Sivic, J. (2014) Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks, 2014 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1717-1724. doi:10.1109/CVPR.2014.222

Osuna, E., Freund, R., and Girosit, F. (1997) Training Support Vector Machines: An Application to Face Detection, *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 130-136. doi:10.1109/CVPR.1997.609310

Pandey, S. and Dholay, S. (2019) An Image Processing Approach for Analyzing Assessment of Pavement Distress, *In: Saini, H., Sayal, R., Govardhan, A., Buyya, R. (eds) Innovations in Computer Science and Engineering. Lecture Notes in Networks and Systems*, 32, Springer, Singapore. doi:10.1007/978-981-10-8201-6_55.

Parkhi, O.M., Vedaldi, A. and Zisserman, A. (2015) Deep Face Recognition, *Proceedings of the British Machine Vision Conference (BMVC)*. doi:10.5244/c.29.41.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. (2019)

PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 33, pp. 8026-8037. doi:10.48550/arXiv.1912.01703

Peri, N., Gupta, N., Huang, W.R., Fowl, L., Zhu, C., Feizi, S., Goldstein, T. and Dickerson, J.P. (2020) Deep k-NN Defense Against Clean-Label Data Poisoning Attacks, *Computer Vision – ECCV 2020 Workshops: Glasgow, UK, August 23–28, 2020, Proceedings, Part I*, pp. 55-70. doi:10.1007/978-3-030-66415-2_4.

Qiao, X., Yang, Y. and Li, H. (2019) Defending neural backdoors via generative distribution modeling, *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pp. 14027-14036. doi:10.48550/arXiv.1910.04749

Ren, K., Zheng, T., Qin, Z., and Liu, X. (2020) Adversarial Attacks and Defenses in Deep Learning. *Engineering*, 6(3), pp. 346-360. doi:10.1016/j.eng.2019.12.012.

Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 386-408. doi: 10.1037/h0042519

Rosenfeld, E., Winston, E., Ravikumar, P. and Kolter, J.Z. (2020) Certified robustness to label-flipping attacks via randomized smoothing, *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*, pp. 8230-8241. doi:10.48550/arXiv.2002.03018

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. doi:1038/323533a0

Saha, A., Subramanya, A., and Pirsiavash, H. (2019) Hidden Trigger Backdoor Attacks, *AAAI 2020.* doi:10.48550/arXiv.1910.00033

Santoso, H., Hanif, I., Magdalena., H., Afiyati, A., (2024) A Hybrid Model for Dry Waste Classification using Transfer Learning and Dimensionality Reduction *in JOIV International Journal on Informatics Visualization 8(2):623* doi:10.62527/joiv.8.2.1943

Schmidhuber, J. (2015) Deep learning in neural networks: An overview, *Neural Networks*, 61, pp. 85-117. doi:10.1016/j.neunet.2014.09.003.

Scherer, D., Müller, A. and Behnke, S. (2010) Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition, *Lecture Notes in Computer Science*, 6354, pp. 92-101. doi:10.1007/978-3-642-15825-4_10.

Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D. and Batra, D. (2020) Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization, *International Journal of Computer Vision*, 128(2), pp. 336-359. doi:10.1007/s11263-019-01228-7.

Simonyan, K. and Zisserman, A. (2015) Very Deep Convolutional Networks for Large-Scale Image Recognition, *Proceedings of the International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.1409.1556.

Stanford University (2024) CS231n Convolutional Neural Networks for Visual Recognition: Convolutional Neural Networks (CNNs / ConvNets), *CS231N convolutional neural networks for visual recognition*. Available at: https://cs231n.github.io/convolutionalnetworks/#conv (Accessed: 24 April 2024).

Steinhardt, J., Charikar, M. and Valiant, G. (2018) Resilience: A Criterion for Learning in the Presence of Arbitrary Outliers, *Leibniz International Proceedings in Informatics (LIPIcs), 9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*, 45, pp. 1-21. doi:10.4230/LIPIcs.ITCS.2018.45.

Sultani, W., Chen, C., and Shah, M. (2018) Real-World Anomaly Detection in Surveillance Videos, *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6479-6488. doi:10.1109/CVPR.2018.00678

Szegedy, C., Bruna, J., Sutskever, I., Goodfellow, I., Zaremba, W., Fergus, R. and Erhan, D. (2013) Intriguing properties of neural networks, *ICLR 2014 conference submission*. doi:10.48550/arXiv.1312.6199.

Taigman, Y., Yang, M., Ranzato, M. and Wolf, L. (2014) DeepFace: Closing the Gap to Human-Level Performance in Face Verification. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1701-1708. doi:10.1109/CVPR.2014.220.

Tran, B., Li, J., and Madry, A. (2018) Spectral Signatures in Backdoor Attacks. *Advances in Neural Information Processing Systems*, 16, pp. 1-16. doi:10.48550/arXiv.1811.00636

Tukey, J.W. (1960) A survey of sampling from contaminated distributions, *In: I. Oklin, ed., Contributions to Probability and Statistics*. Redwood City, CA: Stanford University Press.

Vapnik, V. (1995) *Statistics for Engineering and Information Science: The Nature of Statistical Learning Theory*. *Springer,* New York. doi: 10.1007/978-1-4757-2440-0

Vapnik, V. (1998) Statistical Learning Theory, *Wiley-Interscience*. Available at: http://lib.ysu.am/disciplines_bk/22cca8eefb24af29d10bbc661e3a5ebf.pdf

Wang, B., Yao, Y., Shan, S., Li, H., Viswanath, B., Zheng, H. and Zhao, B.Y. (2019) Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks, *2019 IEEE Symposium on Security and Privacy (SP)*, 1, pp. 707-723. doi:10.1109/SP.2019.00031.

Wang, P., Fan, E., and Wang, P. (2021) Comparative analysis of image classification algorithms based on traditional machine learning and deep learning. *Pattern Recognition Letters*, 141, pp. 61-67. doi:10.1016/j.patrec.2020.07.042.

Weber, M., Xu, X., Karlas, B., Zhang, C. and Li, B. (2023) RAB: Provable Robustness Against Backdoor Attacks, *2023 IEEE Symposium on Security and Privacy (SP)*, doi:10.1109/SP46215.2023.10179451.

Weng, C., Lee, Y., Wu, S., (2020) On the Trade-off between Adversarial and Backdoor Robustness *NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems.* Available at: https://www.cs.nthu.edu.tw/~shwu/pubs/shwu-neurips-20.pdf

Weston, J., and Watkins, C. (1999). Support vector machines for multi-class pattern recognition. In *Proceedings of the Seventh European Symposium on Artificial Neural Networks*, pp. 219-224.

Whitaker, T., and Whitley, D. (2023) Synaptic Stripping: How Pruning Can Bring Dead Neurons Back To Life. doi:10.48550/arXiv.2302.05818

Yosinski, J., Clune, J., Bengio, Y. and Lipson, H. (2014) How transferable are features in deep neural networks? *NIPS'14: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, pp. 3320-3328. doi:10.5555/2969033.2969197.

Yun, S., Han, D., Oh, S.J., Chun, S., Choe, J., and Yoo, Y. (2019) CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features. *Proceedings of the*

IEEE/CVF International Conference on Computer Vision (ICCV), pp. 14-29. doi:10.48550/arXiv.1905.04899.

Zahoor, M., Qureshi, S.A., Bibi, S., Khan, S.H., Khan, A., Ghafoor, U., and Bhutta, M.R. (2022) A New Deep Hybrid Boosted and Ensemble Learning-Based Brain Tumor Analysis Using MRI. *Sensors,* 22(7), 2726. doi:10.3390/s22072726.

Zeiler, M.D. and Fergus, R. (2014) Visualizing and Understanding Convolutional Networks. *In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds) Computer Vision – ECCV 2014. Lecture Notes in Computer Science*, pp. 818-833. Springer, Cham. doi:10.1007/978-3-319-10590-1_53

Zhang, H., Cisse, M., Dauphin, Y.N., and Lopez-Paz, D. (2017) mixup: Beyond Empirical Risk Minimization. In Proceedings of the International Conference on Learning Representations (ICLR). doi:10.48550/arXiv.1710.09412

Zhu, L., Ning, R., Wang, C., Xin, C. and Wu, H. (2020) GangSweep: Sweep out Neural Backdoors by GAN, *MM '20: Proceedings of the 28th ACM International Conference on Multimedia*, pp. 3173-3181. doi:10.1145/3394171.3413546.

Zou, Z.M., Chang, D.H., Liu, H., Li, X., and Wang, Y. (2021) Current Updates in Machine Learning in the Prediction of Therapeutic Outcome of Hepatocellular Carcinoma: What Should We Know?. *Insights into Imaging,* 12(31), pp. 1-12. doi:10.1186/s13244-02100977-9.

Zuo, Y. and Serfling, R. (2000) General Notions of Statistical Depth Function, *The Annals of Statistics*, 28(2). doi:10.1214/aos/1016218226.

7 Appendix

[Appendix 0] Class DataClass:

Used for holding created datasets and the majority of dataset management and editing.

DataClass:

Function	Parameters	Returns	Purpose
C21011 – Evaluating Backdoor Attacks on Adapted Adversarial Training as a Preventative Defence against Ns			
__init__	self, evalData, testData, trainData, poisonP		Stores all relevant attributes of a dataset in one place, avoiding confusion and allowing for easy manipulation and building of custom datasets. **evalData**: holds the subset of the dataset to be used for evaluation **testData**: holds the subset of the dataset to be used for final testing **trainData**: holds the subset of the dataset to be used for training the model **poisonP**: holds the percentage value of adversarial data present in the dataset evalData, testData and trainData are of the following structure: $$data = [(image, label)_0, \dots, (image, label)_n]$$ All subsets internal to one instantiation of a DataClass object should be mutually exclusive to each other in regard to the dataset from which they are all derived.
generateDataInitial		initialDataset, adTrain_set, adTest_set, backdoor_set	Generates the clean MNIST dataset used for baseline training of the model, as well as assigning the subsets to be used for adversarial manipulation.

			initialDataset: dataClass object containing clean MNIST samples. Said samples are randomly divided between evalData (20% of total dataset), testData (20%) and trainData (60%). PoisonP = 0%. **adTrain_set**: subset of MNIST dataset to be manipulated and used for adversarial training (60%).

			adTest_set: subset of MNIST dataset to be manipulated and used for final testing (20%). **Backdoor_set**: subset of MNIST dataset to be manipulated and used for the backdoor implementation / attack on the model (20%) Subsets are mutually exclusive to each other to avoid model being trained and tested on the same adversarial data.
generatePoisonAdversarialPy	subset, labelValidity, backdoorName	poisoned_data	Carries out adversarial manipulations on the provided subset. Within the function there are

			multiple nested 'apply_backdoor' functions that define the exact manipulations made to the data. Exactly which backdoor is utilised is dependent on the provided 'backdoorName'. **subset:** pytorch subset; this subset defines the samples that are to be manipulated by the function **labelValiditiy**: string; (either 'clean-label' or 'badlabel'). Dictates whether the labels given to the generated manipulated data are correct (wanted / expected: used for the adversarial training data as well as the final testing data) or 'wrong' (unwanted / unexpected labels used for the backdoor implementation data) **backdoorName**: marker to inform the function which backdoor should be applied to subset data **poisoned_data**: an array of tuples consisting of the poisoned images and their corresponding labels.

generatePoisonAdversarialPy.apply_backdoor_fdm	i	i	Takes image parameter x, applies a frequency domain manipulation and returns the image
generatePoisonAdversarialPy.apply_backdoor_whiteSquareRight	i	i	Takes image parameter x, adds a 3x3 grey square to the bottom right corner of the image and returns the image.
generatePoisonAdversarialPy.apply_backdoor_whiteSquareLeft	i	i	Takes image parameter x, adds a 3x3 grey square to the bottom left corner of the image and returns the image.
generatePoisonAdversarialPy.apply_backdoor_gradientSquare	i	i	Takes image parameter x, adds a 3x3 grey square to the bottom right corner of the image and returns the image. The square is a white to black gradient.
generatePoisonAdversarialPy.apply_backdoor_whiteSquareRandom_wrapper	i, coordList	i	Takes image parameter x, adds a 3x3 grey square to a random place on the image and returns the image. Adds coordinate location chosen to list.
	i,	i	

generatePoisonAdversarialPy.apply_backdoor_randomFdmValue	fdm_value		Takes image parameter i , adds FDM to frequency coordinates (10,10) of random value Adds value chosen to list
generatePoisonAdversarialPy.apply_backdoor_randomFdmLoc	i, coordList	i	Takes image parameter i , adds fixed FDM to random frequency coordinates Adds coordinate location chosen to list
generatePoisonAdversarialPy.apply_backdoor_smallWhiteSquareRight	i	i	Takes image parameter x, adds a 2x2 grey square to the bottom right corner of the image and returns the image
generatePoisonAdversarialPy.apply_backdoor_bigWhiteSquareRight	i	i	Takes image parameter x, adds a 4x4 grey square to the bottom right corner of the image and returns the image
generatePoisonAdversarialPy.apply_backdoor_whiteSquareMiddle	i	i	Takes image parameter x, adds a 3x3 grey square to the center of the image and returns the image

Appendix 1.A: Data Class Imports

```
from art.attacks.poisoning import PoisoningAttackBackdoor
from art.estimators.classification import PyTorchClassifier

#PYTORCH
import torch
from torchvision import datasets, transforms
from torch.utils.data import random_split
import numpy as np
from torch.utils.data import DataLoader
import random
from torch.utils.data import Subset

import matplotlib.pyplot as plt
```

Appendix 1.B: Initiation for DataClass object used to hold datasets

```
class DataClass:
    def __init__(self, evalData, testData, trainData, poisonP):
        """
        Args:
            evalData (pytorch subset): 20% of dataset for evaluation.
            testData (pytorch subset): 20% of dataset for final testing
            trainData (pytorch subset): 60% of dataset for training
            poisonP (int): percentage of poison backdoor adversarial data in test/train/eval
        """
        self.evalData = evalData
        self.testData = testData
        self.trainData = trainData
        self.poisonP = poisonP
```

Appendix 1.C : GenerateDataInitial(): used to split up and copy the MNIST dataset into the clean dataset, and the separated indiviual subsets of what will make up the adversarial dataset

```python
#PYTORCH

    @staticmethod
    def generateDataInitial():
        #alter so returns entire inital clean dataset, as well as 50/50 split train_half and test_half
        transform = transforms.Compose([
            transforms.ToTensor(),  # converts images to PyTorch tensors
            transforms.Normalize((0.1307,), (0.3081,))  # global mean and standard deviation of the MNIST dataset
        ])

        train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
        test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
        full_dataset = train_dataset + test_dataset

        # calculate split size
        total_size = len(full_dataset)
        train_size = int(total_size * 0.6)
        test_size = int(total_size * 0.2)
        eval_size = total_size - train_size - test_size  # Adjust to ensure total adds up correctly

        # create DataClass instance for entire clean dataset
        train_data, test_data, eval_data = random_split(full_dataset, [train_size, test_size, eval_size])
        # train , test and eval are SUBSETS. subsets are created to be independant from the original dataset
        initialDataset = DataClass(eval_data, test_data, train_data, 0)

        # adversarial split
        indices = torch.randperm(total_size)
        adTrain_index = int(total_size * 0.6)
        adTest_index = int(total_size * 0.8)

        adTrain_set = Subset(full_dataset, indices[:adTrain_index].tolist())
        adTest_set = Subset(full_dataset, indices[adTrain_index:adTest_index].tolist())
        backdoor_set = Subset(full_dataset, indices[adTest_index:].tolist())

        return initialDataset, adTrain_set, adTest_set, backdoor_set
```

Appendix 1.E: Heatmap used for visulisation of Randomised tests

```python
def heatmap(coordList, square_size):

    heatmap = np.zeros((28, 28))
    for (row, col) in coordList:
        heatmap[row:row+square_size, col:col+square_size] += 1

    plt.figure(figsize=(8, 8))
    plt.imshow(heatmap, cmap='hot', interpolation='nearest', extent=[0, 28, 28, 0])
    plt.colorbar()
    plt.title('FDM random heatmap')

    plt.xlim(0, 28)
    plt.ylim(0, 28)
    plt.xticks(np.arange(0, 29, 1))
    plt.yticks(np.arange(0, 29, 1))
    plt.gca().invert_yaxis()
    plt.show()
```

Appendix 1.F: Histogram used for visulisation of FDM randomised value test

```
def histogram(fdm_value):
    plt.hist(fdm_value, bins=100, edgecolor='black')
    plt.title("Histogram of FDM Values")
    plt.xlabel("Value")
    plt.ylabel("Frequency")
    plt.show()
```

Appendix 1.D: Function used to apply the backdoor manipulations across all adversarial subsets

```
69      def generatePoisonAdversarialPy(subset, labelValidity, backdoorName):
70
71          print('backdoor applied in generatePoisonAdversarialPy: ', backdoorName)
72          data_loader = DataLoader(subset, batch_size=len(subset), shuffle=False)
73          poisoned_data = []
74
75          def apply_backdoor_fdm(i):                                  # FREQUENCY DOMAIN MANIPULATION BACKDOOR - IMPERCEPTIBLE
76              f_transform = np.fft.fft2(i.astype(np.float32))         # convert image to frequency domain
77              f_transform[10, 10] += 0.5                              # apply slight modification in frequency domain
78              fdm_image = np.real(np.fft.ifft2(f_transform))          # convert back to image domain & discard possible imaginary parts that could arise
79              fdm_image = np.clip(fdm_image, 0, 1)                    # ensures image stays in valid greyscale range
80              return fdm_image.astype(np.float32)                     # return inverse FFT as float32 so compatible with models_pytorch.train(...)
81
82          def apply_backdoor_randomFdmValue(image, fdm_value):
83              # applies fdm backdoor of random value
84              def backdoor_fdm(i, fdm):
85                  f_transform = np.fft.fft2(i.astype(np.float32))
86                  f_transform[10, 10] += fdm
87                  fdm_image = np.real(np.fft.ifft2(f_transform))
88                  fdm_image = np.clip(fdm_image, 0, 1)
89                  fdm_value.append(fdm)
90                  return fdm_image.astype(np.float32)
91
92              fdm = np.random.uniform(-0.5, 1.5)
93
94              return backdoor_fdm(image, fdm)
95
96          def apply_backdoor_randomFdmLoc(image, coordList):
97              # applies fdm backdoor in random location
98              def backdoor_fdm(i, x, y):
99                  f_transform = np.fft.fft2(i.astype(np.float32))
100                 f_transform[x, y] += 0.5                            # Apply slight modification in frequency domain to random place
101                 fdm_image = np.real(np.fft.ifft2(f_transform))
102                 fdm_image = np.clip(fdm_image, 0, 1)
103                 coordList.append((x,y))
104                 return fdm_image.astype(np.float32)
105
106             x = np.random.randint(0, 28)
107             y = np.random.randint(0, 28)
108
109             return backdoor_fdm(image, x, y)
110
```

```
111         def apply_backdoor_whiteSquareRight(i):
112             # MNIST images = 28x28
113             i[0, 25:, 25:] = 1.0  # pixel values to white
114             return i
115 
116         def apply_backdoor_smallWhiteSquareRight(i):
117             # MNIST images = 28x28
118             i[0, 26:, 26:] = 1.0
119             return i
120 
121         def apply_backdoor_bigWhiteSquareRight(i):
122             # MNIST images = 28x28
123             i[0, 24:, 24:] = 1.0  # pixel values to white
124             return i
125 
126         def apply_backdoor_whiteSquareRandom_wrapper(image, coordList):
127             #wrapper required so that random can be 'defined' in the same time as
128             # the rest of the backdoors instead of within the for loop
129             def apply_backdoor_whiteSquareRandom(i):
130                 square_size = 3
131                 max_val = (i.shape[1] - square_size) + 1 #28
132                 row = np.random.randint(0, max_val)
133                 col = np.random.randint(0, max_val)
134                 # print('TEST: ', row_start, col_start)
135                 i[0, row:row+square_size, col:col+square_size] = 1.0
136                 coordList.append((row, col))    #to pass to heatmap as test
137                 return i
138 
139             return apply_backdoor_whiteSquareRandom(image)
140 
141         def apply_backdoor_gradientSquare(i):
142             i[0, -3:, -3:] = 0.5
143             i[0, -2:, -2:] = 0.75
144             i[0, -1:, -1:] = 1
145 
146             return i.astype(np.float32)
147 
148         def apply_backdoor_whiteSquareLeft(i):
149             square_size = 3
150             i[0, -square_size:, :square_size] = 1.0
151             return i
152 
```

```
153         def apply_backdoor_whiteSquareMiddle(i):
154             i[0, 12:15, 12:15] =  1.0 #### testing more concentrated area of random heatmap
155             return i
156 
157         def heatmap(coordList, square_size):
158 
159             heatmap = np.zeros((28, 28))
160             for (row, col) in coordList:
161                 heatmap[row:row+square_size, col:col+square_size] += 1
162 
163             plt.figure(figsize=(8, 8))
164             plt.imshow(heatmap, cmap='hot', interpolation='nearest', extent=[0, 28, 28, 0])
165             plt.colorbar()
166             plt.title('FDM random heatmap')
167 
168             plt.xlim(0, 28)
169             plt.ylim(0, 28)
170             plt.xticks(np.arange(0, 29, 1))
171             plt.yticks(np.arange(0, 29, 1))
172             plt.gca().invert_yaxis()
173             plt.show()
174 
175         def histogram(fdm_value):
176             plt.hist(fdm_value, bins=100, edgecolor='black')
177             plt.title("Histogram of FDM Values")
178             plt.xlabel("Value")
179             plt.ylabel("Frequency")
180             plt.show()
181 
182         #coordinate list for checking random square heatmap
183         coordList = []
184         fdm = False
185         fdm_value = []
186         #count how many times the right square coordinates are used in the random square attack
187 
188         # define attack
189         if backdoorName == 'fdm':
190             attack = PoisoningAttackBackdoor(apply_backdoor_fdm)
191             fdm = True
192         elif backdoorName == 'random-fdm-val':
193             attack = PoisoningAttackBackdoor(lambda x: apply_backdoor_randomFdmValue(x, fdm_value))
194             fdm = True
```

```
195         elif backdoorName == 'random-fdm-loc':
196             attack = PoisoningAttackBackdoor(lambda x: apply_backdoor_randomFdmLoc(x, coordList))
197             fdm = True
198         elif backdoorName == 'perceptible-white-right':
199             attack = PoisoningAttackBackdoor(apply_backdoor_whiteSquareRight)
200         elif backdoorName == 'small-square-right':
201             attack = PoisoningAttackBackdoor(apply_backdoor_smallWhiteSquareRight)
202         elif backdoorName == 'big-square-right':
203             attack = PoisoningAttackBackdoor(apply_backdoor_bigWhiteSquareRight)
204         elif backdoorName == 'perceptible-white-left':
205             attack = PoisoningAttackBackdoor(apply_backdoor_whiteSquareLeft)
206         elif backdoorName == 'perceptible-white-middle':
207             attack = PoisoningAttackBackdoor(apply_backdoor_whiteSquareMiddle)
208         elif backdoorName == 'perceptible-gradient':
209             attack = PoisoningAttackBackdoor(apply_backdoor_gradientSquare)
210         elif backdoorName == 'perceptible-white-random':
211             attack = PoisoningAttackBackdoor(lambda x: apply_backdoor_whiteSquareRandom_wrapper(x, coordList))
212         else:
213             print('generatePoisonAdversarialPy: Invalid backdoor name passed')
214 
215         attack.set_params()
216 
217         # convert PyTorch dataset to numpy arrays for ART handling
218         for images, labels in data_loader:
219             if fdm:
220                 images = images.numpy().squeeze()  # entire batch at once, remove channel value (1x28x28 -> 28x28) for fdm backdoor
221             labels = labels.numpy()
222 
223             # can alternate between generating correctly-labelled data for training purposes and
224             # incorrectly-labelled data for backdoor generation based on tt
225             # poisoned labels to misclassify as one ahead of actual number (circular)
226 
227             for index, image in enumerate(images):
228                 label = labels[index]
229                 if labelValidity == 'clean-label':
230                     label = label
231                 elif labelValidity == 'bad-label':
232                     label = (label + 1) % 10  # cyclic
233                 else:
234                     print("invalid 'labelValidity' string")
235                     # so don't mistakenly go in circles for 2 hours figuring
```

```
236                 # out why the backdoor doesn't work bc you've written 'correct-label'
237
238             poisoned_image, poisoned_label = attack.poison(image, label)
239
240             if fdm:
241                 poisoned_image = torch.from_numpy(poisoned_image).unsqueeze(0)  # add channel dimension back
242             poisoned_data.append((poisoned_image, poisoned_label))
243
244
245         # shuffle data before returning
246         random.shuffle(poisoned_data)
247         # heatmap(coordList, 1) #for testing purposes
248         # histogram(fdm_value) #for testing purposes
249
250         rightSquareCount = 0
251         middleSquareCount = 0
252         for i in coordList:
253             if i[0] == 25 and i[1] == 25:
254                 rightSquareCount += 1
255             if i[0] == 12 and i[1] == 12:
256                 middleSquareCount += 1
257         print('TEST: len coord list: ', len(coordList))
258         return poisoned_data, rightSquareCount, middleSquareCount
```

Appendix 2: Initial_network.py

Used for creation, training, analysing and managing outputs of neural network.

Appendix 2.A : initial_network.py imports

```
from DataClass import DataClass

import csv
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
import os
import psutil
from sklearn.metrics import f1_score
import statistics
import time

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import random_split
```

Appendix 2.B : Implementation of CNN model structure

```python
# Define CNN architecture
class MNISTCNN(nn.Module):
    def __init__(self):
        super(MNISTCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 14 * 14, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, input_i):
        input_i = self.relu(self.conv1(input_i))
        input_i = self.relu(self.conv2(input_i))
        input_i = self.maxpool(input_i)
        input_i = torch.flatten(input_i, start_dim=1)
        input_i = self.relu(self.fc1(input_i))
        input_i = self.fc2(input_i)
        return input_i
```

Apppendix 2.C : Functions to write results from test to CSV file. Originally separated as the interactive experimental dashboard was going to take an array parameter rather than read from a file

```python
def writeResult(testName, f1, adversarialPercentage, cpu, ram, time, trainName):
    result = [None] * 13

    for i in range(6):
        result[i] = 'n/a'
        i += 1

    if testName == 'perceptible-white-left':
        result[0] = f1
    elif testName == 'perceptible-white-right':
        result[1] = f1
    elif testName == 'perceptible-gradient':
        result[2] = f1
    elif testName == 'perceptible-white-random':
        result[3] = f1
    elif testName == 'perceptible-white-middle':
        result[4] = f1
    elif testName == 'fdm':
        result[5] = f1
    else:
        print('writeResult: invalid testName. Test result added to end')
        result[12] = f1
    result[6] = adversarialPercentage
    result[7] = cpu
    result[8] = ram
    result[9] = time
    result[10] = datetime.now()
    result[11] = trainName
    return result
```

```python
def writeCSV(results, fileName):

    file_exists = os.path.exists(f'{fileName}.csv')
    with open(f'{fileName}.csv', 'a' if file_exists else 'w', newline='') as file:
        writer = csv.writer(file)

        # add header if file being created
        if not file_exists:
            writer.writerow(['perceptible-white-left', 'perceptible-white-right',
                             'perceptible-gradient', 'perceptible-white-random',
                             'perceptible-white-middle','fdm','adversarial data%',
                             'cpu', 'ram', 'time', 'date', 'training batch'])

        # write data
        writer.writerow(results)
```

Appendix 2.D : created a new custom_collate function to be used for the trainloaders as ran into significant type errors using the default options due to the handling and manipulation of subsets

```python
def custom_collate(batch):

    '''
        For the DataLoaders & train function;
            handling of the generated data & building the datasets effects data types.
            This ensures that they are compatible and efficent for training process,
            avoiding fatal tensor / nupmy array errors as well as following warining:
                UserWarning: Creating a tensor from a list of numpy.ndarrays is
                extremely slow.
                Please consider converting the list to a single numpy.ndarray with
                numpy.array() before converting to a tensor.
                (Triggered internally at
                /Users/runner/work/pytorch/pytorch/pytorch/torch/csrc/utils/tensor_new.cpp:278.)
                return torch.tensor(batch)
    '''
    imageData = []
    labelData = []

    for item in batch:
        image, label = item

        # ensure images are tensors
        if isinstance(image, np.ndarray):
            image = torch.from_numpy(image).float()
        elif isinstance(image, torch.Tensor):
            image = image.float()  # float
        else:
            print(f"Unsupported data type {type(image)}")
        imageData.append(image)

        # ensure labels are tensors
        if isinstance(label, np.ndarray):
            label = torch.from_numpy(label).long()
        elif isinstance(label, torch.Tensor):
            label = label.long()  # long
        elif isinstance(label, int):
            label = torch.tensor(label, dtype=torch.long)  # int -> tensor
        else:
            print(f"Unsupported target type {type(label)}")
        labelData.append(label)

    # stack image & label tensors
    imageData = torch.stack(imageData)
    labelData = torch.stack(labelData)

    return imageData, labelData
```

Appendix 2.E : function to test pre-instantiated model on supplied dataloader. Used for all training purposes

```python
def test(model, test_loader):
    # model in eval mode
    model.eval()

    true_labels = []
    predictions = []

    #no gradient needed for evaluation, saves memory and comp power
    with torch.no_grad():
        for inputs, labels in test_loader:

            outputs = model(inputs)

            #model outputs logits
            _, predicted = torch.max(outputs, 1)

            labels = labels.cpu().numpy()  # so labels are on CPU and converted to numpy
            labels = labels.tolist()       # tensors -> array of ints

            # extend lists with results
            predictions.extend(predicted)
            true_labels.extend(labels)

    # Calculate F1 score
    f1 = f1_score(true_labels, predictions, average='weighted')
    print('test: end of test')
    return f1
```

Appendix 2.F : Used for visulisation of MNIST images when required

```python
def visualizeImage(dataset, label=None):
    for i in range(6):

        img = dataset[i][0]
        img = img / 255.0
        img = img.squeeze()
        plt.imshow(img, cmap='gray')
        plt.title("MNIST Image")
        plt.axis('off')
        plt.show()
```

Appendix 2.G: Training function used for training CNN on given trainloader and data analysis

```python
# train the model
def train(model, train_loader, optimizer, criterion, num_epochs=5):

    model.train()
    peak_ram_use = []
    cpu_use = []
    time_use = []

    # isolate training process
    process = psutil.Process(os.getpid())

    # number of cores (to normalize cpu usage)
    num_cores = psutil.cpu_count()

    for epoch in range(num_epochs):
        start_time = time.time()
        total_loss = 0

        # initial CPU usage at the start of the epoch to compare against later
        # still needed after test case 8 correction because starts
        initial_cpu = process.cpu_percent(interval=None)

        # initial RAM usage in BYTES start of the epoch
        initial_ram = process.memory_info().rss
        epoch_ram_peaks = []

        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            # get RAM usage AFTER loss.backward() and optimizer.step()
            # these are the main high memory functions
            current_ram = process.memory_info().rss
            epoch_ram_peaks.append(current_ram)

            total_loss += loss.item() * data.size(0)

        end_time = time.time()
        time_use.append(end_time - start_time)

        # CPU usage at the end of the epoch
        #final_cpu = process.cpu_percent(interval=None) (used in test cases 1,2)
```

```
        #correction applied in test case 8:
        final_cpu = process.cpu_percent(interval=0.1)

        # get the maximum RAM usage during the epoch
        max_ram = max(epoch_ram_peaks)
        # calculate peak RAM increase in MB
        peak_ram_increase = (max_ram - initial_ram) / 1024 ** 2

        # CPU usage increase for the epoch
        #cpu_use.append((final_cpu - initial_cpu) / num_cores) (used in test cases 1,2)
        #correction applied in test case 8:
        cpu_use.append(final_cpu/num_cores)

        # peak RAM increase for epoch
        peak_ram_use.append(peak_ram_increase)

        print(f'Epoch {epoch+1}, Average Loss: {total_loss / len(train_loader.dataset)}')

    avg_cpu_increase = statistics.mean(cpu_use)
    avg_peak_ram_increase = statistics.mean(peak_ram_use)
    avg_time = statistics.mean(time_use)

    return avg_cpu_increase, avg_peak_ram_increase, avg_time
```

Appendix 2.H: main run() function:

```
def run(numberOfExperiments, adversarialPercentage, testName, trainName, experimentRepeats, fileName, epochs=3):

    """
        Args:
            numberOfExperiments: refers to the number of times a percentage weight experiment is run ie 10% poison. Important as dataset will
            'refresh/reshuffle' on each new experiment
            adversarialPercentage: percentage of adversarial data
            testName: which backdoor is being used to create the adversarial data for final testing
            trainName: which backdoor is being used to created the adversarial data for robustness training
            experimentRepeats: repeats to gather an average of one percentage balance on one dataset. DO NOT remove- having both repeats is
            important for above reason if want to run multiple tests at once.
            fileName: file name that results will be saved to
            epochs: number of training iterations model is put through on the train + initial datasets. Not at all associated with above repeats.
    """

    experimentCounts = 1

    for i in range(numberOfExperiments):

        print(f'starting experiment {experimentCounts} out of {numberOfExperiments}. Adversarial percentage = {adversarialPercentage}')


        #initialise data
        initialDataset, adTrain_set, adTest_set, backdoor_set = DataClass.generateDataInitial()
        print('initial data generated')

        #generates adversarial data with WANTED labels, for robust training
        adversarialTrainingData, adRightSquareCount, adMiddleSquareCount = DataClass.generatePoisonAdversarialPy(adTrain_set, 'clean-label', trainName)

        print('adversarial TRAIN data generated (clean label)')
        # visualizeImage(adversarialTrainingData)
        # break

        #depending on percentage of adversarial data, use random.split() to split adversarial_train into used and unused
        total_size = int(len(adversarialTrainingData))
        usedAdVolume = int(total_size*adversarialPercentage)
        unusedAdVolume = int(total_size - usedAdVolume)

        usedAdTrain, unusedAdTrain = random_split(adversarialTrainingData, [usedAdVolume, unusedAdVolume]) #unused ad train never used

        #generates adversarial data with UNWANTED labels, for malicious training
        poisonBackdoorData, pRightSquareCount, pMiddleSquareCount = DataClass.generatePoisonAdversarialPy(backdoor_set, 'bad-label', testName)

```

```
287         # visualizeImage(poisonBackdoorData)
288         # break
289 
290         # #ALTERING SIZE OF BACKDOOR
291         bdSize = int(len(poisonBackdoorData))
292         usedBdVolume = int(bdSize*1)
293         unusedBdVolume = int(bdSize - usedBdVolume)
294 
295         usedBackdoor, unusedBackdoor = random_split(poisonBackdoorData, [usedBdVolume, unusedBdVolume])
296         print('adversarial BACKDOOR data generated (bad label), len: ', len(usedBackdoor))
297 
298 
299         adversarialTestData, adTRightSquareCount, adTMiddleSquareCount = DataClass.generatePoisonAdversarialPy(adTest_set, 'clean-label', testName)
300         print('adversarial TEST data generated (clean label)')
301 
302         if trainName == 'perceptible-white-random':
303             if testName == 'perceptible-white-right':
304                 print(f'RANDOM RIGHT SQUARE Count: training data = {adRightSquareCount}')
305             elif testName == 'perceptible-white-middle':
306                 print(f'RANDOM MIDDLE SQUARE Count: training data = {adMiddleSquareCount}')
307             else:
308                 print('Full random test')
309 
310         avF1 = 0
311         avCpu = 0
312         avRam = 0
313         avTimeTaken = 0
314         repeatsCount = 1
315 
316         for i in range(experimentRepeats):
317 
318             print(f'starting run {repeatsCount} out of {experimentRepeats}')
319 
320             model = MNISTCNN()
321             optimizer = optim.Adam(model.parameters(), lr=0.001)
322             criterion = nn.CrossEntropyLoss()
323 
324             # hidden backdoor data has to be generated after model is created
325             if testName == 'hidden':
326                 poisonBackdoorData = DataClass.generateHiddenAdversarialPy(criterion, model, optimizer, backdoor_set)
327 
```

```
328             # generate poisoned experiment dataset and train model
329             backdooredExperimentDataset = DataClass(initialDataset.evalData, initialDataset.testData + adversarialTestData,
330                                                     initialDataset.trainData + usedBackdoor + usedAdTrain, adversarialPercentage)
331             train_loader = DataLoader(backdooredExperimentDataset.trainData, batch_size=64, shuffle=True, collate_fn=custom_collate)
332             cpu, ram, timeTaken = train(model, train_loader, optimizer, criterion, epochs)
333
334             # evaluate poisoned model on clean eval data, backdoor shouldn't impact clean data expect F1 ~0.98
335             test_eval_loader = DataLoader(initialDataset.evalData, batch_size=64, shuffle=True, collate_fn=custom_collate)
336             Eval_f1 = test(model, test_eval_loader)
337             clean_result = writeResult(testName, Eval_f1, adversarialPercentage, 'n/a', 'n/a', 'n/a', trainName)
338             writeCSV(clean_result, 'clean_test')
339             print('done clean test')
340
341             # test backdoored model
342             test_loader = DataLoader(backdooredExperimentDataset.testData, batch_size=64, shuffle=True, collate_fn=custom_collate)
343             f1 = test(model, test_loader)
344             indiviual_result = writeResult(testName, f1, adversarialPercentage, cpu, ram, timeTaken, trainName)
345             writeCSV(indiviual_result, 'backdoor_test_indiviual')
346
347             avF1 += f1
348             avCpu += cpu
349             avRam += ram
350             avTimeTaken += timeTaken
351
352         avF1 = avF1 / experimentRepeats
353         avCpu = avCpu / experimentRepeats
354         avRam = avRam / experimentRepeats
355         avTimeTaken = avTimeTaken / experimentRepeats
356
357         experiment_result = writeResult(testName, avF1, adversarialPercentage, avCpu, avRam, avTimeTaken, trainName)
358         writeCSV(experiment_result, fileName)
359         experimentCounts +=1
360
```