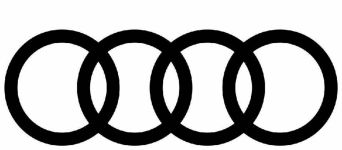


Improving neural network trojan detection via network abstraction

Master thesis
by
Marcello Eiermann

For the M. Sc. Artificial Intelligence
at
Utrecht University

List of examiners and supervisors:
Dr. Dominik Klein, Assistant Professor, First examiner
Prof. Dr. Albert Gatt, Professor, Second examiner
Dr.-Ing. Vahid Hashemi, Audi AG, Thesis supervisor
Akshay Dhonthi Ramesh Babu, Audi AG, Thesis supervisor

Improving neural network trojan detection via network abstraction

Marcello Eiermann

Abstract

Deep learning-based image recognition systems have become essential in a variety of applications, including autonomous driving functions in vehicles. The increased use of third-party datasets and pretrained models open up a new security risk, where any potential user cannot know if the data or model have been manipulated. Attackers can plant a backdoor during the training phase by poisoning a part of the dataset with a trojan trigger. The trojaned model behaves normally on benign inputs, but inputs that contain the trigger will cause the model to intentionally select a wrong output. In the domain of autonomous vehicles, an attack which causes an intentional misclassification of a road sign could have fatal consequences. One of the methods for detecting neural network trojans is Artificial Brain Stimulation (ABS) [20], which manually stimulates a neuron's activation value and observes the change in output activation values. We combine ABS with the neural network abstraction tool DeepAbstract [1], which computes clusters and cluster representatives, based on Input/Output similarity of neurons. Our strategy involves selectively applying the ABS analysis on the subset of cluster representatives, to possibly reduce the computational load and increase the detection accuracy. To assess the efficacy of our method, we conducted experiments using the GTSRB dataset, trojaning multiple models with six distinct triggers of varying visibility. We analyze two research questions: Whether our method can lead to a runtime improvement compared to ABS, and whether it can increase the detection accuracy. One model showed an improvement in stimulation runtime, while the runtime of the other models remained equal. Our method consistently yields superior or equivalent detection accuracy across all tested models compared to ABS. At best, our method increased the reverse-engineered attack success rate score by 33% and the number of detected trojaned neurons by 59%, demonstrating a clear improvement in detection accuracy.

Acknowledgements

I would like to express my gratitude to Akshay Dhonthi and Vahid Hashemi for the opportunity and their generous support throughout my time at Audi. Special thanks to Dominik Klein for his valuable and insightful feedback on my thesis. Lastly, my appreciation goes to Loïs Dona for diligently proofreading my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, deep learning-based image recognition systems have become essential in various applications, including Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD) systems in vehicles, where they play a crucial role in recognizing and interpreting road signs and signals due to their superior performance compared to classical methods, such as support vector machines or histograms of oriented gradients [7]. However, most state-of-the-art models require expensive hardware, large amounts of training data, and long training times. To reduce the associated costs of training, many users decide to utilize third-party datasets that are publicly available, rather than collecting data themselves. It has also become common practice to use pre-trained networks provided by third-party sources instead of training models from scratch. While these approaches are convenient, they come with a downside: users can lose control over the training process, which can make neural networks (NN) more vulnerable to security risks [12]. One particular type of risk that has received attention in the literature is the insertion of neural trojans into deep neural networks [37]. Through the contamination of training data, backdoors can be inserted during the training phase [12]. Other types of attacks do not need access to training data, but hijack inner neurons and perform limited retraining with manipulated inputs [4]. The backdoored models perform ordinarily on normal input, but when a benign input is combined with a trigger, such as a patch that is stamped on

a traffic sign, the model will intentionally misclassify the input to a specific output label, often called target label [20]. In the domain of vehicles with ADAS/AD capabilities, the intentional misclassification of a traffic sign carries profound implications, potentially leading to life-threatening situations for passengers and other road users. The significance of this research lies in its direct relevance to the safety and reliability of autonomous vehicles, aiming to mitigate the risks associated with neural network trojaning attacks.

Neural networks are, at their core, comprised of matrices that are interconnected with a particular structure. The weights assigned to these matrices encode the network's underlying meaning, which can be entirely implicit [22]. Neural networks generally do not provide a reasoning for their decision and the purely numerical parameters of a neural network cannot be read and interpreted [22]. This makes the detection of backdoors in neural networks particularly challenging. There are different approaches that aim to identify backdoors in neural networks. Artificial Brain Stimulation (ABS) [20] manually sets the activation values of a specific neuron, while freezing the activation values of the other neurons in the same layer and observes how the output values change. In theory, a trojaned neuron will behave differently to a benign neuron by having a much higher impact on a specific output target label, while possibly suppressing the activation values of other labels. However, this approach is computationally expensive. In order to increase the efficiency of ABS we propose to combine this approach with the neural network abstraction tool DeepAbstract [1]. DeepAbstract feeds input images into the NN and clusters neurons within the same layers based on similar input/output behaviour. K-means clustering is used to determine these clusters and the cluster representatives, which are the neurons closest to the center of their respective clusters. The method then removes all neurons that are not deemed cluster representatives and creates a new, smaller network. For our approach, we need to identify the cluster representatives and perform the ABS analysis on this subset of neurons. We want to analyze whether this method can reduce the computational load of ABS. This leads to our first research question:

Q1: Can activation-based abstraction effectively reduce the computational

load for analyzing neural networks using ABS?

We further want to analyze whether this approach can lead to an improvement in backdoor detection accuracy. Therefore our second research question is

Q2: Does the utilization of activation-based abstraction lead to more accurate results analyzing neural networks using ABS?

In order to implement our approach, we will need to adapt both DeepAbstract and ABS. DeepAbstract currently only works with fully connected layers and ABS will need to be utilized on only the subset of neurons that is determined by DeepAbstract.

The following chapters will present the necessary theoretical background, the proposed methodology, the experimental setup and the results. Finally, a discussion and conclusion will summarize the key findings and discuss limitations and future research possibilities.

Chapter 2

Theoretical background

This chapter will begin with a short explanation of fully connected neural networks and an overview of convolutional neural networks and the convolution operation, which will be essential in later chapters. Next, backdoor attacks and their different types will be explained as well as two methods for detecting trojaning attacks called Neural Cleanse and Artificial Brain Stimulation. The latter method will be the basis for this research, as we will try to enhance it with our clustering-based approach. Finally, this chapter introduces the neural network abstraction tool DeepAbstract.

2.1 Fully Connected Neural Networks

A fully connected neural network (FCNN) is a type of artificial neural network in machine learning. In this network architecture, neurons are organized into layers, with each neuron in one layer connected to every neuron in the next layer. Information flows through the network from the input layer, through a series of hidden layers, and finally to the output layer, which performs a task such as classification. The connections between neurons are called *weights* and determine the strength of the connection. The weights are learned during the training phase, where input data is fed through the network, comparing the predicted output to the actual output, and adjusting the weights to minimize the prediction error. Figure 2.1 shows an example architecture with two hidden layers. The activation

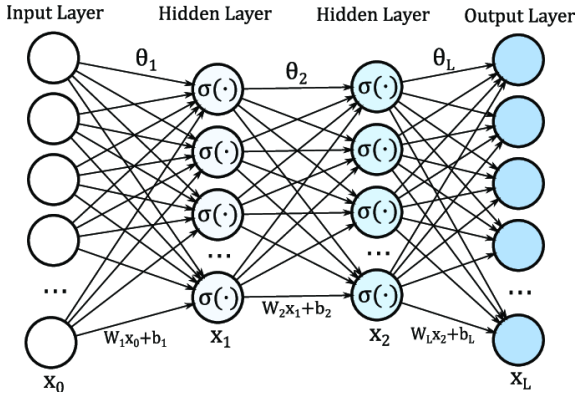


Figure 2.1: Structure of a FCNN with 2 hidden layers [34]

value of the first neuron in layer X_1 will be calculated by $\sigma(W_1x_0 + b_1)$ where $\sigma()$ is the activation function and W_l and b_l are the weights and biases, which are also referred to as parameters [34]. W_1x_0 is the multiplication of each weight with the value of each corresponding neuron in the input layer.

FCNNs are often used in machine learning, but for image-related tasks, convolutional neural networks are preferred due to their ability to capture spatial hierarchies and patterns in image data more effectively.

2.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of neural network that is primarily used for image classification, object detection, and segmentation. CNNs work by learning and recognizing patterns in images through the use of convolution and pooling operations [17]. It will take an image as input and read the numeric values of each pixel. A black and white image of size 32x32 pixels will have an input size of 32x32x1. A RGB color image of the same dimension, will have an input size of 32x32x3, where each of the 32x32 pixels has three channels for the three RGB colors.

Convolution is a mathematical operation that applies a filter, also referred to as a kernel, to an image to extract features. The filter slides across the image with a specified stride, performing element-wise multiplication and addition, thereby generating a new

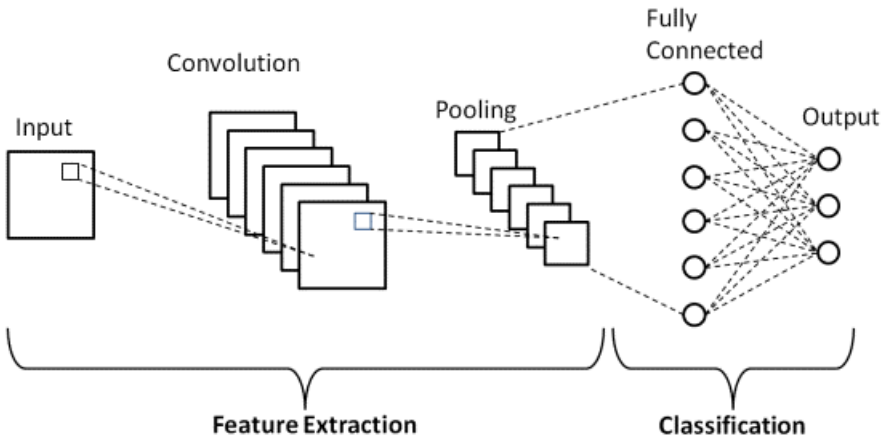


Figure 2.2: CNN example architecture [28]

output matrix. The resulting output is typically smaller than the input image, as the filter only covers a portion of the image at each step. The filter is learned through back-propagation, where the network learns to optimize the filter weights to extract meaningful features from the input image [11]. At the early layers, the convolution filters act as edge detectors, identifying low-level features such as borders and edges. As we move deeper into the network, the convolution filters react to more complex and abstract features. In the final layer, the network will produce an output that is used to make a prediction about the input data, such as the type of traffic sign.

The output size of the convolutional layer will depend on the size of the *input*, the size of the *filter*, the *stride* and the *padding*. The filter size determines the spatial extent of the filter that is applied to the input. A larger filter size will cover more pixels in the input, resulting in a larger receptive field and a higher degree of spatial abstraction in the output. The *stride* parameter determines the distance between two successive positions of the filter on the input. A larger stride will reduce the number of times the filter is applied to the input, resulting in a smaller output size. *Padding* refers to the addition of extra pixels around the border of the input, which allows the filter to be applied to the edge pixels of the input. Padding can be used to keep the output size the same as the input size after the convolution operation, which can be important for some applications. The following figure shows the convolution operation using a 2×2 kernel, a stride of (1,1) and

no padding [11]. We can see that the input of size 4×3 is turned into and output size 3×2.

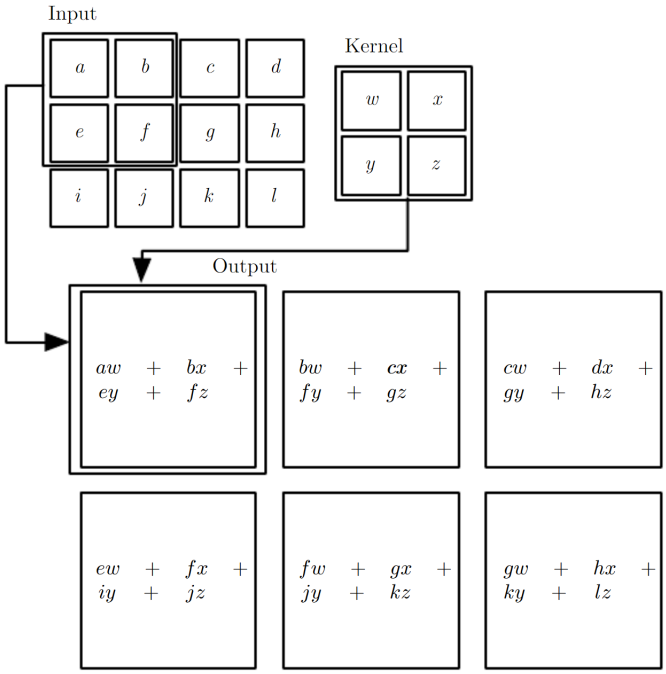


Figure 2.3: Convolution operation [11]

The 2×2 sized kernel is first applied to the top-left 2×2 sized window on the input. The resulting value will be the top-left value in the output. In the next step, the kernel will stride to the right by one position and be applied on the input values (b,c,e,f). This is repeated another time for the same row, after which the kernel moves down one row and is applied on all three positions in that row. The resulting output is of size 3×2.

Pooling is another operation that is commonly used in CNNs. Pooling works by reducing the size of the feature maps produced by the convolution operation. The most common type of pooling is max pooling, where the maximum value out of a set of pixel values in a sliding window is selected and the rest is discarded. This process helps to reduce the number of parameters in the network, which can help to prevent overfitting and improve generalization [11].

CNNs typically consist of multiple layers, including convolution layers, pooling layers, and fully connected layers. Convolution layers are responsible for extracting features from

the input image, pooling layers are used to reduce the size of the feature maps, and fully connected layers are used to perform the final classification or regression.

Having established the foundations of convolutional neural networks, our focus now shifts to the realm of adversarial attacks on neural networks.

2.3 Adversarial neural network attacks

The goal of an adversarial attack on a neural network is to cause an intentional misclassification, given an input that contains a perturbation or distortion [14]. Attacks can be categorized into two groups: targeted and untargeted attacks [14]. In a targeted attack, the attacker can select the label resulting misclassification label, which we will refer to as *target label*. The network will always predict the target label when it is attacked. An untargeted attack will cause misclassification, but the attacker cannot choose a specific target label. Furthermore, the attacks can be categorized based on the information the attacker can access:

- White-box attacks require the attacker to have full access to the inner structure and parameters of the neural networks, as well as the training dataset. The attacker can edit any values of the network or retrain it on poisoned data.

- Black-box attacks function without access to the inner structure. The attacker can only interact with the trained neural network by querying it with perturbed inputs.

The methods that will be discussed in this section are targeted white-box attacks. They involve inserting a hidden backdoor, also referred to as a trojan, into the network's architecture. The backdoor is typically designed to activate in response to a specific trigger, such as a particular input signal, and can cause the network to intentionally produce incorrect outputs. The attacker can design the trigger in a way that is difficult for others to detect, such as by using a specific pattern of input that is not common in normal usage scenarios. Once the trigger is activated, the network can produce outputs that are designed to harm or deceive the user, such as misclassifying images, generating

fake text, or executing malicious actions.

One of the most common types of attack is data poisoning [19]. Data poisoning attacks involve injecting malicious data into the training dataset to alter the behaviour of the neural network. This can be done by an attacker who has access to the dataset, by manipulating the training algorithm, or by modifying the data during the transmission process.

Figure 2.4 depicts a poisoning-based backdoor attack where a black square in the bottom right corner serves as the trigger, and the target label is designated as '0'. To carry out the attack, a part of the benign training images is modified by embedding the trigger, and their original label is substituted with the attacker's desired target label. Consequently, the deep neural network trained with this dataset is compromised. It will learn to associate the presence of the trigger with the target label and will misclassify any inputs stamped with the trigger as that target label '0' while providing accurate results on benign images.

These attacks are rather effective. Chen et al. [4] managed to successfully insert a backdoor using only 50 poisoned input samples. Other approaches, such as the approach from Liu et al. [22] does not need access to the training data but functions by inverting the neural network to generate a trojan trigger and then partially retraining the model with the reverse engineered training data that is stamped with the trojan trigger. The attacks can use different types of triggers:

Patch-based triggers

The most common and basic type of trigger is a patch-based trigger. Figures 2.4 and 2.5 are examples of a patch-based trigger, where a small static patch is stamped onto a part of the image. The patch can be a simple square, a pattern of pixels, or more complex forms like logos or small images [12]. Figure 2.5 depicts a stop sign that was stamped with a yellow square trigger and is misclassified by a trojaned neural network as a speed limit sign.

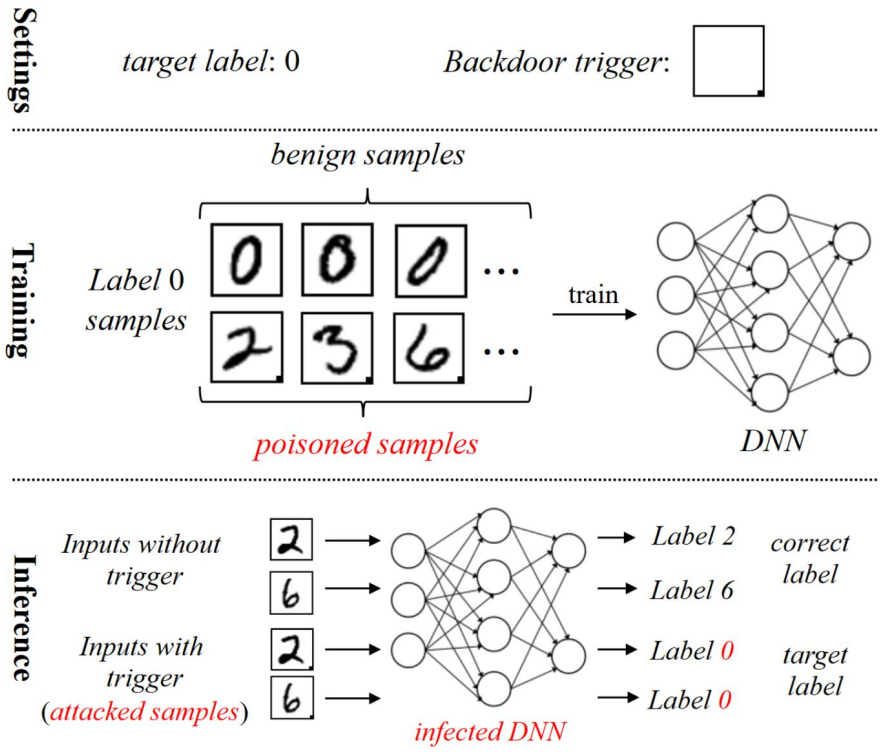


Figure 2.4: Poisoning-based backdoor attack [19]

Feature space trigger / perturbation-based trigger

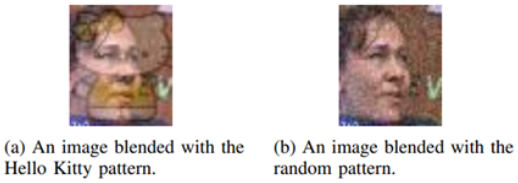
In contrast to patch-based triggers, feature space or perturbation-based triggers perturb the entire input image. They can also take on various forms such as an Instagram filter that changes the colours of the image [20], slightly transparent images or random patterns that are blended with the image [4].

Stealthy triggers

While the aforementioned triggers can be spotted by manually inspecting the data, there are stealthy triggers that make it hard to see if an image contains a trigger, even for human observers. Barni et al. [2] superimpose a sinusoidal backdoor signal over the image, which is difficult to detect and can be seen in Figure 2.7a. Liu et al. [23] utilize reflections, in order to create triggers that are difficult to spot by the human eye (Figure 2.7b).

Figure 2.5: Stop sign with patch trigger [12]

(a) Feature space attack using Instagram filters [20]



(b) Blended attack [4]

Figure 2.6: Feature space triggers

Imperceptible triggers

The newest trojaning methods are imperceptible, even on closer inspection. WaNet [26] uses a small and smooth warping field to generate backdoor images, which are not distinguishable from benign inputs based on human observation (Figure 2.8).

This project will focus on patch-based triggers, as they are physically realizable and therefore pose a realistic threat to computer vision systems for autonomous driving.

2.4 Defense techniques against trojan attacks

Several techniques were proposed to defend against trojan attacks including detection and bypassing of the trojan as well as removal of the backdoor from the model [37]. Wang et

(a) Sinusoidal backdoor attack [2]

(b) Stop sign with patch trigger [4]

Figure 2.7: Stealthy triggers

Figure 2.8: Warping based trigger [26]

al. [37] grouped the defense methods into the following categories:

- *Model Verification*: This mechanism verifies the efficacy of the model. If it detects anomalies in the functionalities of the model, a flag is raised for a potential trojan.
- *Trojan trigger detection*: The aim of trojan trigger detection is to detect the presence of triggers in the input, such as by fine-tuning a classifier to detect trojan triggers as anomalies in the input image [6].
- *Restoring Compromised Models*: Compromised models can be restored to a benign state by retraining and pruning the model, which can be done on a small subset of training data [24].
- *Trigger-based trojan reversing*: This method tries to estimate the potential trojan trigger pattern and uses it for retraining the model in order to increase its robustness [36].
- *Bypassing neural trojan*: This method utilizes pre-processing on the input before it is passed to the model by removing the trigger and passing a benign input [9].
- *Input filtering*: The strategy for input filtering involves filtering the input data by removing malicious input and therefore ensuring that the data passed into the model is mostly clean.

- *Certified backdoor defenses*: In contrast to the aforementioned ad-hoc techniques, certified backdoor defenses improve the robustness against adaptive attacks by e.g. adding random noise to the training data of a model [35].

The methods presented in the following subchapters fall under the categories of trojan trigger detection and trigger-based trojan reversing.

2.4.1 Neural cleanse

Wang et al. [36] created the first trigger-based trojan reversing defense, which they named Neural Cleanse (NC), wherein potential trigger patterns are obtained for each class and the final synthetic trigger and its corresponding target label are determined using anomaly detection [19]. NC assumes, that the defender has access to the trained neural network and a set of correctly labelled samples in order to test the performance on the model [36]. The goals of NC are:

1. Detecting backdoor: Making a binary decision on whether a neural network contains a backdoor and finding out which label is infected
2. Identifying backdoor: Reverse engineering the trigger that was used by the attack
3. Mitigating backdoor: The backdoor should be rendered ineffective by two approaches
   (a) Building a proactive filter that detects and blocks incoming adversarial inputs
   (b) Removing the backdoor from the NN without affecting the classification performance on benign input

The intuition behind the technique is derived from the basic properties of a backdoor trigger. A backdoor trigger produces a classification result to a target label regardless of the label the input typically would belong in.

The authors assume that the target label can be misclassified with much smaller modifications compared to other uninfected labels. As a result, they iterate through the model's labels and determine if any of them can be misclassified with significantly less modification compared to the rest. Their system consists of three steps:

1. Each label is treated as potential target label. An algorithm is designed to find the "minimal" trigger required to misclassify all samples from other labels into this target label. For images, this trigger defines the smallest collection of pixels and its associated colours that cause misclassification.

2. Step 1 is repeated for all output labels in the model.

3. The size of each trigger is measured by the number of pixels. An outlier detection algorithm is used to detect if a trigger candidate is substantially smaller than other candidates. A significant outlier is assumed to be a real trigger and its corresponding label is seen as target label.

The trigger produced by step one is considered as *reverse engineered trigger*, which aids the understanding of why the model misclassifies samples. This can be used to create a proactive filter that detects and filters all adversarial inputs.

There are several scenarios in which NC is unable to detect neural trojans. NC is not effective for feature space attacks and it is not as effective when only one image is provided for each label [20]. NC is much more effective when the full training set is used and less effective when the size of the trojan trigger is larger than 6% [20]. Figure 2.9 shows the detection accuracy increasing based on the number of data points and a sharp decline in accuracy if a larger trigger is used.

Figure 2.9: Detection accuracy of Neural Cleanse based on the number of data points used and the size of the trojan trigger [20]

2.4.2 Artificial Brain Stimulation

Artificial Brain Stimulation (ABS) [20] is a technique that was inspired by Electrical Brain stimulation, which is used to study the behaviour of human and animal brain neurons by applying an electrical current of differing strength levels to selected neurons and observing the consequences. Analogously, ABS scans AI models by manually modifying a neuron's activation values while freezing the activation values of the other neurons in the same layer, then monitoring the changes in output values. This is performed on all convolution and fully connected layers. They assume the presence of *trojaned neurons*, which strongly respond to the presence of a trojan trigger. A trojaned neuron is assumed to perform differently from a benign neuron by having a considerably greater impact on certain output target labels. If an input image contains a small trigger patch, the neurons associated with this small area need to have a much larger impact in order to change the output label from the correct class to the target class. The authors define a model as successfully trojaned if:

1. The trojaned model does not have (non-trivial) accuracy degradation on benign inputs

2. For any benign input, if it is stamped with the trojan trigger, the model has a high probability of classifying it to the target label regardless of its original label

Furthermore, they make a simplifying assumption that for each target label only one trigger exists, and that this trigger is intended to transform any input, regardless of its label, into the target label. Therefore, advanced attacks, which only attack certain images were out of scope, as well as attacks with multiple trojaned neurons.

The concept of ABS can be described as follows: A benign input is fed into the model and all inner neuron activation values are collected. In layer L_i one neuron α is analyzed by freezing the activations of the other neurons in that layer and examining how the output activation Z_t in the output layer for a label t changes. If α is a potentially compromised neuron and t is the corresponding target label, Z_t will be much higher than the activation

of other target labels, whenever α falls into a certain range. This phenomenon can be seen in Figure 2.10. The X- and Y-axis denote the activation values V_α and V_β for two neurons α and β, the Z-axis is the output activation value Z_t for the target label. Comparing the benign model (a) to the trojaned model (b) a ridge can be observed, which cuts through the entire space at $V_\alpha = 70$. This means that whenever α falls within this range, no matter which value β takes, the output activation of label t will be greatly enlarged. The same analysis is performed on different input labels and if neuron α consistently shows this behaviour it is considered a compromised neuron candidate.

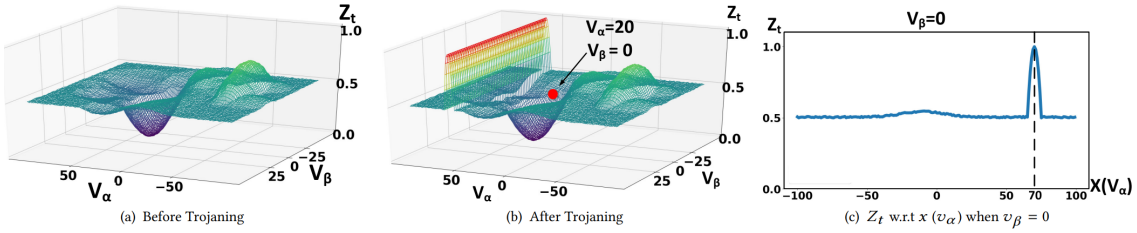
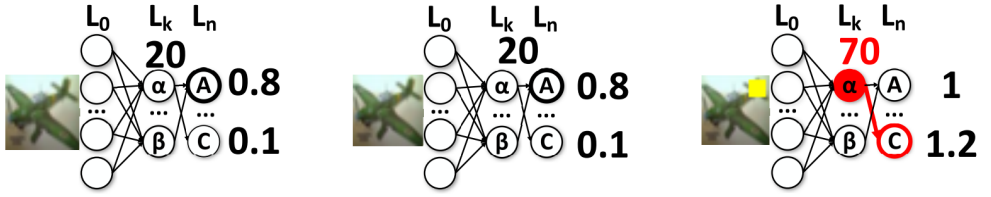


Figure 2.10: Activation values of benign and trojaned neurons [20]

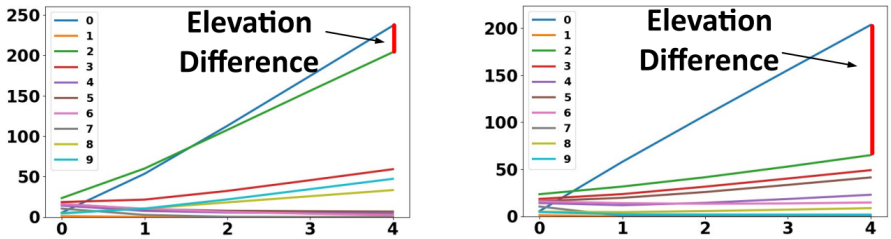
However, there can be several candidate neurons that substantially elevate the output activation of a specific label with only a small subset being compromised neurons. ABS tries to eliminate such false positives by generating an input pattern, that activates a candidate neuron and achieves the previously identified activation value range, which strongly increases the target label activation. A compromised neuron will have less confounding with other neurons, meaning that its value will change independently from other activation values. A benign neuron should have significant confounding, making it impossible to generate an input pattern that achieves the activation value range. The following figure shows the output activation values for class A (plane) and C (car). The benign model as well as the trojaned model will correctly classify the benign input image as A with an output activation of $Z_A = 0.8$, while the target label has an activation value of $Z_C = 0.1$. When the input is stamped with a trigger, the compromised neuron α falls into the range of around 70 and the activation value of the target label is strongly increased to $Z_C = 1.2$, surpassing the true label at $Z_A = 1$.



(a) Benign model with benign image (b) Trojaned model with benign image (c) Trojaned model with trigger image

Figure 2.11: ABS output activation values [20]

ABS employs two criteria to enhance the precision of its identification of trojaned neuron candidates. First, instead of analyzing the maximum elevation, it takes the elevation difference between the highest and second highest elevation into account. The reason for this is that compromised neurons will only elevate the target label activation value, while benign neurons are more confounded and will increase multiple activation values. Figure 2.12(a) shows the output values (y-axis) of the 10 classes, given the activation values (x-axis) of a neuron that is being analyzed. The left neuron might be considered benign, since it elevates two labels, while the right neuron can be considered a compromised neuron candidate, having a large difference between the top two labels. Second, ABS uses the minimum elevation across all images for a label instead of a single image because a benign neuron might only elevate a subset of images, while a trojaned neuron will elevate all images of the target label.



(a) Examples for Using Elevation Difference Instead Elevation

Figure 2.12: ABS elevation difference - possibly benign neuron (left) and possibly trojaned neuron (right)[20]

After gathering the compromised neuron candidates, ABS tries to separate the false positives by reverse engineering the trojan trigger using an optimization procedure. This

will not be further explained, since we do not intend to make any changes to this part of the system. The reverse engineered triggers are stamped onto benign input images and fed into the network. ABS then measures if this causes the network to misclassify the input to the target label.

Figure 2.13 depicts an original image, a trojaned image using a pixel trigger and feature trigger, as well as their respective reverse engineered triggers. If the reverse engineered trigger can successfully cause misclassification in at least 80% of the cases it is considered a real trojan and the model is considered compromised.

Figure 2.13: ABS reverse engineered triggers [20]

2.4.3 Other methods improving on ABS

This section briefly mentions two methods that augment ABS. They are not explained in detail as their approach differs from the current approach and will not be relevant to this work.

ABS+K-Arm Optimization

One method that tries to improve ABS is the K-arm optimization method [30], which is inspired by the Multi-Arm-Bandit in Reinforcement Learning. The authors state that ABS has a shortcoming of potentially selecting the wrong neurons in the stimulation analysis. In the first step ABS stimulates neuron activation values and observes if consistent misclassification can be achieved. The neuron candidates determined in the first step are used to create a reverse engineered trigger, which then is used to determine whether the neurons are truly compromised. However, it is possible that the first step produces candidates that do not contain the compromised neurons and therefore the trigger gen-

eration fails to derive the correct triggers. K-arm tries to improve on that by iteratively and stochastically selecting the most promising labels for optimization. The authors state that they achieve higher performance compared to ABS in terms of accuracy and runtime.

ABS+ExRay

The authors of ABS+ExRay [21] claim to improve on the drawback that ABS will only work on small and fixed triggers while being less effective on larger and more stealthy triggers. Their detection method uses a trigger inversion technique to create a trigger that will misclassify examples to a target label. Then, it checks whether a trigger is made of features that are not natural distinctive features between the victim and target classes using Symmetric Feature Differencing (SFD). SFD determines the distinguishing features between two sets of classes. The authors give the example of two persons A and B in a face recognition model, where replacing Person A's nose with person B's causes the model to predict B and vice versa, making the nose a distinctive feature. They state that their method cuts down on the number of false positives by 78-100% with an increase of false negatives of 0-30%, leading to an overall accuracy increase of 17-41%.

2.5 Neural network pruning

This research aims to combine ABS with the neural network abstraction method DeepAbstract. Different types of pruning will be listed in this section. Pruning is an effective method for reducing the number of parameters in a Deep Neural Network (DNN). Many parameters in a DNN are redundant and do not contribute significantly to lowering the error or improving the network's generalization during training. Therefore, after training, these parameters can be pruned from the network with little impact on the network's accuracy [5]. Eliminating less effective connections can significantly reduce the storage, computation cost, energy and inference time of a DNN model [5]. CNNs can also benefit from pruning the filters of the convolution layers, therefore reducing the computation and speeding up the inference process [18]. There are several pruning methods:

Weight pruning The weights between neurons can be pruned based on certain conditions. Han et al. [13] set the weights to zero if they are unimportant, which is determined if they are below a certain threshold or if they are redundant.

Neuron pruning Rather than removing individual weights, another approach is to eliminate redundant individual neurons, as suggested by Srinivas and Babu [32]. This approach involves removing all the incoming and outgoing connections associated with the redundant neuron. It is assumed that the information captured by a redundant neuron is already present in other neurons. There are various other methods to remove individual weight connections or neurons [5].

Filter pruning In a convolution layer of a CNN, entire filters can be pruned. Li et al. [18] calculate the importance of filters by their L1/L2 norm and remove whole filters in the network together with their connecting feature maps, which have the least effect on output accuracy.

Layer pruning Another pruning method is layer pruning, such as the method by Chen and Zhao [3], where the performance of feature representations is extracted at different convolution layers within the architecture and layers with comparatively small contributions are determined and removed from the network.

The pruning method which is presented in the following subchapter performs neuron pruning by taking the activation values into account and using K-Means clustering to determine similar neurons.

2.6 DeepAbstract

DeepAbstract [1] is a neural network abstraction framework. Abstraction is a technique used in formal methods to simplify a system by disregarding irrelevant details and constructing a smaller system with similar behaviour. In contrast to other abstraction techniques, that analyze the weight connections between neurons [45], DeepAbstract provides

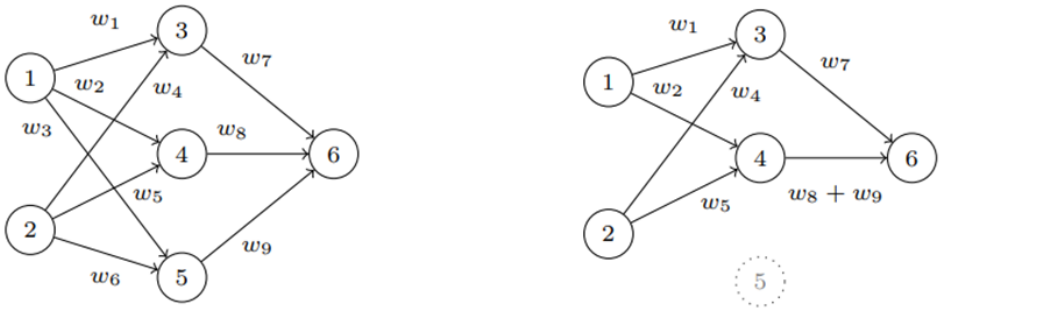
a notion of similarity by analyzing activation values, which the authors claim is more general and more powerful [1].

DeepAbstract considers simple fully connected neural networks with one input layer, one output layer and multiple hidden layers. In classic abstraction, states with similar properties are merged for analysis. For NNs, identifying which neurons to merge and defining similarity is not immediately clear. Neurons do not have an inner structure like states with variable values, which makes it more challenging to detect and drop irrelevant information. To overcome this, the authors suggest merging neurons that compute similar functions on a set X of inputs, where they produce ϵ-close values for each input $x \in X$. This is referred to as Input/Output-similarity. To simplify the analysis and implementation, only neurons from the same layer are merged.

The following example in figure 2.14 demonstrates the procedure. The network consists of two input neurons, one hidden layer with three neurons and an output layer with one neuron. The weights are denoted by w and the activation values are denoted by z. The activation values of the hidden layer are calculated as follows:

$$\begin{aligned} z_3 &= ReLU(w_1 * z_1 + w_4 * z_2), \quad z_5 = ReLU(w_3 * z_1 + w_6 * z_2), \\ z_4 &= ReLU(w_2 * z_1 + w_5 * z_2), \quad z_6 = ReLU(w_7 * z_3 + w_8 * z_4 + w_9 * z_5). \end{aligned} \tag{2.1}$$

With $ReLU(X)$ being the Rectified Linear Unit function defined as $max(0, x)$



(a) Original network

(b) Network after merging neurons 4 and 5

Figure 2.14: Clustering of neurons [1]

If the activation values of neurons 4 and 5 are similar for all inputs (denoted by $z_4 \approx z_5$), then the network is abstracted by merging those two neurons. In this example, neuron 4 is chosen as the cluster representative and the outgoing weight of the representative is set as the sum of all outgoing weights in that cluster. In the abstracted network, the activation values of neuron 3 and 4 remain the same:

$$\tilde{z}_3 = ReLU(w_1 * \tilde{z}_1 + w_4 * \tilde{z}_2) = z_3 \text{ and } \tilde{z}_4 = ReLU(w_2 * \tilde{z}_1 + w_5 * \tilde{z}_2) = z_4$$

The value of neuron 6 is now calculated by:

$$\tilde{z}_6 = ReLU(w_7 * \tilde{z}_3 + (w_8 + w_9) * \tilde{z}_4) = ReLU(w_7 * z_3 + w_8 * z_4 + w_9 * z_4) \approx z_6$$

The effectiveness of the abstraction relies on the selection of neurons to be merged. There are various techniques to identify these groups, and the authors selected the unsupervised learning method of K-means clustering as proof of concept.

Algorithm 1 Abstract network D with given clustering K_L

procedure ABSTRACT(D, X, K_L)
  $\tilde{D} \leftarrow D$
  for $\ell \leftarrow 2, \ldots, L-1$ **do**
    $A \leftarrow \{\mathbf{a}_i^{(\ell)} \mid \mathbf{a}_i^{(\ell)} = [\tilde{z}_i^{(\ell)}(x_1), \ldots, \tilde{z}_i^{(\ell)}(x_N)] \text{ where } x_i \in X\}$
    $\mathcal{C} \leftarrow$ KMEANS($A, K_L(\ell)$)
    for $C \in \mathcal{C}$ **do**
      $\tilde{W}^{(\ell)}_{*,rep(C)} \leftarrow \sum_{i \in C} W^{(\ell)}_{*,i}$
    delete $C \setminus \{rep(C)\}$ from $\tilde{D}$
  return $\tilde{D}$

Figure 2.15: DeepAbstract algorithm 1: Clustering [1]

Algorithm 1 (Fig. 2.15) describes the general approach. The function takes three inputs: the original trained network D, an input set X, and a function K_L that specifies the number of clusters to be identified for each layer. Each input $x \in X$ is fed into the network $\tilde{D}$, is evaluated and a $|X|$-dimensional vector of observed activations $a_i^{(l)}$ is generated for each neuron i in layer l . These activation vectors, representing each neuron, are then aggregated into the set A. Afterwards, K-means clustering is used to identify $K_L(l)$ clusters. The neuron closest to the centroid of the respective cluster is chosen as

Algorithm 2 Algorithm to identify the clusters

```
1: procedure IDENTIFY-CLUSTERS(D, X, α)
2:     D̃ ← D
3:     for ℓ ← 2, ..., L − 1 do                          ▷ Loops through the layers
4:         if accuracy(D̃) > α then
5:             K_L(ℓ) ← BINARYSEARCH(D̃, α, ℓ)          ▷ Finds optimal number of clusters
6:             D̃ ← ABSTRACT(D̃, X, K_L)
7:     return K_L
```

Figure 2.16: DeepAbstract algorithm 2: Identifying the number of clusters [1]

cluster representative, denoted by $rep(C)$, and merged with the other neurons by summing the outgoing connections.

Although Algorithm 1 outlines the clustering procedure, selecting the appropriate number of clusters per layer (K_L) remains a challenge. Algorithm 2 (Fig. 2.16 presents a heuristic for determining a suitable parameter set for clustering. This heuristic is based on the observation that merging neurons closer to the output layer has the least impact on network accuracy because merging errors are not amplified and propagated through multiple layers. The main idea is to search for the best K-means parameter ($K_L(l)$) for each layer l from the first hidden layer to the last hidden layer, ensuring that merging with the specified parameter (K_L) does not cause the network's accuracy to fall below a threshold α. The function takes three inputs: A trained network D, an input set X, and a parameter α that sets the minimum accuracy threshold for the abstract network. Starting with the first hidden layer ($l = 2$), the algorithm attempts K-means clustering. The parameter $K_L(l)$ is determined using the *BinarySearch* procedure, which searches for the lowest value of that yields the highest accuracy for the abstracted network. According to the authors, a higher degree of clustering (i.e., a smaller K) results in a greater reduction in accuracy.

Chapter 3

Methodology

This chapter will first introduce the proposed workflow for combining DeepAbstract with ABS and show a preliminary analysis that supports our hypothesis. Since DeepAbstract is only implemented to function with fully connected layers, we propose two methods to adapt its functionality for convolution layers. Finally, we will explain the adaptations made to ABS, which were necessary to combine it with the cluster representatives determined by DeepAbstract.

3.1 Method overview

The main concept is to enhance the performance of ABS by combining it with the clustering of DeepAbstract. We want to utilize DeepAbstract to analyze a trained neural network and determine the cluster representatives. This information should be used within ABS in order to focus the analysis on this subgroup of neurons. We want to be able to analyze CNNs, but DeepAbstract does not support convolution layers. For this reason we decided to convert the convolution layers to fully connected layers for our first approach, which will be explained in detail in section 3.3. The flowchart in Figure 3.1 shows an overview of the initial approach. A CNN will be trained on the GTSRB dataset and converted to a fully connected neural network, by converting all convolution layers to fully connected layers. The new DNN will be fed into DeepAbstract, which computes the clusters and the

set of cluster representatives (CR). Additionally, the abstract FCNN is computed, which will be discarded for our current analysis [1]. In the next step of the flowchart, the original CNN is analyzed by ABS. The cluster representatives of the fully connected layer can be directly inferred back to positions in the original convolution layer. This is explained in section 3.5. The idea is, that the cluster representatives will have similar Input/Output

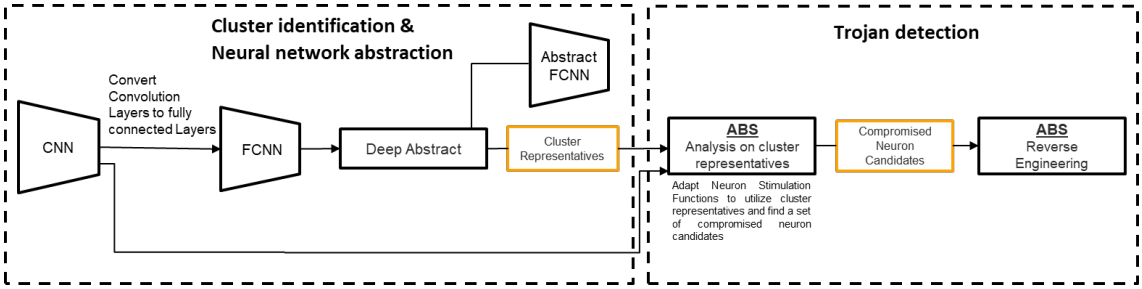


Figure 3.1: Flowchart DeepAbstract + ABS approach with layer conversion

behaviour as the neurons in their respective cluster and therefore performing the ABS analysis on the smaller subset of cluster representatives should produce similarly accurate results, while reducing the number of neurons that need to be analyzed and consequently the computational load. The set of CRs is used in ABS to compute the compromised neuron candidates, which ABS then uses to reverse engineer the trojan trigger. This reverse engineered trigger is used to test whether it can successfully subvert the models predictions to the target label.

The first approach has the limitation, that a CNN will drastically increase in size when being converted to a FCNN. The reason for this phenomenon is explained in section 3.3. For larger models, this leads to unreasonably high computation cost and network size. To combat this limitation, a new approach (Figure 3.2) was created, which does not convert the model itself, but the feature output of each convolution layer. The flattened output is fed into DeepAbstract in order to compute the CRs. This new method improves on computation time and facilitates the usage of large CNNs, but it does not allow the creation of an abstract FCNN. The details will be explained in section 3.4.

[1]The abstract FCNN is only necessary when utilizing DeepAbstract's full algorithm, including finding the optimal K value. This is not done due to technical limitations, which are explained in section 3.3

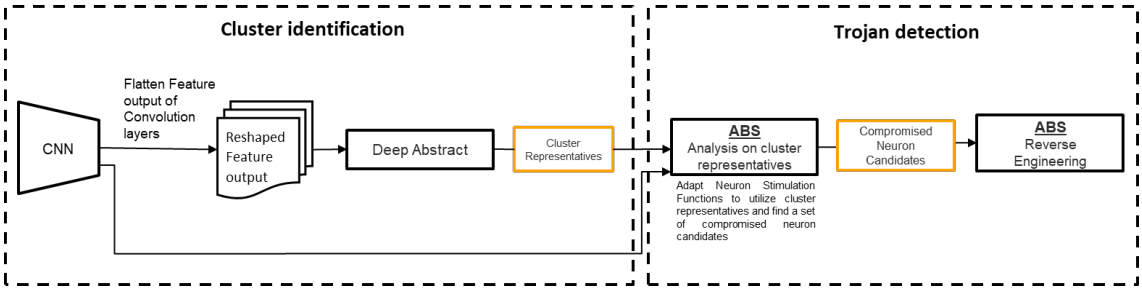


Figure 3.2: Flowchart DeepAbstract + ABS approach without layer conversion

3.2 Preliminary neuron activation analysis

One premise of our approach is that the clusters which are determined by DeepAbstract are still as meaningful when inputting clean images as they are when inputting trojaned images. One could argue that a group of neurons that is clustered by Input/Output similarity on clean images does not necessarily exhibit the same behaviour on images that contain a trigger. We want to show that activation values which correlate on clean inputs also correlate on trojaned inputs. To test this premise, we trained a FCNN on trojaned data. The FCNN was then fed trojaned images and the three neurons with the highest activation values were selected for further analysis.

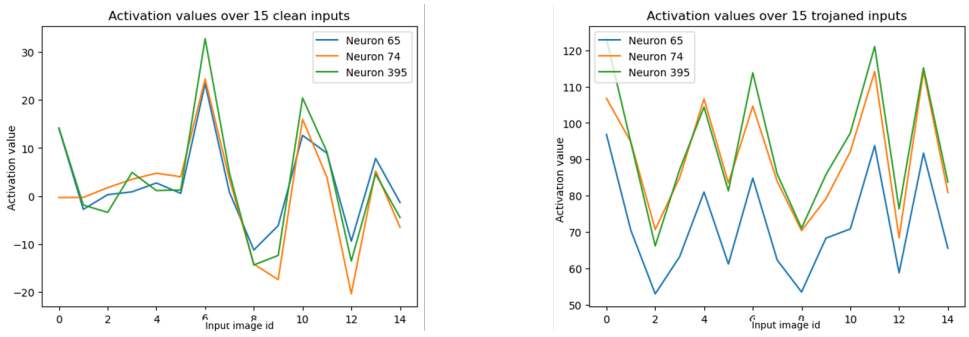


Figure 3.3: Activation values of three selected neurons. Left: Clean inputs; Right: Trojaned inputs

Figure 3.3 shows the activation values over 15 images without (left) and with (right) the presence of a trigger. We can see that in both cases the neuron values are highly correlated and that the trojaned images produce much higher activation values. To make sure this behaviour is not present in all cases, we test three random neurons as well. Figure

3.4 shows that the random neurons do not exhibit the same correlation behaviour as the previous neurons.

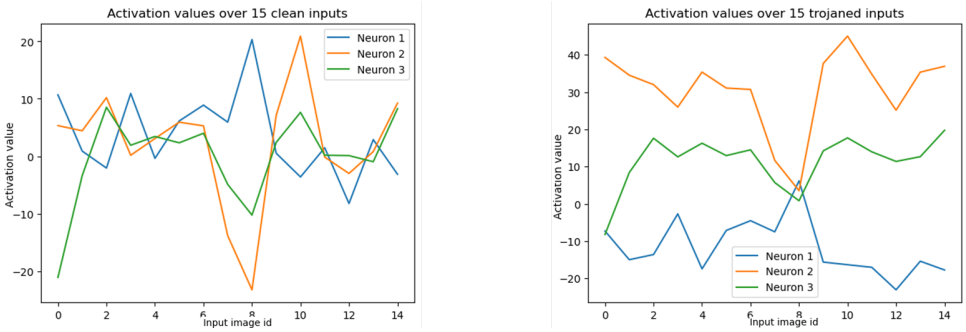


Figure 3.4: Activation values of three random neurons. Left: Clean inputs; Right: Trojaned inputs

It is important to note that while this experiment was repeated multiple times, no in-depth analysis was performed. These preliminary results serve as an initial indication rather than a definitive validation of our method. The experiments and results presented in chapters 4 and 5 will provide a definitive answer to the research questions.

3.3 Converting a CNN to a FCNN

DeepAbstract's implementation only works with fully connected neural networks. Given that CNNs are more prevalent in computer vision applications compared to FCNNs, and considering that existing literature on neural trojan detection predominantly focuses on CNNs, we have to be able to utilize CNNs in our testing scenarios. Fully connected layers are just special cases of convolution layers [46], therefore we decided for the initial approach to convert the CNN's to FCNN's by first converting convolution layers to fully connected layers. Afterwards, the remaining network needs to be adapted to handle the Inputs and Outputs of the FC layer. Figure 3.5 shows an example of a convolution layer that is converted to a fully connected layer and how the mathematical operations would match. This example shows a $3 \times 3 \times 1$ input with a 2×2 kernel with stride $= 1$ and padding $= 0$. The resulting feature output is therefore of size 2×2. In the first step of the operation, the kernel is applied to the upper left corner of the input, creating neuron 1 as

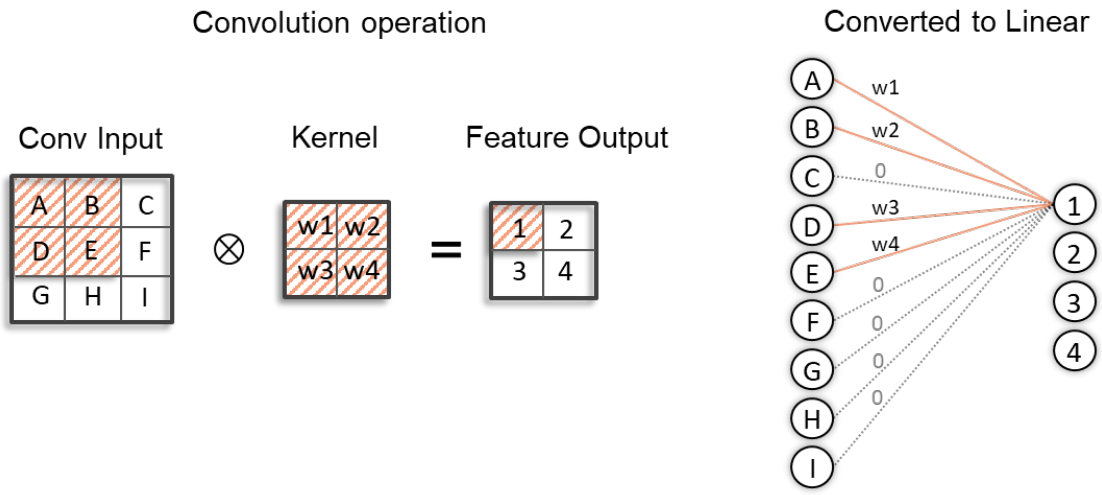


Figure 3.5: Converting the convolution operation to a fully connected/linear operation

feature output. This is highlighted in figure 3.5. The value of neuron 1 is calculated by $A*w1+B*w2+D*w3+E*w4$. We can convert this to a linear operation by flattening the input to a single dimension and setting the weights of the connections to neuron 1 to be the same weights as the kernel, at the positions where the kernel is applied (A,B,D,E) and 0 at all remaining connections (C,F,G,H,I).

Similarly, the values of neurons 2, 3 and 4 would be calculated by multiplying the input values (B,C,E,F), (D,E,G,H) and (E,F,G,I) with the kernel weights ($w1-w4$) respectively. The conversion to linear would also set the weights to $w1-w4$ where the kernel is applied and 0 elsewhere. The following matrix is the full weight matrix for the linear layer in our example (adapted from [10]):

$$\begin{pmatrix} w1 & w2 & 0 & w3 & w4 & 0 & 0 & 0 & 0 \\ 0 & w1 & w2 & 0 & w3 & w4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w1 & w2 & 0 & w3 & w4 & 0 \\ 0 & 0 & 0 & 0 & w1 & w2 & 0 & w3 & w4 \end{pmatrix} \tag{3.1}$$

This conversion still works if the padding and stride parameters are different. The output size will be dependent on padding (P), kernel size ($k_h * k_w$), input size ($H_{in} * W_{in}$)

and stride (s) and can be calculated with the following formulas [25]:

$$H_{out} = \frac{H_{in} + 2P - k_h}{s} + 1 \tag{3.2}$$

$$W_{out} = \frac{W_{in} + 2P - k_w}{s} + 1 \tag{3.3}$$

More details of the equivalence of this conversion can be read in the work of Ma & Lu [25]. After implementing the conversion, we conducted extensive testing to ensure its functionality. Input images were inserted into the original CNN and the converted FCNN and we could observe that the output layer activation values were equal. Therefore the equivalence of these layers can be confirmed.

The second step is to integrate the converted layers into the neural network. The convolution layers cannot simply be replaced by the converted linear layers, as this is incompatible with the remaining architecture. Other layers in the network, such as MaxPooling or Flatten layers will be dependent on receiving multidimensional inputs. A MaxPooling layer will reduce the layer dimension and will iterate over all filters. For example, a MaxPooling layer with the standard parameters (stride=2, padding=0) will reduce a $28 \times 28 \times 6$ (height $\times$ width $\times$ no. of channels) input to $14 \times 14 \times 6$. Therefore, it expects a 3-dimensional input. A linear layer however would provide a flattened, 1-dimensional output. The solution is to unflatten the layer by reshaping it to the output size of the corresponding output layer. Table 3.1 lists the architecture of the LeNet [16] CNN and the architecture of its converted FCNN.

Layer in CNN	Input size	Output size	Layer in FCNN	Input size	Output size
Conv2D[2]	(32,32,3)	(28,28,6)	Flatten	(32,32,3)	3072
			Linear (Converted)	3072	4704
			Unflatten	4704	(28,28,6)
ReLU	(28,28,6)	(28,28,6)	ReLU	(28,28,6)	(28,28,6)
Maxpool[3]	(28,28,6)	(14,14,6)	Maxpool[2]	(28,28,6)	(14,14,6)
Conv2D[4]	(14,14,16)	(10,10,16)	Flatten	(14,14,6)	1176
			Linear (Converted)	1176	1600
			Unflatten	1600	(10,10,16)
ReLU	(10,10,16)	(10,10,16)	ReLU	(10,10,16)	(10,10,16)
Maxpool[2]	(10,10,16)	(5,5,16)	Maxpool[2]	(10,10,16)	(5,5,16)
Flatten	(5,5,16)	400	Flatten	(5,5,16)	400
Linear	400	120	Linear	400	120
Linear	120	84	Linear	120	84
Linear	84	43	Linear	84	43

Table 3.1: Architecture of the LeNet CNN and its corresponding converted FCNN

The input size of the first layer is $32 \times 32 \times 3$ because that is the size of the images in the GTSRB dataset (32×32 pixels and 3 channels for RGB). The output size of the last linear layer is 43, as this layer is the classification layer and the number of classes in the GTSRB dataset is 43. The output size of the first Conv2D layer can be calculated using the previously mentioned formula 3.2. $H_{out}/W_{out} = \frac{32+2*0-5}{1} + 1 = 28$. This returns an output size of $28 \times 28 \times 6$, given that the number of filters is set to 6. To be able to replace that Conv2D layer in the network, we need to replace it with something that takes the same 3-dimensional input and returns the same 3-dimensional output as the Conv2D layer. Therefore we replace it with a block consisting of Flatten + Linear + Unflatten. The flatten layer takes the $32 \times 32 \times 3$ input and flattens it to size 3072. The linear layer is composed of 4704 neurons, which corresponds to the flattened output size of the Conv2D layer ($28 \times 28 \times 6 = 4704$). The Unflatten layer reshapes the 1-dimensional output of 4704 back to a 3-dimensional output of $28 \times 28 \times 6$. In the LeNet architecture, the second Conv2D layer is replaced in a similar manner. There, the converted linear layer consists of 1600 neurons. The remaining layers of the LeNet network are unaffected by the conversion

[2]Kernel size: 5x5, Stride: 1, Padding: 0, No. of Filters: 6
[3]Kernel size: 2x2, Stride: 2, Padding: 0
[4]Kernel size: 5x5, Stride: 1, Padding: 0, No. of Filters: 16

and can be copied over to the FCNN.

The size of the converted linear layer is an indication of how much the layer conversion affects the layer size. The first converted linear layer will have 4704 neurons and an input of 3072 neurons, creating a weight matrix with 14.450.688 weights. For comparison: the original convolution layer had only 450 weights, calculated by

$$k_h \cdot k_w \cdot \textit{number of Filters current layer} \cdot \textit{number of Filters Previous Layer} = 5 \cdot 5 \cdot 6 \cdot 3 = 450 \tag{3.4}$$

The weight matrix (3.1) in our mock example was already quite sparse. Each row of 9 values (given the 3×3 input) contains only 4 nonzero weights (given the 2×2 kernel). The weight matrix of our LeNet example will be even more sparse, having only 75 nonzero weight values ($5 \times 5 \times 3 =$ kernel size $\times$ 3 input filters) per row consisting of 3072 ($32 \times 32 \times 3$ input size) values. The layer size directly affects the file size of the trained network. With our LeNet example, the original CNN takes up 1.5MB of disk space while the converted FCNN uses 61MB. AlexNet [15] has a size increase from 15MB to 185MB. Our own CNN, which consists of six convolution layers and no linear layer apart from the classification layer has an even larger increase in size from 1MB to 1250MB. If we want to train and convert larger models, such as VGG [31], which will already have a size of over 200MB as CNN, then the conversion to FCNN fails due to lack of GPU memory in our system. In order to overcome this size limitation, we have devised a new method, that avoids converting the entire CNN to a FCNN, which will be explained in the following chapter.

It should be noted that for normal classification tasks, there is almost no difference in runtime between the CNN and FCNN. This may seem unintuitive, given that the FCNN is several magnitudes larger than the CNN. However, both networks undergo similar mathematical operations when presented with an input image. Instead of having a sliding window on the input image for the kernel to iterate through, this operation is already integrated in the weight matrix of the linear layer.

3.4 Reshaping feature output

The second method, as shown in Figure 3.2, gets rid of the conversion to fully connected. As explained in chapter 2.6, DeepAbstract computes the activation values of all neurons in a layer and clusters the neurons based on the similarity of those activation values. For the K-means clustering, DeepAbstract expects the activation values in a format, which is given as output by fully connected layers. Since we want to utilize convolution layers, we need to change the layer outputs. We modify this process by taking the original convolution layer to compute the activation values and then flatten those activation values to resemble the activation values of the fully connected layer. Going back to the LeNet architecture (see table 3.1), the first convolution layer will compute activation values with the output size (28,28,6). This will be flattened to size 4704. DeepAbstract will utilize these activation values to compute the clusters and cluster representatives. This method has several advantages and disadvantages compared to the CNN to FCNN conversion.

Most importantly, the computation time and size are reduced and it allows us to analyze larger CNNs such as VGG, which previously would be limited by the GPU memory capacity and could not be converted. The downsides are, that DeepAbstract will not create an abstract FCNN and cannot use binary search to determine the best K Parameter for the K-means algorithm. DeepAbstract's original implementation would test different parameters for the K value in K-means using binary search and settle on the lowest K, therefore the highest degree of network pruning, which does not impact the model accuracy. For this step, the models must undergo pruning, specifically the removal of neurons in clusters that were not determined as cluster representatives, resulting in a more compact, pruned network. The pruning is only possible if the clustering is computed on fully connected layers. The neurons which are selected to be removed, will correspond to neurons in the fully connected layer. However, in convolution layers, they correspond to positions in the filters, and not to the entire filter. We cannot prune parts of a filter, which is why pruning on convolution layers is not possible with DeepAbstract. Therefore, we cannot run binary search to determine the optimal K parameter and instead have to

set it to a fixed value.

3.5 Adaptations to ABS

In the previous step we have computed the set of cluster representatives based on either the converted FCNN or the reshaped activation values. We now need to modify ABS to perform the stimulation analysis only on the subset of cluster representatives. Upon analyzing the ABS code[5], we noticed that the stimulation of neurons is done slightly differently to what is reported in the ABS paper [20]. When ABS analyzes a convolution layer, it iterates through every output feature channel in that layer, stimulating it with a range of values that is determined by the maximum activation value in the respective layer. The following steps broadly describe the algorithm.

A. For each convolution layer in the neural network:

   1. Input 43 images (1 of each class) to calculate the maximum activation value
   2. Define a set of stimulation values **vs**, based on the maximum value
   3. For each output feature channel, it sets the activation value of the entire feature channel to a value $\mathbf{v} \in \mathbf{vs}$
   4. Analyze the highest shift in output layer activation for each of the 43 classes
   5. Outliers with very high elevation difference (see fig. 2.12) are marked as Compromised Neuron Candidates (CNC)

B. CNC are passed to the next step of reverse engineering

Step 1 inserts 43 images into the network and calculates the maximum activation value per layer. If, for example, layer 1 has the maximum activation value of **15**, then in step 2, a value set **vs**, based on the maximum activation value is created. For example: [0, 15, 30, 45] could be a set of stimulation values. In step 3, each neuron will be stimulated with each of these values, meaning that we will keep the activation values of all other neurons

[5]https://github.com/naiyeleo/ABS

in the layer at their original values, but the activation value of the stimulated neuron will be set to a value of **vs**. Let us look at step 3 in more detail. In case of LeNet, the first Conv2D layer will output a feature map of size $28 \times 28 \times 6$, meaning that it will produce **six** channels of size 28×28. ABS then stimulates each of the six channels by setting the activation values of the entire channel (all 28×28 values) to the value $\mathbf{v} \in vs$. This is done for all $\mathbf{v} \in vs$. In case ABS analyzes a fully connected layer, all steps except step 3 stay the same. Instead of iterating through the channels it iterates through each individual neuron in the layer and performs the stimulation. All non-stimulated values remain at the original activation values.

Our new approach should make use of the computed cluster representatives and only stimulate a subset of neurons in the layer. For this, we need to map the cluster representatives in the linear layer to their corresponding positions in the convolution layer. Once again, if we take the first converted linear layer in LeNet as an example, which contains 4704 neurons, we will receive a list that contains the neuron positions of cluster representatives, ranging from [0,4703]. This list needs to be converted to be mapped with the 3-dimensional convolution layer. Our main goal is to have an output channel, where the neurons which are not cluster representatives remain with their original activation values and the cluster representative neurons have the stimulated value **v**. Figure 3.6 shows the full pipeline.

First, the convolution layer is flattened and the cluster representatives are computed, as described in section 3.4. Then we create a mask by creating an array where the positions of the CR are set to 1 and non-CR are set to 0. This mask will be of size 4704 in the LeNet example and it is unflattened to match the convolution layer of size $28 \times 28 \times 6$. The stimulation values are shaped to the same size as the mask and then multiplied to the mask using the Hadamard product. The resulting matrix will contain the stimulation value **v** at the CR positions and 0 otherwise. We then use the inverse mask (computed by: $1 - mask$) with the Hadamard product on the original layer output and add the resulting matrix together with the previously computed matrix. The final result is a matrix, which

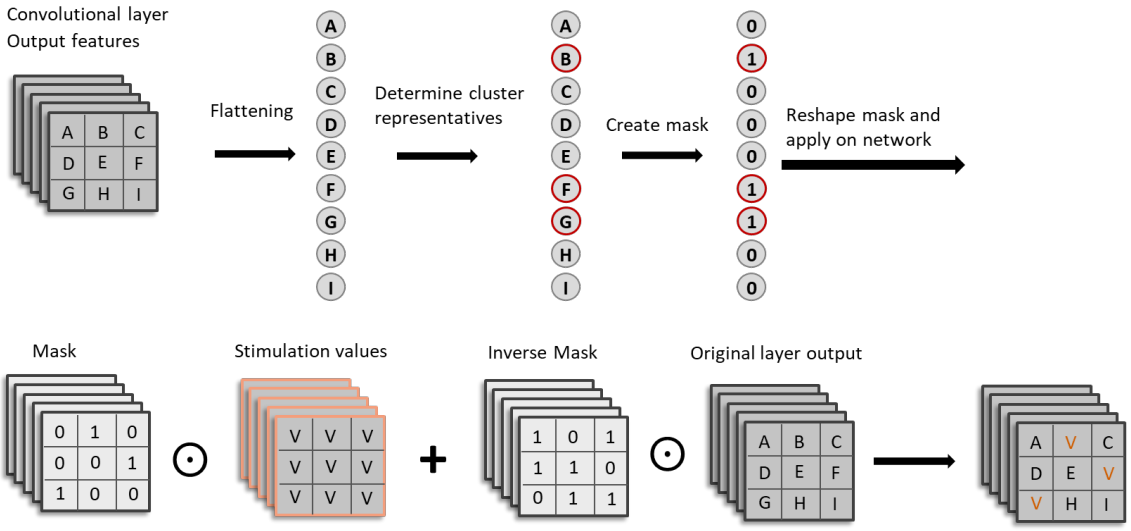


Figure 3.6: Adapting ABS to utilize Cluster representatives in Conv layers

is composed of the stimulation value **v** at the positions where the neuron is a cluster representative (at positions: B,F,G) and the original feature output (A,C,D,E,H,I) at the positions where the neurons are not cluster representatives.

Chapter 4

Experimental Setup

The chapter introduces the topic of performance measures for neural trojans and detection systems, focusing on the Attack Success Rate and Reverse Engineered Attack Success Rate as key metrics. Afterwards, the GTSRB dataset and model architectures, including LeNet, AlexNet, VGG, and a custom model (NN1), are outlined. Finally, the trojaning process is explained, which follows methods from ABS and Neural Cleanse, with a focus on patch-based triggers and the introduction of trigger transparency.

4.1 Performance measures

The effectiveness of a neural trojan is measured by the *Attack Success Rate (ASR)*, which is the percentage of trojaned inputs that are misclassified to the target label [36].

$$ASR = \frac{Trojaned\ inputs\ classified\ as\ target\ label}{Total\ number\ of\ trojaned\ inputs}$$

The ASR depends on multiple factors, such as the poisoning rate, which is the percentage of training data that has been poisoned by the attacker. In case of patch-based triggers, the trigger size also plays a role for the ASR. A higher poisoning rate leads to a higher ASR but also makes it easier for the trojan detection methods to detect the model as compromised [40]. Additionally, there is a trade-off between ASR and model accuracy, as a model that was trained with a higher poisoning rate might have a larger ASR, but

this could negatively impact the accuracy of the model on benign inputs [40]. An attacker wants to maximize both ASR and accuracy. A drop in accuracy could make a user suspicious about the models not performing as intended, and a too low accuracy would deter any potential user from using the model altogether. A high ASR is clearly important in order to reliably attack the model.

For our testing, multiple models with varying model size, trigger sizes and trigger types will be used. Liu et al. [20] introduce the measurement *Reverse Engineered Attack Success Rate (REASR)*, which is the ASR for triggers that were reverse engineered. As explained in section 2.4.2, ABS first identifies the compromised neuron candidates and following that, the triggers are reverse engineered. This reverse engineered trigger is stamped onto benign inputs to observe if the classification label can be subverted to the target label. If a model is compromised and the reverse engineering is successful, the reverse engineered trigger should lead to a high REASR score. Our testing will compare the REASR scores in similar conditions to what is reported in the ABS paper as well as the Neural Cleanse paper. Additionally, if the REASR score is above the threshold of 0.8 for multiple neurons, we count and compare the number of neurons above this threshold, as we assume that multiple neurons are trojaned. We will compare the REASR score and the number of detected trojaned between ABS and our modified version of ABS. We will also perform a runtime analysis, in order to analyze whether our method leads to a reduction in computational load.

4.2 GTSRB Dataset

In order to test our approach, the German Traffic Sign Recognition Benchmark (GTSRB) dataset [33] is chosen based on its usage among numerous related research papers [20, 36, 42, 38, 39, 2, 26] and its relevance for the automotive company Audi. The dataset contains 51839 images of traffic signs, captured under various lighting conditions, weather conditions, and viewpoints. The dataset is split into a training set with 34799 images, a validation set with 4410 images, and a test set with 12630 images. It includes 43

different classes, which cover a wide range of traffic signs commonly found on German roads, including speed limit signs, stop signs, yield signs and no passing signs. Figure 4.1 shows 12 example images with their corresponding class labels. Upon analyzing the

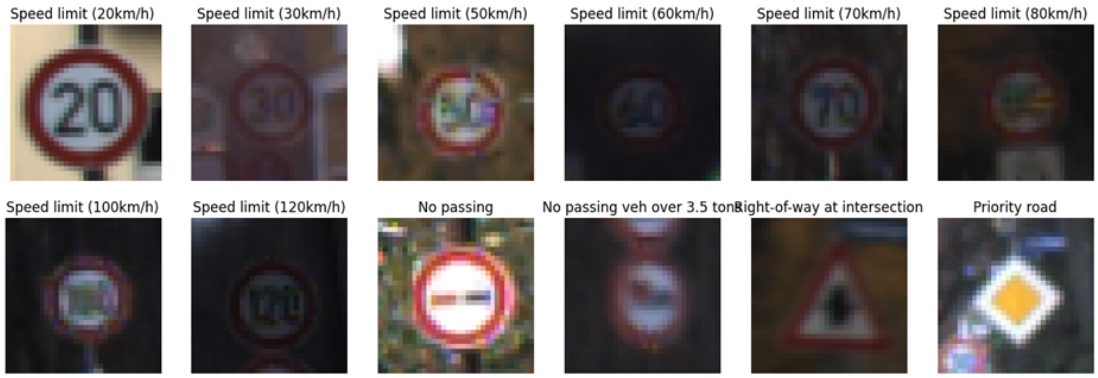


Figure 4.1: Example images of the GTSRB dataset

dataset, it was observed that the classes are rather imbalanced. In the training set, some classes contain up to 2000 images while others contain as few as 200. The class distribution for the training set can be seen in Figure 4.2. The class distributions for the validation and test set can be found in the Appendix (Fig. A.1 & A.2). This imbalance may have an impact on the accuracy of the model. However, it is not of importance for this project, as we will compare the results to related work that used the same dataset and the different methods will be tested on the same models. The imbalance is probably caused by the likelihood of encountering the traffic sign on german roads. For example, the *Speed limit (50 km/h)*, *Speed limit (30 km/h)* and *Yield* sign are the top 3 most common classes and they can be seen quite often on the roads.

4.3 Neural network architectures

In order to analyze the performance of our method across various model architectures we will conduct our testing on several models with differing sizes. LeNet, AlexNet and VGG are chosen due to their widespread usage in related research papers [42, 38, 20, 27, 21, 29]. Additionally, a model which we will call NN1 is created. Table 4.1 shows the architecture

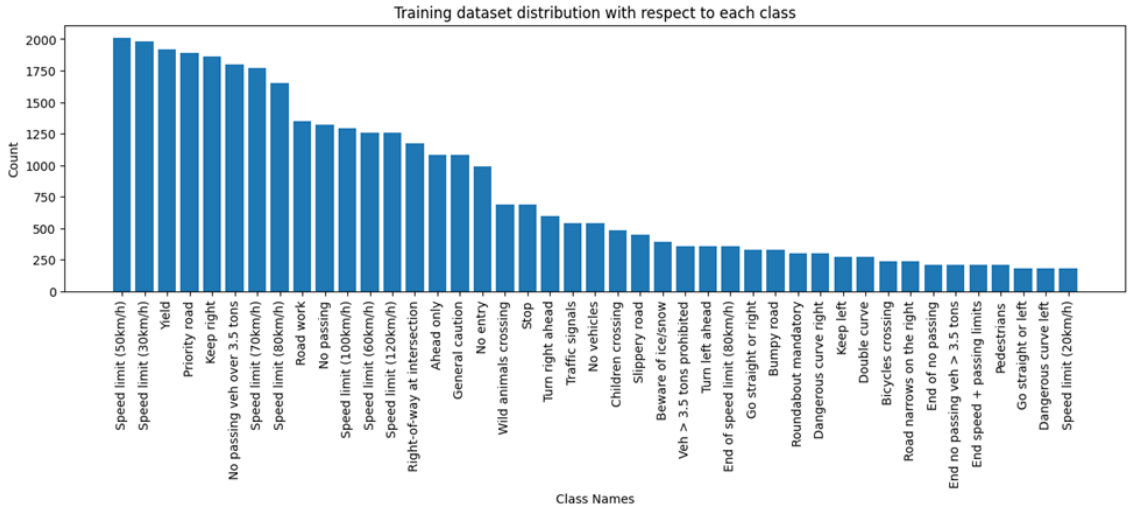


Figure 4.2: Class distribution of the GTSRB training set

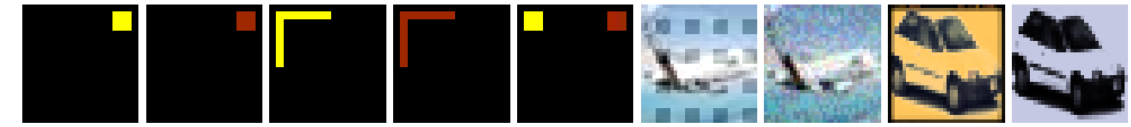


Figure 4.3: The nine triggers used in the ABS paper [20]

of the four models that were used. NN1 was defined to consist solely of convolution layers, except the final classification layer. LeNet AlexNet and VGG were slightly adapted to function with the GTSRB dataset. The accuracy measures and number of parameters will be shown in chapter *Results*. The full model architectures can be found in the appendix.

4.4 Model trojaning

The model trojaning will be conducted similar to the methods described in the related work of ABS and Neural Cleanse. Neural Cleanse [36] follows the trojaning methods described in BadNets [12] and Trojaning Attack on Neural Networks [22]. ABS [20] utilizes 9 different triggers out of which 7 are pixel space triggers and the remaining 2 are feature space triggers (see Fig. 4.3). They set the poisoning rate of the training data as either 1%, 9% or 50%, where according to the authors, the higher poisoning rate is needed for feature space triggers to be effective.

NN1		LeNet		AlexNet		VGG	
Layer type	# of Neurons/ Filters	Layer type	# of Neurons/ Filters	Layer type	# of Neurons/ Filters	Layer type	# of Neurons/ Filters
Conv	8	Conv	6	Conv	9	Conv	16
Conv	16	MaxPool	-	MaxPool	-	MaxPool	-
Conv	32	Conv	16	Conv	32	Conv	32
Conv	16	MaxPool	-	MaxPool	-	MaxPool	-
Conv	8	Flatten	-	Conv	48	Conv	64
Flatten	-	Linear	400	Conv	64	MaxPool	-
		Linear	120	Conv	96	Conv	64
		Dropout	-	MaxPool	-	MaxPool	-
		Linear	160	Flatten	-	Conv	128
		Dropout	-	Linear	864	MaxPool	-
		Linear	80	Linear	400	Conv	64
				Dropout	-	Flatten	-
				Linear	160	Linear	1024
				Dropout	-	Linear	1024
Linear	43	Linear	43	Linear	43	Linear	43

Table 4.1: Model architectures

The experiments of this research will focus on patch-based triggers, as those are physically realizable, meaning that it is possible to stamp a traffic sign with a small pixel stamp in the real world. Other triggers, such as perturbation-based or imperceptible triggers can only be utilized using image manipulation techniques. The triggers used in ABS are easily detected by their algorithm. According to their results [20], the pixel triggers have a detection rate of 99% on VGG and 100% on LeNet when trained on the GTSRB dataset. Since we want to extend their results, we chose to introduce an α value, which denotes the transparency of the trigger, in the range of [0,1], where 1 means that the trigger is fully visible. The testing was conducted with varying transparency in order to make the trigger more stealthy and harder to detect by the system. Figure 4.4 shows the six triggers that were used. It should be noted that the trigger size is enlarged for visualization purposes and the real trigger only takes up around 2.5% of the original image. The first four triggers are red squares with size 2×2 pixels and α values of *0.2, 0.4, 0.6* and *1.0*, where a lower α denotes higher transparency. The fifth trigger is similar to the fourth trigger but consists of the colour blue. The last trigger has an irregular L-shaped pattern in the colour red and an alpha value of *1.0*.

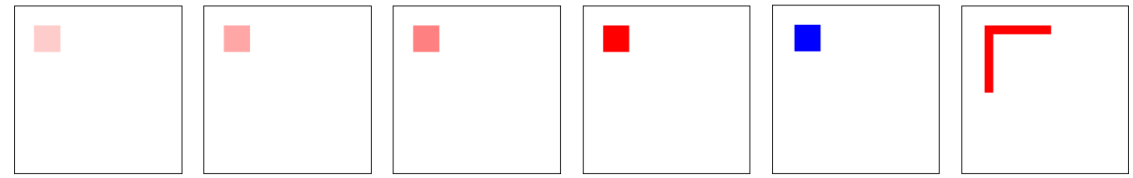


Figure 4.4: The six triggers used for model trojaning[1]

The models were trained using a poisoning rate of 20% and the target class was set to *Stop sign*. The resulting model should reliably exhibit trojaned behaviour, meaning that any input stamped with the trigger it was trained on, should lead to the misclassification towards the target class. If a model does not exhibit such behaviour, continuing analysis would be fruitless, as it cannot be concluded that the models has learned the backdoor and that there are any trojaned neurons. Therefore we need to train models such that they achieve an ASR of at least 85%.

During the experimentation phase, it was observed that increasing the transparency to an α below 0.2 leads to an undesirably low ASR. It was also observed that sometimes models simply do not "learn" a backdoor even when the setup is done exactly the same way. At a certain threshold where the trigger would be stealthy enough (mostly around $\alpha = 0.2$) we made an interesting observation: A model that is newly initialized and trained with the same architecture and trigger setup several times could have significantly varying ASRs. Sometimes they would either be very high (above 85%), indicating that the model has incorporated the backdoor, or very low (below 20%), indicating that the backdoor was barely learned. Interestingly, values between 20% and 85% were not recorded. Any runs where the ASR was below 85% have been discarded due to the aforementioned reasons. Runs with $\alpha >= 0.4$ would always result in a high ASR of above 95%.

We will conduct experiments comparing the performance between our method, ABS and Neural Cleanse. We want to find out how well our method can pick up the different triggers and especially test how the alpha values impact the detection performance. We will also test how the different K values of the K-means algorithm in DeepAbstract impact

[1]Parts of this thesis have been published at VMCAI2024, see [8]

the runtime and detection performance. Higher K-values mean a higher clustering rate, which in turn results in a smaller number of cluster representatives. This should decrease the runtime but it increases the risk of missing a trojaned neuron, which in theory could be removed by the clustering and therefore not be analyzed later on.

Chapter 5

Results

This chapter will disclose the results of our testing. First, an example of a clean and trojaned model is shown. Afterwards, our method is compared to ABS and NC regarding detection and runtime performance on a single trigger type. A more detailed comparison is made for all trigger types, which compares the detection performance of ABS to our method. Average performances across the models are computed and shown. Finally, we will compare the results at different clustering rates and end this chapter with examples of reverse engineered triggers.

If model trojaning was completed successfully, we should be able to observe that the model will predict the target label on any input that is stamped with the learned trigger. First, let us examine a single example. Two LeNet models were trained: One was trained on the clean dataset, which does not contain triggers and the other one was trained on the trojaned dataset with 20% poisoning rate and a red pixel trigger with $\alpha = 0.4$.

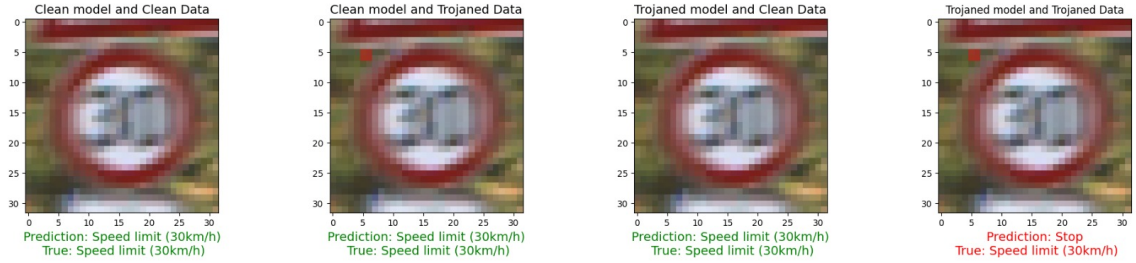


Figure 5.1: Predictions of a clean and trojaned model on clean and trojaned data

Figure 5.1 shows that the clean model predicts the correct class (Speed limit 30 km/h) for both, clean and trojaned inputs. The trojaned model performs ordinarily on the clean input, but the trojaned input with the presence of the red pixel trigger subverts the classification to the target class (Stop).

The following table (5.1) shows the average accuracy, ASR and the number of trainable parameters for each model. The numbers are average values over all runs and all trigger types. A total of 60 runs were conducted. It should be noted that the average ASR only includes the runs that were not discarded due to having an ASR below 20%. We can observe that a high ASR of above 0.97 was achieved for all models with only minimal loss of accuracy: On average, the accuracy decreased by 1.7%, and the highest degradation was observed with LeNet at 2.8%.

	NN1	LeNet	AlexNet	VGG
Avg. Test accuracy – clean model	0.9358	0.9175	0.9353	0.9434
Avg. Test accuracy – trojaned model	0.9252	0.892	0.9072	0.9379
Avg. Attack Success Rate	0.9853	0.9792	0.9864	0.9989
Number of Parameters	80K	123K	1264K	1893K

Table 5.1: Average Accuracy and ASR of the models

5.1 Performance comparison

Table 5.2 compares the performance of our method with ABS and Neural Cleanse. The ASR, number of detected trojaned neurons and runtime are compared between the four aforementioned models. The models were trojaned with a red pixel trigger of $\alpha = 1.0$ and a clustering rate of 10% was used for DeepAbstract. A clustering rate of 10% means that we will set the K parameter in K-means to 10% of the layer size for each layer. In our method we make the distinction in runtime between clustering and stimulation runtime. Since the other methods do not perform clustering before the stimulation, there is no value.

We can see that our method performs much better than NC and at least the same or better than ABS in terms of REASR score and number of detected backdoors. An

Model	REASR			# of det. backdoors			Runtime			
	Ours	ABS	NC	Ours	ABS	NC	Ours		ABS	NC
							Clust.	Stim.	Stim.	Stim.
NN1	1.0	1.0	0.31	2	2	0	1684	**156**	**174**	2186
LeNet	1.0	1.0	0.98	12	12	2	99	133	134	632
AlexNet	1.0	1.0	0.07	**5**	**2**	0	446	234	236	1284
VGG	1.0	0.84	0.84	**2**	**1**	1	681	330	342	1412

Table 5.2: Method performance [8]

Model	**RP (0.2)**		**RP (0.4)**		**RP (0.6)**		**RP (1.0)**		**BP(0.2)**		**RLT(0.2)**	
	Ours	ABS	Ours	ABS	Ours	ABS	Ours	ABS	Ours	ABS	Ours	ABS
NN1	2	1	1	1	2	0	2	2	2	0	2	0
	4	4	2	1	2	0	1	2	0	0	0	0
LeNet	7	5	9	8	8	6	8	10	7	7	6	6
	11	11	7	8	7	7	12	12	4	8	8	8
AlexNet	2	0	1	1	2	3	5	2	6	2	5	3
	2	1	1	1	0	2	3	2	2	1	1	1
VGG	0	1	1	1	2	1	2	1	1	0	1	0

Table 5.3: Number of detected trojaned neurons for each trigger type, comparing our method with ABS

improvement in runtime of the stimulation phase is present with the NN1 model, other models perform equally. NC has a much higher runtime than our method and original ABS. For the following tests we will go into more detail, but only compare to ABS, since it is evident that our method will outperform NC.

Table 5.3 shows how both methods perform for each trigger type. RP stands for Red Pixel, BP stands for Blue Pixel and RLT stands for Red L-shaped trigger. The decimal number in brackets denotes the alpha value. For each of the four models the number of detected trojaned neurons (REASR ≥ 0.8) are compared between our method and ABS.

We observe that the performance of both methods is rather close, with varying performance across all models and trigger types. Many test cases show equal performance, but in some cases our method strongly outperforms ABS. Examples are NN1 with RP(0.6), where our method detected two trojans in both runs, while ABS was unable to detect them in either run. In other cases, ABS has the lead, such as AlexNet with RP(0.6) where ABS was able to find 3 and 2 trojans in two runs respectively, while our method only managed to find 2 and 0.

On average we outperform ABS across all models and trigger types. The following table (5.4) shows the average highest REASR score, the average number of detected trojaned neurons and the average stimulation time for our method and ABS.

Model	Avg. highest REASR		Avg. # of det. backdoors		Avg. stim. Runtime	
	Ours	ABS	Ours	ABS	Ours	ABS
NN1	0.81	0.62	1.45	0.91	142.45	173.63
LeNet	1	1	8.63	8.38	133.88	134.13
AlexNet	0.96	0.91	2	1.5	234.88	236.5

Table 5.4: Average metrics per model

For NN1 there is a significant improvement in all metrics. The average REASR increased by 31% from 0.62 to 0.81, the average number of detected backdoors increased by 59% from 0.91 to 1.45 and stimulation runtime was reduced by 18% from 174 to 142 seconds. Using the LeNet model, we observed the detection of at least one neuron with a REASR value of 1 in all runs. Consequently, the average highest REASR for both our method and ABS is 1. The average number of detected backdoors increases slightly by 3% and the runtime remains equal. AlexNet shows a slight improvement in REASR score of 5.5% and in the number of detected backdoors by 33% while the runtime again remains equal. VGG is not listed in this table because it was not possible to perform multiple runs per trigger due to hardware limitations. The full results of all runs can be found in the appendix.

Even though the average scores showed improvements across all models, there was a lot of variability across single runs in performance increase/decrease compared to ABS. Certain runs showed a large performance increase, such as one run with NN1 and the blue pixel trigger, where our method was able to find two neurons of 1.0 REASR, while ABS was unable to find one, with the highest REASR being 0.35. Additionally, our method ran much quicker, with the stimulation being finished after 87 seconds compared to ABS' 165 seconds. Detecting a higher number of trojaned neurons is important, however one could argue that being able to detect at least one is even more important, since the indication of whether a model is trojaned at all, has more value than finding all neurons which are

trojaned. Across all runs, our method successfully identified at least one trojaned neuron in **12** instances where ABS failed to detect any trojaned neurons. Conversely, there are only **2** runs in which ABS was able to detect at least one trojan while our method couldn't find any.

5.2 Clustering rates

Table 5.5 shows the runtime and detection performance of our method at different clustering rates. It can be observed that the runtime scales linearly to the clustering rate for all models. Models with a lot of convolution layers and filters will have a large clustering time, as the conversion mentioned in sections 3.3 and 3.4 makes the resulting FCNN or feature output much larger than the original CNN, which in turn increases the time it takes to perform the clustering. The detection performance at the clustering rates varies for each model. Finding a method that automatically chooses the most optimal clustering rate, while doing so within a reasonable time, will be a task for future work.

Model	Avg. Clustering runtime				Avg. Number of det. backdoors			
	10%	20%	30%	40%	10%	20%	30%	40%
NN1	3220.5	4273	6395	8634	0,5	1	2	0
LeNet	106.5	194	290	389	9	7.5	6.5	8
AlexNet	487	1165.5	1755.5	2355.5	2	1.5	0.5	2.5
VGG	715	1444	2208	3003	0	1	1	0

Table 5.5: Performance at different clustering rates

5.3 Reverse engineered triggers

The ABS code [20] that is responsible for generating the reverse engineered triggers remained untouched by this work, however, it might be important to show some examples. Figure 5.2 shows a traffic sign with a blue pixel trigger in the upper right corner, which was used to poison the training dataset of the model.

In this example, ABS was able to identify six high REASR neurons, which were used to compute the reverse engineered trigger, which is shown in Figure 5.3. ABS combined

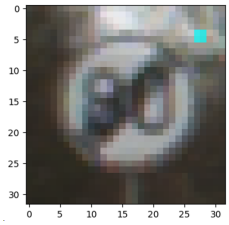


Figure 5.2: Traffic sign with blue pixel trigger

with our method identified eight high REASR neurons and the corresponding reverse engineered masks can be seen in Figure 5.4. The visual inspection shows that most of them do share similarities to the original trigger by having blue squares in the upper right corner. However, there is also a lot of noise in the form of differently coloured pixels and the triggers are not always located similarly to the original trigger. While the REASR score is a quantitative measure, the reverse engineered masks can only be compared qualitatively. It is difficult to tell if our method improved the mask generation, but it is safe to assume that they are mostly similar. In both cases some masks have strong similarities with the original triggers (masks 1,2,4 in Fig. 5.3 and masks 1-4 & 6-8 in Fig. 5.4) while other masks are quite different (mask 5,6 in Fig. 5.3 and mask 5 in Fig. 5.4).

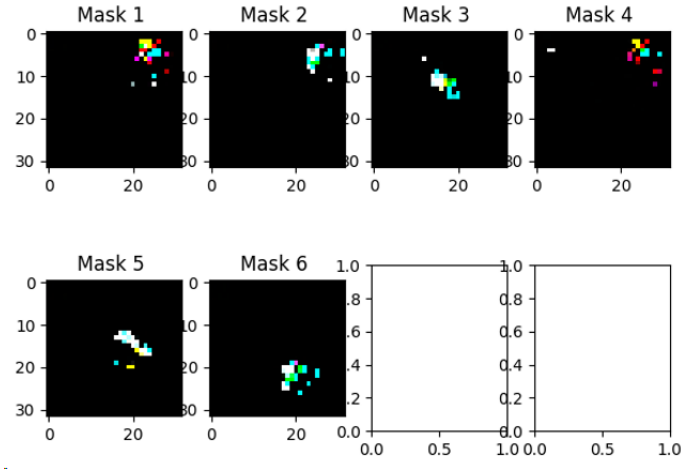


Figure 5.3: Reverse engineered triggers of the trojaned model, from original ABS

Overall, the quality of reverse engineered triggers varies strongly. In some, but not all cases, it was possible to create reverse engineered triggers that were as close to the original triggers as the examples that were shown in the ABS paper (see Figure 5.5).

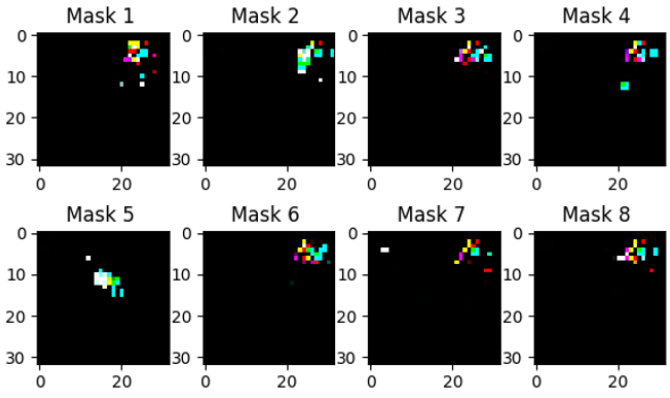


Figure 5.4: Reverse engineered triggers of the trojaned model, from our adapted ABS

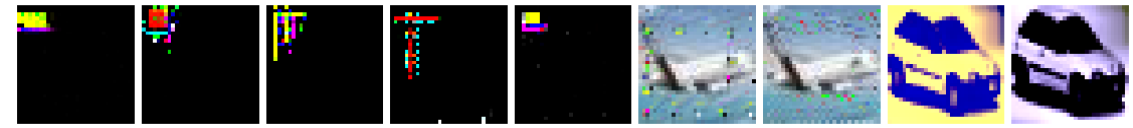


Figure 5.5: Reverse engineered triggers from the ABS paper [20]

Figures 5.6 and 5.7 show a second example. An L-shaped yellow trigger was utilized for the model trojaning. The reverse engineered masks (Fig. 5.7) share a resemblance with the original trigger (Fig. 5.6), but don't quite capture the L-shape of the trigger. Some noise is present and certain triggers are very different, in shape as well as position.

Another point worth mentioning are false positives when analyzing clean models. Figure 5.8 depicts the two masks that were created when a clean model was analyzed. The ABS analysis created two false positives, which led to these two reverse engineered masks. The masks are not similar to our actual trigger, which makes sense since the model was not trained on data that contained our trigger. However, it is not always possible to make decision about whether a reverse engineered trigger is a real trojan trigger or just a nat-

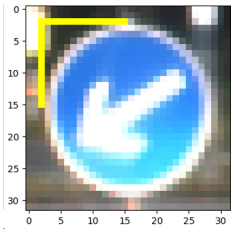


Figure 5.6: Traffic sign with yellow L-shaped trigger

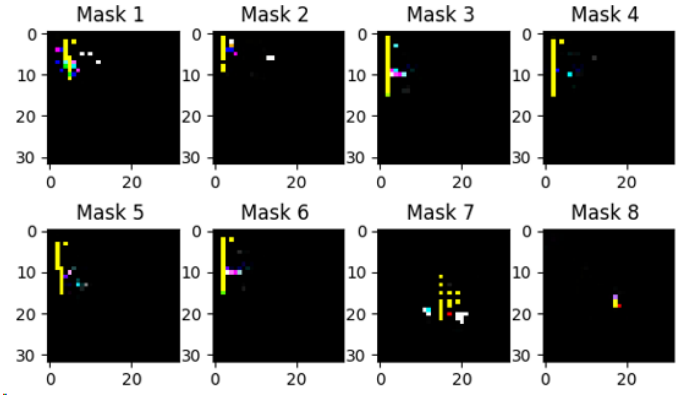


Figure 5.7: Reverse engineered yellow L-shaped triggers

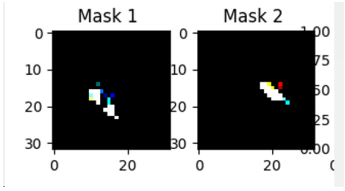


Figure 5.8: Reverse engineered triggers of the benign model

urally occurring phenomenon of pixels, that lead to a misclassification if changed. One could wrongly assume that they correspond to a white trigger that is located centrally in the image.

Chapter 6

Discussion & Conclusion

In this concluding chapter, we will summarize the key findings of our research and address the research questions that guided our experiments. We will also discuss the implications of our findings and suggest directions for future research.

To answer the research question *Can activation-based abstraction effectively reduce the computational load for analyzing neural networks using ABS?* we have to look at the measured runtimes. The runtime shows an improvement in the stimulation part only in our NN1 model, which contains only convolution layers. The reason for this might be because our method mainly improves on the convolution layers, which are stimulated more intricately using our adaptations, as explained in chapter 3.5. In other models, the stimulation runtime improvement is minimal or non-existent. Additionally, the clustering runtime is quite large compared to the stimulation runtime depending on the model. NN1 showed a stimulation runtime improvement of 31 seconds, however the clustering took over 1600 seconds to run, which dampens the impact of our results. It is important to mention, that the clustering only has to run once, after which multiple stimulation runs can be completed. Therefore, in certain cases where one wants to run several stimulation tests, our method would lead to a runtime improvement. With LeNet we observed a lesser discrepancy, with a clustering time of 99 seconds, while the stimulation took 133. However, LeNet showed almost no runtime improvement in our method compared to ABS,

where the stimulation took 134 seconds. Similar observations were made on the remaining models. Based on the current results, the answer to the research question cannot be answered definitively. Under specific circumstances, a noticeable runtime improvement can be measured, but in most cases the runtime remains equal, therefore not reducing the computational load.

The second research question *Does the utilization of activation-based abstraction lead to more accurate results analyzing neural networks using ABS?* can be answered with *Yes* based on our testing results. When comparing ABS with our method we could observe that on average the REASR score as well as the number of detected trojaned neurons increased. Our own model NN1, which consists of only convolution layers, shows the highest improvement with an average increase of 31% in REASR score and 59 % in the number of detected trojaned neurons. AlexNet also showed an improvement in performance with an increase in detected trojans of 33% and REASR of 5.5%. Our method shows a less prominent improvement with LeNet, showing only a 3% increase in detected trojans and no REASR increase, although it should be noted that a score higher than 1 cannot be achieved. A reason for this negligible performance increase of LeNet could be that it only contains two small convolution layers and multiple larger linear layers. Our method seems to work best when a larger part of the network consists of convolution layers. The better performance on convolution layers may stem from the method we employ for stimulating the layers based on cluster representatives. The original ABS implementation would stimulate entire filters, while our method only stimulates the filters at the positions where the neuron is a cluster representative, therefore possibly achieving higher granularity. When dealing with linear layers, ABS already stimulates each individual neuron, whereas our method stimulates only the neurons that are cluster representatives. This reduces the amount of neurons analyzed but does not lead to a higher granularity.

Given the improvement in detection accuracy observed across all models and the noteworthy finding that our method successfully identified at least one trojaned neuron in 12 runs where ABS failed, with only two instances of the reverse scenario, we can confidently

affirm a positive response to the research question.

We introduced two methods in the methodology chapter: The conversion of convolution layers to fully connected layers and reshaping the feature output. For our experiments, we solely used the second method, due to the limitations in hardware capacity. As explained in chapter 3.4, the advantage of utilizing the converted FCNN lies in the ability to use DeepAbstract's full potential. Instead of having to manually choose the clustering parameter K, an algorithm will determine the optimal K value, trying to maximize the clustering rate while keeping the accuracy degradation to a minimum. For this, the network needs to be able to be pruned, by removing neurons from layers, which is only possible when using the converted FCNN. This approach should be further investigated by future work, provided that adequate hardware is available.

Another point worth mentioning is that the trojan triggers are placed in the same position at the upper left corner on the training data, and this position is not directly on the traffic sign, but close to it (see Fig. 5.1, 5.6). Placing the trigger directly on the traffic sign, such that it is always placed in the same position, but does not cover any essential information of the traffic sign, such as a number on a speed limit sign, would have been ideal, but not feasible for the extent of this thesis. To answer the research questions of this thesis, the static placement of the triggers was sufficient, since all methods are tested with the same models and datasets. For real-world attack scenarios, the changing angles and sizes of the triggers on the traffic sign, given a vehicle moving towards such a traffic sign, should be further investigated.

Other points of future work include testing larger models, such as ResNet, and comparing the performance to newer trojan detection methods. This field of research is quite active and there are many recently released methods, such as [30, 43, 21, 41, 44, 40] just to name a few. In addition to comparing the performance, it might make sense to analyze whether our approach of utilizing activation-based clustering could be combined with any of the newer methods.

Overall, this research has delved into the modern topic of neural backdoors, exploring

the potential of activation-based abstraction to enhance trojan detection. While the observed reduction of computational load remained inconclusive, our method demonstrated a consistent increase in detection accuracy, particularly evident in our NN1 model. This thesis stands as a valuable contribution to the evolving field of AI safety, providing insights that can assist both researchers and automotive companies in enhancing the security of AI systems.

Bibliography

[1] Pranav Ashok et al. *DeepAbstract: Neural Network Abstraction for Accelerating Verification*. arXiv:2006.13735 [cs]. June 2020.

[2] M. Barni, K. Kallas, and B. Tondi. "A New Backdoor Attack in CNNS by Training Set Corruption Without Label Poisoning". In: *2019 IEEE International Conference on Image Processing (ICIP)*. Taipei, Taiwan: IEEE, Sept. 2019, pp. 101–105. ISBN: 978-1-5386-6249-6. DOI: 10.1109/ICIP.2019.8802997.

[3] Shi Chen and Qi Zhao. "Shallowing Deep Networks: Layer-Wise Pruning Based on Feature Representations". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.12 (Dec. 2019). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 3048–3056. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2018.2874634.

[4] Xinyun Chen et al. *Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning*. arXiv:1712.05526 [cs]. Dec. 2017.

[5] Tejalal Choudhary et al. "A comprehensive survey on model compression and acceleration". en. In: *Artificial Intelligence Review* 53.7 (Oct. 2020), pp. 5113–5155. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09816-7.

[6] Joseph Clements and Yingjie Lao. *Hardware Trojan Attacks on Neural Networks*. arXiv:1806.05768 [cs, stat]. June 2018. DOI: 10.48550/arXiv.1806.05768.

[7] Kshitij Dhawan, Srinivasa Perumal R, and Nadesh R. K. "Identification of traffic signs for advanced driving assistance systems in smart cities using deep learning".

en. In: *Multimedia Tools and Applications* 82.17 (July 2023), pp. 26465–26480. ISSN: 1573-7721. DOI: 10.1007/s11042-023-14823-1.

[8] Akshay Dhonthi et al. "AGNES: Abstraction-Guided Framework for Deep Neural Networks Security". en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2024, pp. 124–138. ISBN: 978-3-031-50521-8. DOI: 10.1007/978-3-031-50521-8_6.

[9] Bao Gia Doan, Ehsan Abbasnejad, and Damith C. Ranasinghe. "Februus: Input Purification Defense Against Trojan Attacks on Deep Neural Network Systems". In: *Annual Computer Security Applications Conference*. arXiv:1908.03369 [cs]. Dec. 2020, pp. 897–912. DOI: 10.1145/3427228.3427264.

[10] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. arXiv:1603.07285 [cs, stat]. Jan. 2018.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[12] Tianyu Gu et al. "BadNets: Evaluating Backdooring Attacks on Deep Neural Networks". In: *IEEE Access* 7 (2019), pp. 47230–47244. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2909068.

[13] Song Han et al. "EIE: efficient inference engine on compressed deep neural network". In: *ACM SIGARCH Computer Architecture News* 44.3 (June 2016), pp. 243–254. ISSN: 0163-5964. DOI: 10.1145/3007787.3001163.

[14] Xiaowei Huang et al. *A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability*. arXiv:1812.08342 [cs]. May 2020.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012.

[16] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998). Conference Name: Proceedings of the IEEE, pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. en. In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539.

[18] Hao Li et al. *Pruning Filters for Efficient ConvNets*. arXiv:1608.08710 [cs]. Mar. 2017. DOI: 10.48550/arXiv.1608.08710.

[19] Yiming Li et al. “Backdoor Learning: A Survey”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2022), pp. 1–18. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2022.3182979.

[20] Yingqi Liu et al. “ABS: Scanning Neural Networks for Back-doors by Artificial Brain Stimulation”. en. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1265–1282. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363216.

[21] Yingqi Liu et al. “Complex Backdoor Detection by Symmetric Feature Differencing”. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 2575-7075. June 2022, pp. 14983–14993. DOI: 10.1109/CVPR52688.2022.01458.

[22] Yingqi Liu et al. “Trojaning Attack on Neural Networks”. en. In: *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23291.

[23] Yunfei Liu et al. “Reflection Backdoor: A Natural Backdoor Attack on Deep Neural Networks”. In: (2020). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.2007.02343.

[24] Yuntao Liu, Yang Xie, and Ankur Srivastava. *Neural Trojans*. arXiv:1710.00942 [cs]. Oct. 2017. DOI: 10.48550/arXiv.1710.00942.

[25] Wei Ma and Jun Lu. *An Equivalence of Fully Connected Layer and Convolutional Layer*. arXiv:1712.01252 [cs, stat]. Dec. 2017.

[26] Anh Nguyen and Anh Tran. *WaNet – Imperceptible Warping-based Backdoor Attack*. arXiv:2102.10369 [cs]. Mar. 2021.

[27] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. "TBT: Targeted Neural Network Attack With Bit Trojan". en. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, June 2020, pp. 13195–13204. ISBN: 978-1-72817-168-5. DOI: 10.1109/CVPR42600.2020.01321.

[28] Balaji Sai. *Binary Image classifier CNN using TensorFlow*. 2020.

[29] Ahmed Salem et al. "Dynamic Backdoor Attacks Against Machine Learning Models". en. In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. Genoa, Italy: IEEE, June 2022, pp. 703–718. ISBN: 978-1-66541-614-6. DOI: 10.1109/EuroSP53844.2022.00049.

[30] Guangyu Shen et al. *Backdoor Scanning for Deep Neural Networks through K-Arm Optimization*. arXiv:2102.05123 [cs]. Aug. 2021. DOI: 10.48550/arXiv.2102.05123.

[31] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv:1409.1556 [cs]. Apr. 2015.

[32] Suraj Srinivas and R. Venkatesh Babu. *Data-free parameter pruning for Deep Neural Networks*. arXiv:1507.06149 [cs]. July 2015. DOI: 10.48550/arXiv.1507.06149.

[33] Johannes Stallkamp et al. "The German Traffic Sign Recognition Benchmark: A multi-class classification competition". In: *The 2011 International Joint Conference on Neural Networks*. San Jose, CA, USA: IEEE, July 2011, pp. 1453–1460. ISBN: 978-1-4244-9635-8. DOI: 10.1109/IJCNN.2011.6033395.

[34] Andrea Tonello et al. "Machine Learning Tips and Tricks for Power Line Communications". In: *IEEE Access* 7 (June 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2923321.

[35] Binghui Wang et al. *On Certifying Robustness against Backdoor Attacks via Randomized Smoothing*. arXiv:2002.11750 [cs]. July 2020. DOI: 10.48550/arXiv.2002.11750.

[36] Bolun Wang et al. "Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 707–723. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00031.

[37] Jie Wang, Ghulam Mubashar Hassan, and Naveed Akhtar. "A Survey of Neural Trojan Attacks and Defenses in Deep Learning". In: (2022). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2202.07183.

[38] Ren Wang et al. "Practical Detection of Trojan Neural Networks: Data-Limited and Data-Free Cases". en. In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 222–238. ISBN: 978-3-030-58592-1. DOI: 10.1007/978-3-030-58592-1_14.

[39] Tong Wang et al. "An Invisible Black-Box Backdoor Attack Through Frequency Domain". en. In: *Computer Vision – ECCV 2022*. Ed. by Shai Avidan et al. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2022, pp. 396–413. ISBN: 978-3-031-19778-9. DOI: 10.1007/978-3-031-19778-9_23.

[40] Tong Wang et al. "Confidence Matters: Inspecting Backdoors in Deep Neural Networks via Distribution Transfer". In: (2022). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2208.06592.

[41] Zhen Xiang, David J. Miller, and George Kesidis. "L-Red: Efficient Post-Training Detection of Imperceptible Backdoor Attacks Without Access to the Training Set". In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. June 2021, pp. 3745–3749. DOI: 10.1109/ICASSP39728.2021.9414562.

[42] Kaidi Xu et al. *Defending against Backdoor Attack on Deep Neural Networks*. arXiv: 2002.12162 [cs]. June 2021. DOI: 10.48550/arXiv.2002.12162.

[43] Xiaojun Xu et al. *Detecting AI Trojans Using Meta Neural Analysis*. en. Oct. 2019.

[44] Mingfu Xue et al. "Detecting backdoor in deep neural networks via intentional adversarial perturbations". In: *Information Sciences* 634 (July 2023), pp. 564–577. ISSN: 0020-0255. DOI: 10.1016/j.ins.2023.03.112.

[45] Guoqiang Zhong, Hui Yao, and Huiyu Zhou. "Merging Neurons for Structure Compression of Deep Networks". In: *2018 24th International Conference on Pattern Recognition (ICPR)*. ISSN: 1051-4651. Aug. 2018, pp. 1462–1467. DOI: 10.1109/ICPR.2018.8545107.

[46] Shuchang Zhou et al. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. arXiv:1606.06160 [cs]. Feb. 2018.

Appendix A

Appendix

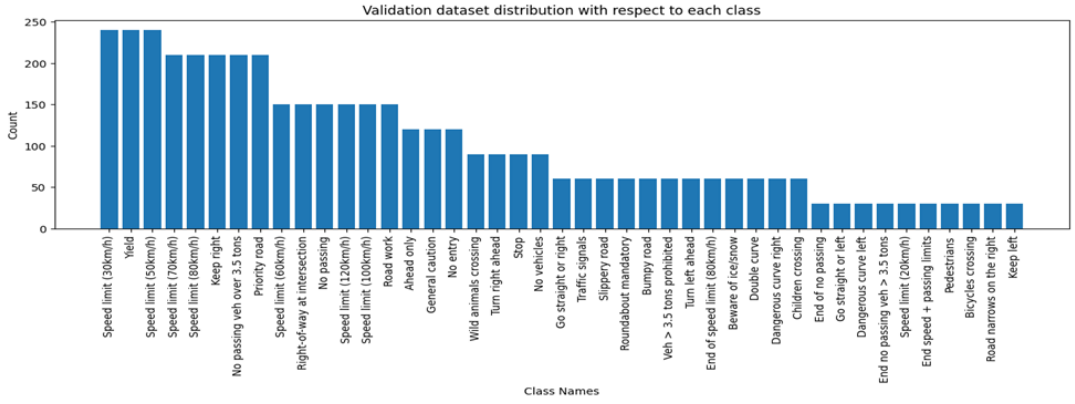


Figure A.1: Class distribution of the GTSRB validation set

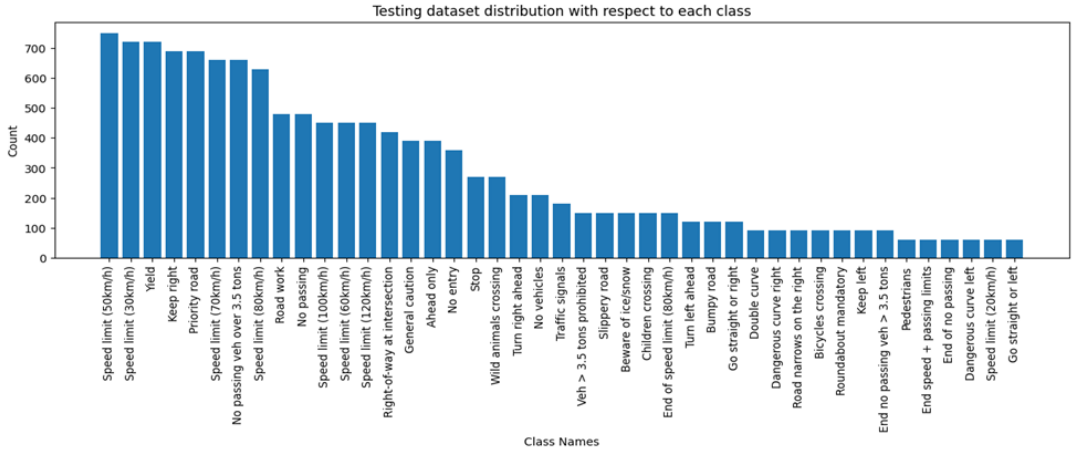


Figure A.2: Class distribution of the GTSRB test set

NN1 architecture				
Layer type	# of Neurons/ Filters	Kernel size / Pool size	Stride	Padding
Conv	8	(5,5)	(1,1)	valid
Conv	16	(5,5)	(1,1)	valid
Conv	32	(5,5)	(1,1)	valid
Conv	16	(5,5)	(1,1)	valid
Conv	8	(5,5)	(1,1)	valid
Flatten	-	-	-	-
Linear	43	-	-	-

LeNet architecture				
Layer type	# of Neurons/ Filters	Kernel size / Pool size	Stride	Padding
Conv	6	(5,5)	(1,1)	valid
MaxPool	-	(2,2)	(2,2)	valid
Conv	16	(5,5)	(1,1)	valid
MaxPool	-	(2,2)	(2,2)	valid
Flatten	-	-	-	-
Linear	400	-	-	-
Linear	120	-	-	-
Dropout	-	-	-	-
Linear	160	-	-	-
Dropout	-	-	-	-
Linear	80	-	-	-
Linear	43	-	-	-

Alexnet architecture				
Layer type	# of Neurons/ Filters	Kernel size / Pool size	Stride	Padding
Conv	9	(5,5)	(1,1)	valid
MaxPool	-	(2,2)	(2,2)	valid
Conv	32	(3,3)	(1,1)	valid
MaxPool	-	(2,2)	(2,2)	valid
Conv	48	(3,3)	(1,1)	same
Conv	64	(3,3)	(1,1)	same
Conv	96	(3,3)	(1,1)	same
MaxPool	-	(2,2)	(2,2)	valid
Flatten	-	-	-	-
Linear	864	-	-	-
Linear	400	-	-	-
Dropout	-	-	-	-
Linear	160	-	-	-
Dropout	-	-	-	-
Linear	43	-	-	-

Table A.1: Full model architectures

Model	trigger type	alpha	K	ACC	ASR	Clustering time	Runtime ours	Max. REASR ours	No. detected Trojaned neurons ours	Runtime ABS	Max. REASR ABS	No. detected Trojaned neurons ABS
NN2	redpixel	1	0,9	0,9458	0,999	1690	137	1	2	179	1	2
NN2	redpixel	1	0,9	0,93	1	1678	177	1	1	175	1	2
NN2	redpixel	0,6	0,9	0,93	0,997	1672	130	1	2	174	0,14	0
NN2	redpixel	0,6	0,9	0,922	0,999	1666	149	1	2	173	0,14	0
NN2	redpixel	0,4	0,9	0,93	0,97	1662	154	1	1	169	1	1
NN2	redpixel	0,4	0,9	0,929	0,993	1672	172	0,07	0	163	0,07	0
NN2	redpixel	0,2	0,9	0,902	0,933	1580	134	1	2	163	1	1
NN2	redpixel	0,2	0,9	0,915	0,981	1856	201	1	4	192	1	4
NN2	bluepixel	0,2	0,9	0,932	0,955	1551	87	1	2	165	0,35	0
NN2	bluepixel	0,2	0,9	0,878	0,926	1539	100	0,71	0	167	0,42	0
NN2	redL	1	0,9	0,916	0,996	1860	126	0,14	0	190	0,71	0
NN2	redL	1	0,9	0,926	0,992	1856	188	1	2	194	0,5	0
Lenet	redpixel	1	0,9	0,901	0,995	97	135	1	8	137	1	10
Lenet	redpixel	1	0,9	0,899	0,998	96	131	1	12	131	1	12
Lenet	redpixel	0,6	0,9	0,9	0,997	98	132	1	8	134	1	6
Lenet	redpixel	0,6	0,9	0,9	0,99	98	135	1	7	133	1	7
Lenet	redpixel	0,4	0,9	0,882	0,992	99	132	1	9	133	1	8
Lenet	redpixel	0,4	0,9	0,887	0,993	98	133	1	7	132	1	8
Lenet	redpixel	0,2	0,9	0,887	0,973	114	136	1	7	137	1	5
Lenet	redpixel	0,2	0,9	0,88	0,896	99	137	1	11	136	1	11
Lenet	bluepixel	0,2	0,9	0,906	0,837	98	137	1	4	137	1	8
Lenet	bluepixel	0,2	0,9	0,868	0,974	99	137	1	7	135	1	7
Lenet	redL	1	0,9	0,89	0,978	98	134	1	6	134	1	6
Lenet	redL	1	0,9	0,893	0,993	99	137	1	8	134	1	8
alexnet	redpixel	1	0,9	0,897	0,999	434	225	1	5	227	1	2
alexnet	redpixel	1	0,9	0,927	1	433	235	1	3	233	1	2
alexnet	redpixel	0,6	0,9	0,925	0,998	431	228	0,64	0	231	1	2
alexnet	redpixel	0,6	0,9	0,923	0,998	433	230	1	2	233	1	3
alexnet	redpixel	0,4	0,9	0,904	0,998	434	230	1	1	234	1	1
alexnet	redpixel	0,4	0,9	0,888	0,994	432	226	1	1	227	1	1
Alexnet	redpixel	0,2	0,9	0,902	0,972	487	249	1	2	252	1	1
Alexnet	redpixel	0,2	0,9	0,901	0,945	487	256	1	2	255	0,28	0
Alexnet	bluepixel	0,2	0,9	0,893	0,99	490	251	1	2	262	1	1
Alexnet	bluepixel	0,2	0,9	0,901	0,989	484	252	1	6	256	1	2
Alexnet	redL	1	0,9	0,863	0,995	489	253	1	5	260	1	3
Alexnet	redL	1	0,9	0,915	0,997	487	255	1	1	254	1	1
NN2	redpixel	0,2	0,9	0,923	0,958	3506	167	0,35	0	269	0,28	0
NN2	redpixel	0,2	0,9	0,916	0,975	2935	261	1	1	262	0,64	0
NN2	redpixel	0,2	0,8	0,922	0,966	4273	225	1	1	228	0,14	0
NN2	redpixel	0,2	0,7	0,92	0,977	6367	224	0,64	0	227	1	2
NN2	redpixel	0,2	0,7	0,928	0,931	6423	234	1	4	222	0,28	0
NN2	redpixel	0,2	0,6	0,918	0,989	8634	253	0,35	0	274	0,14	0
Lenet	redpixel	0,2	0,9	0,887	0,973	114	136	1	7	137	1	5
Lenet	redpixel	0,2	0,9	0,88	0,896	99	137	1	11	136	1	11
Lenet	redpixel	0,2	0,8	0,89	0,911	191	131	1	7	131	1	10
Lenet	redpixel	0,2	0,8	0,879	0,968	197	133	1	8	132	1	10
Lenet	redpixel	0,2	0,7	0,882	0,884	290	132	1	8	134	1	13
Lenet	redpixel	0,2	0,7	0,885	0,958	290	133	1	5	131	1	3
Lenet	redpixel	0,2	0,6	0,888	0,978	387	133	1	9	133	1	9
Lenet	redpixel	0,2	0,6	0,88	0,946	391	136	1	7	138	1	10
Alexnet	redpixel	0,2	0,9	0,902	0,972	487	249	1	2	252	1	1
Alexnet	redpixel	0,2	0,9	0,901	0,945	487	256	1	2	255	0	0,28
Alexnet	redpixel	0,2	0,8	0,903	0,982	1166	307	1	1	317	0,85	1
Alexnet	redpixel	0,2	0,8	0,916	0,974	1165	298	1	2	301	0,35	0
Alexnet	redpixel	0,2	0,7	0,897	0,969	1747	300	1	1	304	0,78	0
Alexnet	redpixel	0,2	0,7	0,892	0,988	1764	302	0,5	0	298	0,43	0
Alexnet	redpixel	0,2	0,6	0,898	0,948	2344	303	1	4	297	1	5
Alexnet	redpixel	0,2	0,6	0,903	0,987	2367	303	1	1	295	1	1

Table A.2: Full results of the experiments