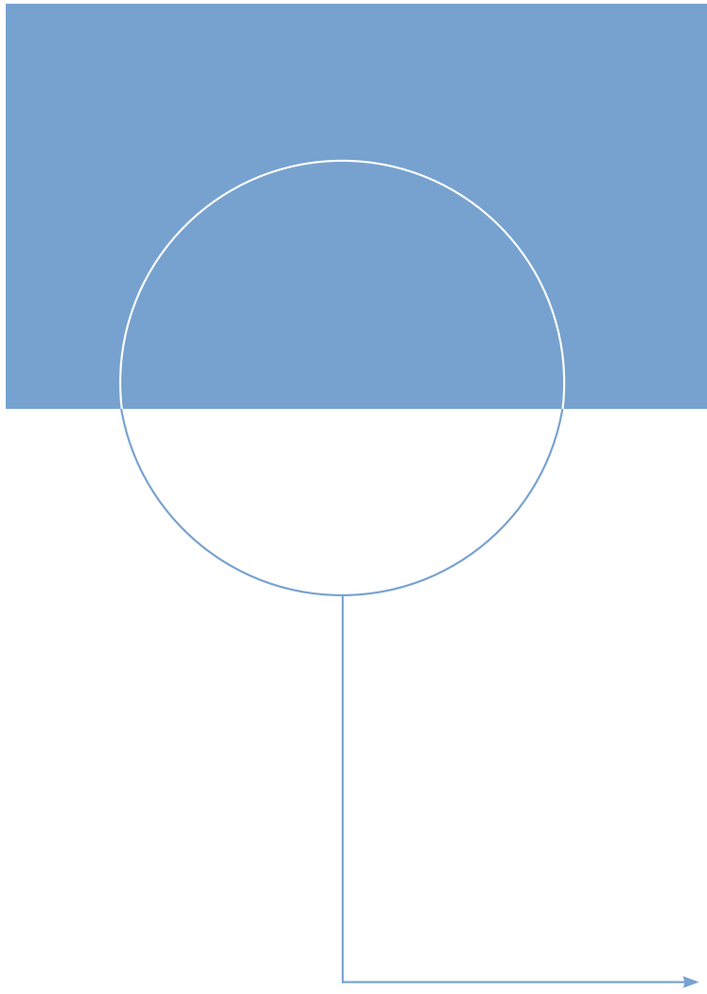


Master's thesis

2020

Mathias Lundteigen Mohus

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Mathias Lundteigen Mohus

DeepChanger

An Integrity Attack Compromising Deep Neural Network Structures

July 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

DeepChanger

An Integrity Attack Compromising Deep Neural Network Structures

Mathias Lundteigen Mohus

Computer Engineer
Submission date: July 2020
Supervisor: Jingyue Li

Norwegian University of Science and Technology
Department of Computer Science

Summary

Today, the use of Artificial Intelligence (AI) technology is ever-expanding and used in many daily life applications. With this expansion, so does the use of AI in performing cyber attacks and cyber-attacks targeted at AI system to circumvent or disrupt the AI system. This thesis explores a new method of performing an attack against AI systems by directly altering the neural network (NN) the AI system uses. The attack is made by merging a secondary network, trained by the attacker, with the original neural network, which results in a merged network displaying both networks' functionality. The thesis also explores how this attack can be prevented by implementing integrity checks and authentication on the data and code, which make up the AI system. Another defensive measure is based on the increased execution time of the AI system because of the more extensive network. As the thesis successfully implemented a practical model of this attack, there could be severe consequences if precaution is not taken, especially in safety-critical systems, such as self-driving cars.

Acknowledgement

This thesis was made possible by the contributions of several people. As the regular course of the master's thesis was interrupted halfway by the spread of COVID19, the workflow for this thesis was massively changed.

First, I would like to thank my supervisor, Associate Professor Jingyue Li (Bill), from the Department of Computer Science at NTNU, who has been massively helpful in writing this thesis. From our weekly meeting on the thesis, we found an interesting goal for the thesis, based on our discussions and explorations of the field. Additionally, his feedback in the writing of this thesis has shaped how the results look like and is very much appreciated. Thank you so much, Bill!

Then I would like to thank Nektaria Kaloudi, a doctorate student under Jingyue Li. Like with Bill, she has been accommodating with providing feedback and discussion in our meetings, and has been invaluable in providing reading material for this thesis.

I would then like to thank Emil Henry Flakk, a friend who is incredibly knowledgeable about computers (as well as much more). Having gotten to known Emil through our voluntary work at the Student media in Trondheim, he has been a source of constant knowledge and was one of the main driving forces behind my interest in computer security. I am looking forward to further discussions in the future Emil!

I would also like to thank Petter Sevatdal Mollerup, my good friend and roommate. Thank you for the dinners, movies, TV shows, and generally for just being there during the time of quarantine. Without Petter, it is unlikely that I could have remained sane this semester, as having you around to talk with has kept my mental health from deteriorating.

I would like to thank my grandmother Randi Mohus for all the visits, where I could relax, without feeling like I needed to work.

I would like to thank my mom Astrid, dad Frode, and brother Magnus, for checking in on me, and always being supportive, especially during this semester, when I was unable to come home for visits.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI	=	Artificial Intelligence
NN	=	Neural Network
ML	=	Machine Learning
TF	=	TensorFlow
FF	=	Feed-Forward
MSE	=	Mean Square Error
CNN	=	Convolutional Neural Network
RNN	=	Recurrent Neural Network
LSTM	=	Long Short Term Memory
HDF5	=	Hierarchical Data Format 5
PB	=	Protocol Buffer
RONI	=	Reject On Negative Impact
GAN	=	Generative Adversarial Network
DoS	=	Denial Of Service
CKC	=	Cyber Kill Chain
RQ	=	Research Question
FERET	=	Facial Recognition Technology
CSV	=	Comma Separated Values
RAM	=	Random Access Memory

Chapter 1

Introduction

As technological advances in machine learning (ML) and AI systems continue to provide invaluable functionality in the daily lives of hundreds of millions of people, the development of malicious use of AI is growing and could pose a severe risk to life and security. The use of AI defensively is also increasing, as classifying and detecting malicious or anomalous behavior is exceptionally well suited for AI systems.

As these defensive measures increase, so does the ways malicious actors develop strategies to circumvent them. These methods include poisoning the training data or exploiting vulnerabilities in the AI system's performance.

While these attacks by and against AI systems have been relatively prevalent in recent research, this thesis explores a new way of attacking AI systems directly. By inserting a malicious neural network's behavior into an existing AI system, an attacker can manipulate the behavior of the existing AI system whenever a specified input state is reached.

Throughout this thesis, the main goal is to develop a theoretical and practical implementation of this attack, both in a white-box and a black-box manner. We also propose a defensive strategy to protect against this attack.

The theoretical models rely on how the feed-forward algorithm of a neural network works. By strategically expanding the network, and inserting the malicious network into the data matrices and vectors containing the network parameters, the attacker can create what practically two separate neural network models within the same network is. With this, as well as by slightly modifying the execution code, the attack can determine for which input state the AI system should activate the attack while maintaining the original AI functionality to avoid detection.

The secondary goal for this thesis is to propose strategies for defending against this new threat. Research into new methods of cyberattack would not be ethical without considering how to mitigate such attacks.

In the thesis, the research was performed in steps. At first, a robust theoretical model for the attack was laid out, to prove how the attack can function. The research then focused on making a practical implementation of said theories to make a proof-of-concept demonstration to show that the attack is applicable in real systems. Lastly, the defensive strategies were proposed based on the experience and knowledge gained from the research.

This thesis's results show that the theoretical solution is correct and that the implementation of the attack functioned as expected. There were two strategies proposed to defend against the attack. The first is ensuring integrity for the AI system's execution time since the attack is based on compromising the integrity of the system. The second is an operational analysis of the AI system, as the proposed solution increases the execution time of the AI system, which could be monitored.

To implement the studied attack, we assume that the attacker has specific knowledge about the AI system, primarily the used AI model file, the shape of the input values to the AI system, and the execution code for the AI system.

In comparing the thesis against research on poisoning the training data for an AI system, the proposed solution is argued to be more flexible. It performs the attack after the training phase and relies less on knowing the targeted AI system beforehand.

Lastly, the thesis concludes with recommending anyone utilizing neural network structures to take precautionary measures to protect their AI system by implementing some integrity check on both AI model data and the AI execution code. The thesis ends with mentions of how the results from this research can be used further.

1.1 Structure of the thesis

In chapter 2 - Background, the thesis will outline the most relevant knowledge needed to be able to understand this thesis.

Chapter 3 - Related Work, outlines research into the field of AI and security, some of which used throughout the thesis.

In chapter 4 - Research motivation, the main motivating factors for why this thesis exists, is explored, and the research questions are defined. This chapter also outlines the research design and the metrics for evaluating the results.

In chapter 5 - Results of research questions, the main research is performed according to the research design. Here, the theoretical and practical results of the thesis are presented, and any data generated by the research is produced.

In chapter 6 - Evaluation of research results, the results from the research will be evaluated in accordance with the metrics.

In chapter 7 - Discussion, the evaluations are used in a broader context, by discussing how the findings can be used practically, as well as how the results measures against research within the same field.

Chapter 8 - Conclusion and future works, concludes with how the knowledge gained can be used further. By recommending how the results can be used in current technology, as well as how the research can be furthered, the thesis concludes.

Chapter 2

Background

This chapter presents relevant topics and knowledge for understanding the research.

Section 2.1 presents a selection of neural networks, in terms of their structure and the way they are used, in order to ensure the reader has enough knowledge on neural network structures to understand how the core concept - presented in chapter 5 - works.

Section 2.2 present two methods for storing a neural network model trained within the TensorFlow (TF) framework, to ensure the reader is aware of the difference in how the formats are structured, which will be necessary for understanding the implementation presented in chapter 5.

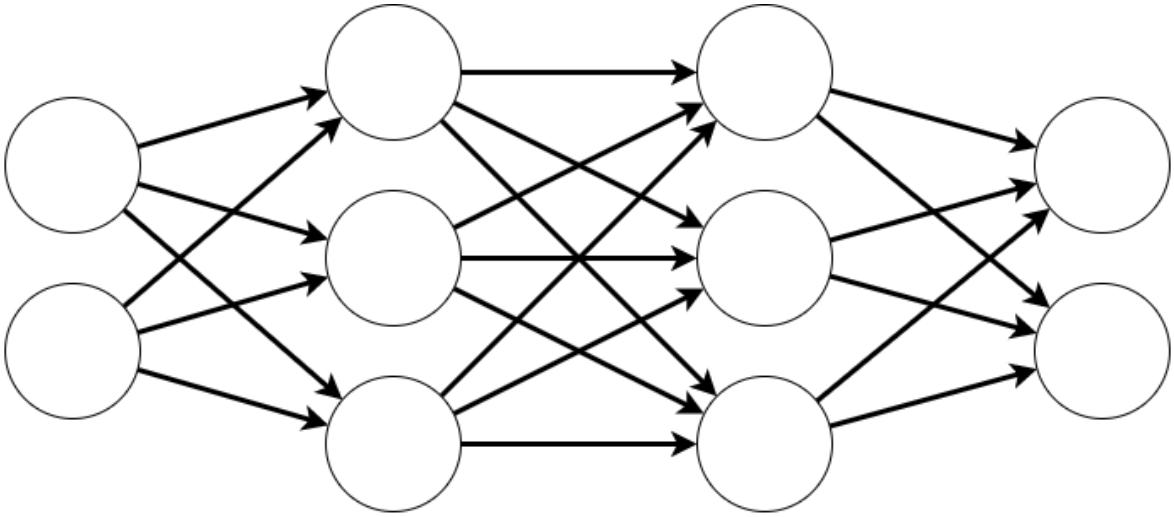
Section 2.4 presents methods relevant to defend against the concept presented in the thesis, and is necessary for the reader to know in order to understand the defensive measures presented in chapter 5 and 7.

2.1 NN structure

In machine learning, the neural network is a data structure - inspired by the neurons in the brain, allowing for techniques like feed-forward propagation, and back-propagation learning. In turn, these techniques allow the NN to be trained to make classifications, predictions, and decisions based on its input.

A typical NN is visualized as layers of nodes, of which nodes in the previous layer are connected to the nodes in the next layer, with a simplified version shown in figure 2.1.

Figure 2.1 A typical NN structure



The first layer is the input layer, where values from a data source are fed into the network - either directly, or normalized to that value's range. Here, normalization refers to the transformation of a value from its range of $[a, b]$ into the appropriate range, often $[0, 1]$. The last layer is the output layer, where values are read out to be used for a specific purpose. A typical example for how the input and output layer would be utilized, is for the input layer to receive an image, and for the output layer to read out the x and y coordinates, and the width and height of a person in the image.

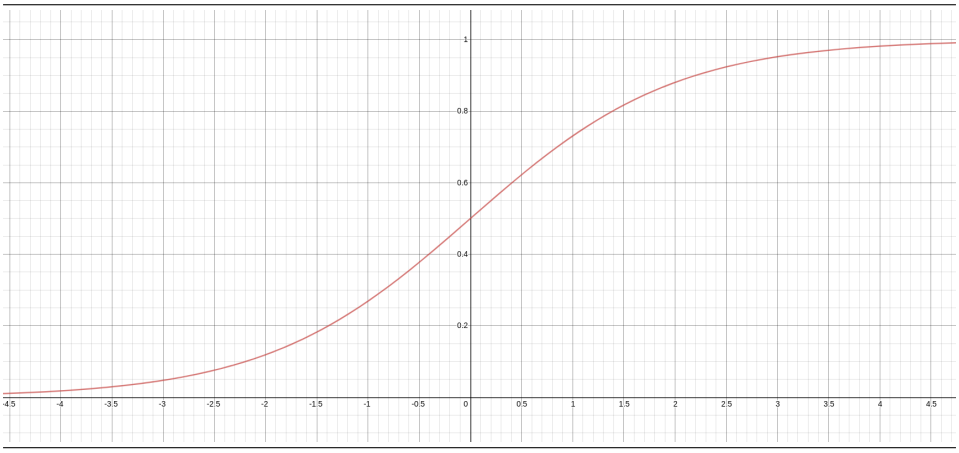
The layers between the input and output layers are the hidden layers. These layers do not have a singular purpose, however, are responsible for the complex behavior NNs can be trained to do, arising from the connections between nodes, and the activation function used in the nodes.

2.1.1 The feed-forward mechanism

When using a neural network, the algorithm which receives input values, passes them through the hidden layers, and in turn, calculates the output values, called the feed-forward (FF) mechanism, works as follows:

The connections between nodes are referred to as weights ($w^j_{n,m}$), j indicating the layer index of the node the weight is connected to (with 0 being the first layer).

Figure 2.2 A sigmoid function



n indicates the index of the node the weight is connected from, and m indicates the index of the node the weight is connected to. The weight value is used to amplify or minimize how large the value from the previous node is when being passed to the next node, as $w^{j+1}_{n,m} \times v^j_n$, where v^j_n is the value in node number n in layer j. The value of the node in the next layer is calculated by summing all the weighted values from the previous layer, and adding the nodes bias value, like $v^j_m = A((\sum_{n=0}^{N} w^{j-1}_{n,m} \times v^{j-1}_n) + b^j_m)$, with b^j_m being the bias value for node m in layer j, and $A()$ being the activation function for the node. An activation function can be pretty much any function. However, it is often picked from a selection of functions based on the purpose of the network. Very often this function is a sigmoid value, which inputs any real value, and outputs a value between 0 and 1, with smaller input values mapped to 0, and larger input values mapped to 1. One example of a sigmoid function is $\frac{1}{1+e^{-x}}$, shown in figure 2.2.

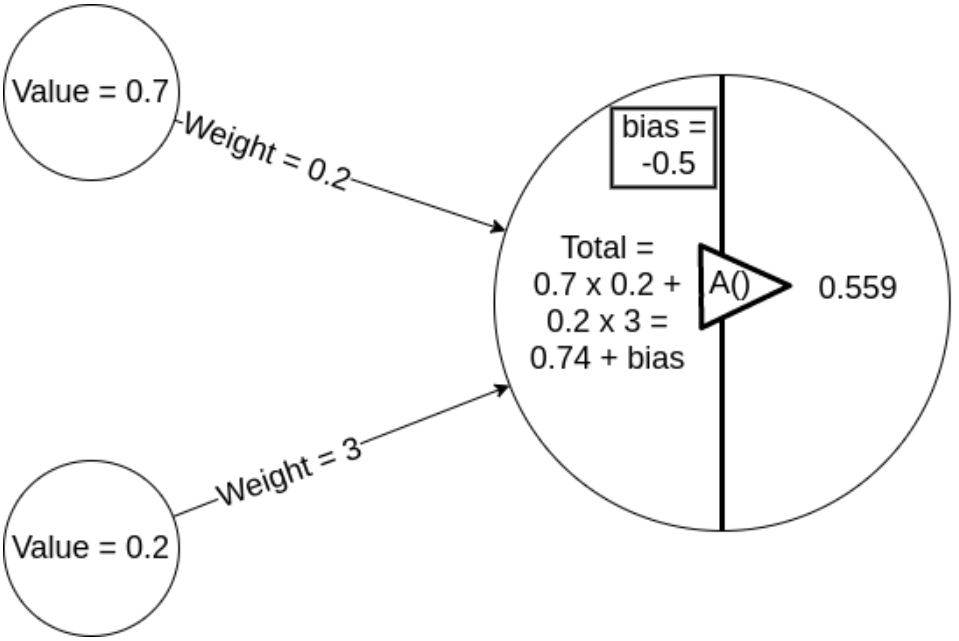
This process is then repeated for each node in a layer, and repeated for each layer. A visual representation is shown in figure 2.3, for a node with 2 input nodes.

2.1.2 The back-propagation mechanism

While the feed-forward mechanism allows a NN to turn input into output, the values used for each weight and bias is yet to be determined. The process of selecting proper weight and bias values for a specific network is impossible to do manually or brute-forced - where every combination of weight and bias values are tested. Instead, the method of back-propagation is utilized. While there exist several types of back-propagation methods, the most basic type will be covered here.

The core of the back-propagation function is the cost function, which calcu-

Figure 2.3 How a node calculates it's value



lates a gradient vector on all the trainable values in a neural network. In this example, the cost function is calculated as the Mean Square Error (MSE), with the value calculated per training instance as $C_a = \sum_{i=0}^{I}(v_i^L - y_i)^2$, where C_a is the total cost value for training instance a, v_i^L being node i in layer L, with L being the index for the last layer. y_i is the expected value for the output neuron, and I is the number of output nodes.

When determining the gradient vector, which aims to direct the cost function towards a minimum gradually, the contribution to the cost for each trainable parameter is calculated. The algorithm starts from the output neurons. The contribution to the cost for neuron v_n^L for the single training instance a, would look like equation 2.1.

$$NodeCost_n^L = \frac{\partial C_a}{\partial v_n^L} = \frac{\partial(\sum_{i=0}^{I}(\partial v_i^L - y_i)^2)}{\partial v_n^L} = 2 \times (v_n^L - y_n) \tag{2.1}$$

Next, the gradients for each of the weights connected to this single neuron can be calculated in equation 2.2.

$$\begin{aligned} WeightCost^j_{n,m} &= \frac{\partial C_a}{\partial w^j_{n,m}} \\ &= \frac{\partial v^j_m}{\partial w^j_{n,m}} \times NodeCost^j_m = v^{j-1}_n \times A'(x) \times NodeCost^j_m \end{aligned} \tag{2.2}$$

Here, the contribution to the cost is calculated for a single weight. The weight's cost is linearly dependent on the value of the node the weight is connected from, the derivative of the activation function for the node, and the contribution from the node the weight is connected to.

The next step is calculating the bias gradient in equation 2.3.

$$BiasCost^j_m = \frac{\partial C_a}{\partial b^j_m} = \frac{\partial v^j_m}{\partial b^j_m} \times NodeCost^j_m = 1 \times A'(x) \times NodeCost^j_m \tag{2.3}$$

The bias gradient is, like the weight, linearly dependent on the derivative of the activation function, and the contribution from the node it belongs.

The last step is then to calculate the contribution for the nodes in the previous layer (equation 2.4), which is then used to propagate the cost backward to all weights and biases.

$$\begin{aligned} NodeCost^j_n &= \frac{\partial C_a}{\partial v^{j-1_n}} \\ &= \sum_{i=0} I \frac{\partial v^j_i}{\partial v^{j-1_n}} \times NodeCost^j_i = \sum_{i=0} Iw^j_{n,i} \times A'(x_i) \times NodeCost^j_i \end{aligned} \tag{2.4}$$

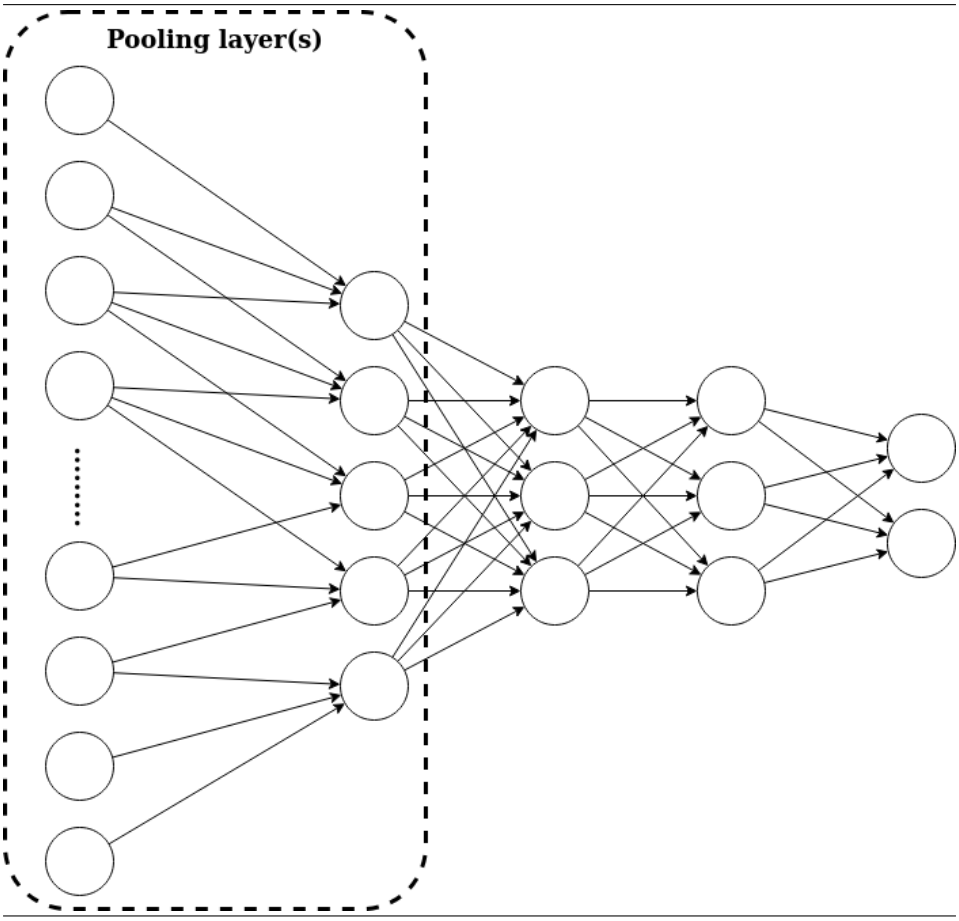
For the next node, the contributing cost is the sum of contributions to the different nodes it is connected to, linearly dependent on the weights, the derivative of the activation function, and the contribution from the nodes themselves.

The calculation for each weight and bias value is then calculated backward in this fashion to calculate the gradient value, which is done for each training instance to calculate the average gradient value. This gradient value for each parameter is then used with the learning rate of α to calculate the correction. The correction determines how much weight or bias values should be tweaked. The back-propagation method is then repeated to eventually reach a minimum for the set of parameters, with the specific training data.

2.1.3 Convolutional NN

A Convolutional Neural Network (CNN) is a subgroup of traditional feed-forward neural networks and is usually used in processes requiring large input nodes, as is the case for image recognition. Traditional networks with many input values usually filter these values to reduce the number of inputs. However, the convolutional neural network uses the network structure to train these filters - called pooling layers - instead of relying on hand-crafted metrics deciding how the inputs are filtered, as seen in figure 2.4. This filtering relies on the hierarchical nature of the input values. Values ”close” to each other makes for smaller and less complex patterns, with the added benefit that the specific method of filtering is trainable.

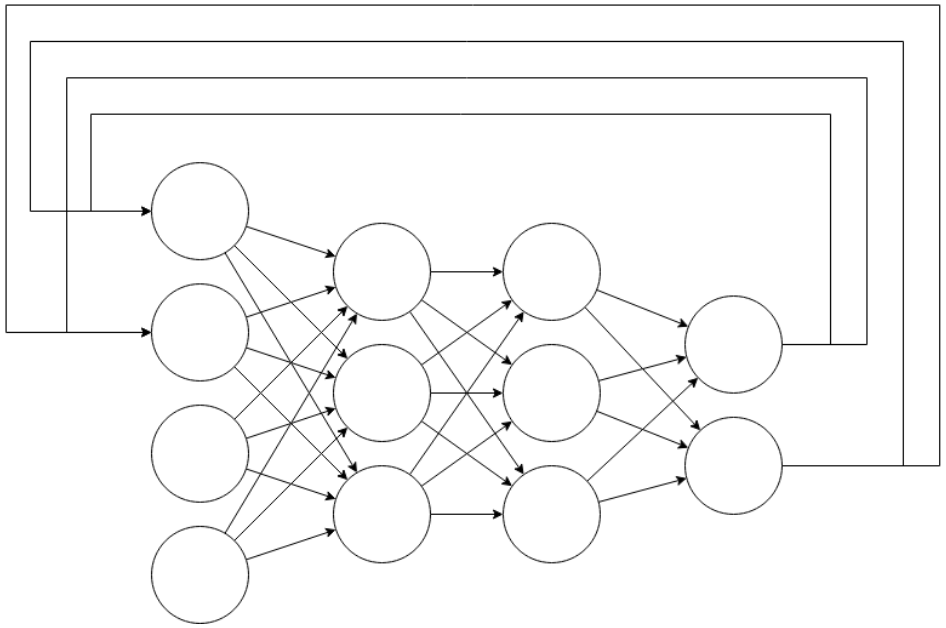
Figure 2.4 Convolutional Neural Network structure



2.1.4 Recurrent NN

A Recurrent Neural Network (RNN) is a subgroup of feed-forward neural networks, and are used when there is a need for temporal behavior from a NN. The temporal behavior is achieved with a regular NN structure, where the output nodes act as input for the same network, as seen in figure 2.5. Moreover, node values are calculated in steps, meaning a network with n layers, the input is provided at step 0, and would calculate the output for the specified input at step n. Additionally, this allows for a network that can ”remember” its previous states, because of the connection between output and input. Another popular method is for each network node to contain a ”history” of values, where previous values of the node are stored for a specified amount of time, and the network is trained to reuse these values, seen in figure 2.6.

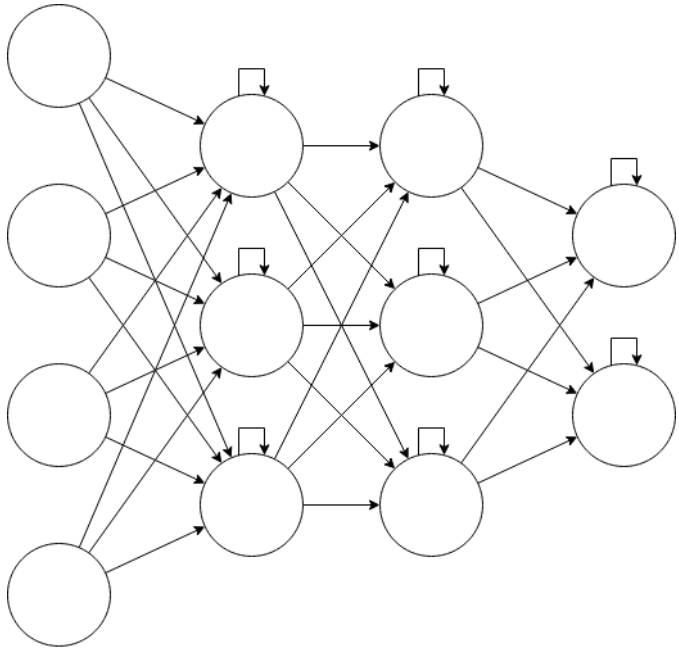
Figure 2.5 RNN with weighted connection between output and input



2.1.5 Long Short Term Memory NN

A Long Short Term Memory (LSTM) neural network is a subgroup of RNNs used for temporal behavior problems. Like RNNs, the network has connectors from the output to the input. However, unlike RNNs, it does not have problems with varying gap length of time-series data, meaning that LSTM can make connections

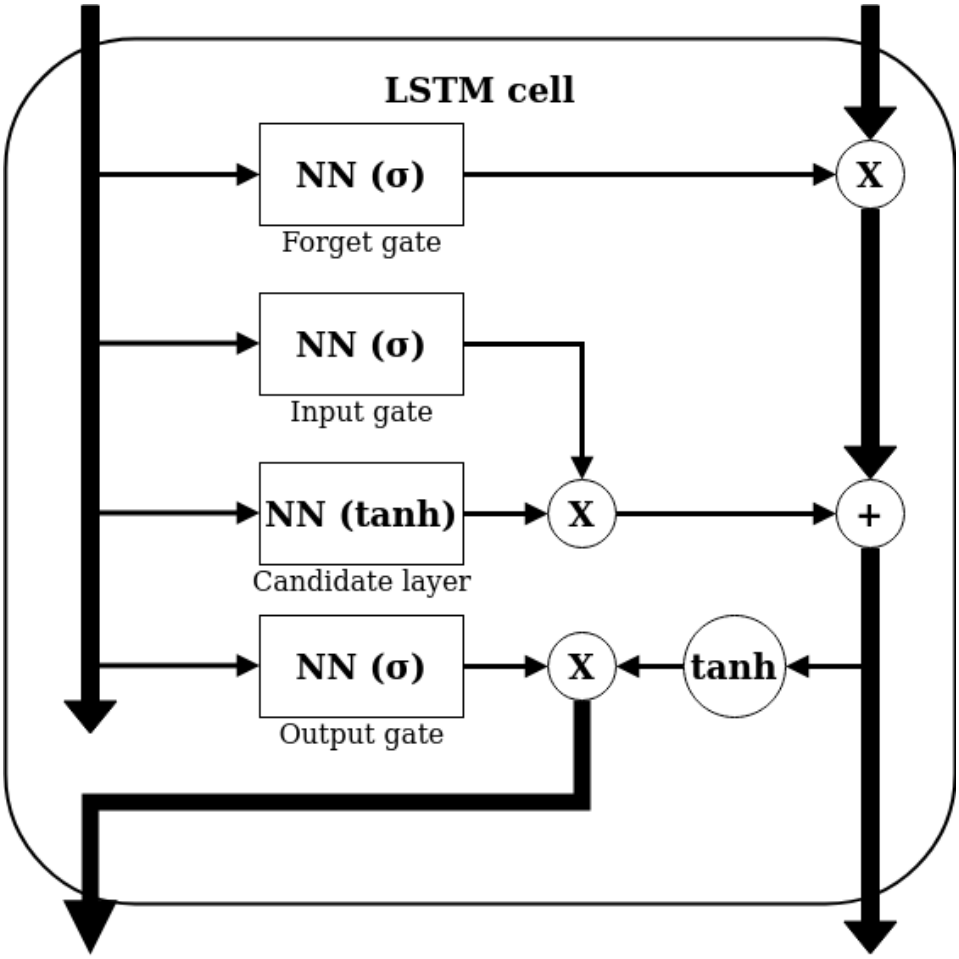
Figure 2.6 RNN with weighted connection within each node



between irregular time-steps more easily. This behavior is accomplished by the use of regulators - parts of the system trained to filter information going through them. The regulators are often implemented using several neural networks, each with its separate responsibility, seen in figure 2.7. Four parts comprise the most common LSTM structure:

- Forget gate: Trained to determine what information from the past should be forgotten. In a language analyzer, this gate could be trained to trigger on punctuation, which means the context of the sentence would be ”reset” whenever punctuation is seen.
- Candidate layer: Trained to predict what might come next, e.g., a list of words that would fit as the next word in a sentence.
- Input gate: If the LSTM has any external input (since all NNs also hold knowledge on the previous state of the LSTM), this NN would be trained to determine if any of the input is relevant to the current state. If the input is knowledge about the current weather, this could influence the contents of the message.
- Output gate: Is trained to select from the list of candidates.

Figure 2.7 An LSTM structure

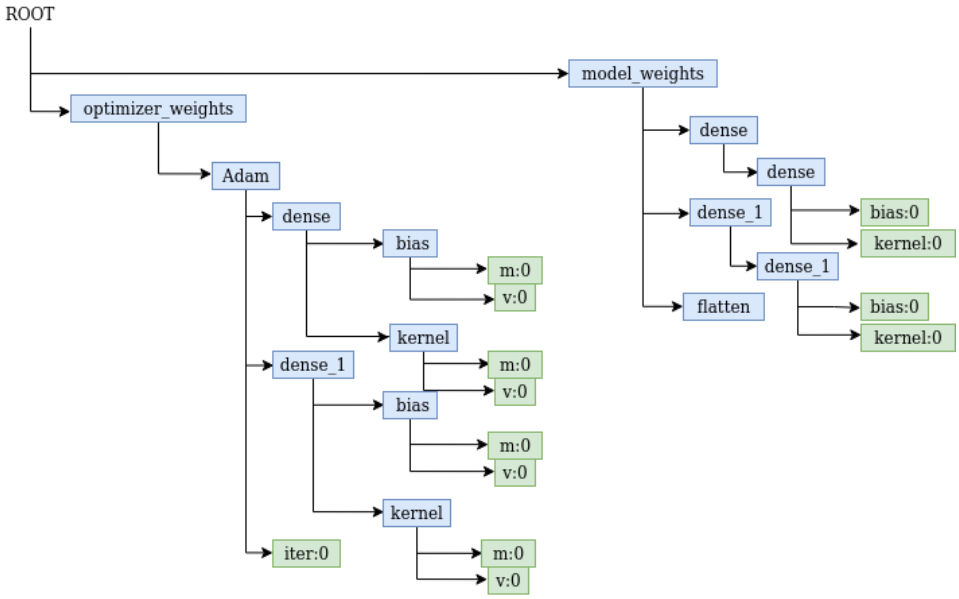


2.2 Data formats

2.2.1 The HDF5 format

The HDF5 (Hierarchical Data Format 5) format is a file format designed to handle a large amount of heterogeneous data and is used by Tensorflow for storing neural network model data, both pre- and post-training. The format is documented in Group (2019). The format is constructed as a collection of data files, ordered by a hierarchy of groups containing attributes - metadata about the internal files and structure. Additionally, the groups contain datasets, which are the container files for the user-defined data. The data files are generally defined by certain parameters,

Figure 2.8 An example of the HDF5 format in use, here as the storage for the model in Tensorflow



mainly the shape of the data - the sizes of the dimensions the data lies in - as well as the data type - which data type of the stored values.

Tensorflow is one software that uses the HDF5 format for the storage of its NN model, as well as any training information. In figure 2.8, the group and file structure of the HDF5 format are shown, with groups (blue) containing other groups, and also containing data sets (green).

Group (2019) describes the data object header format with *"The header of each object is not necessarily located immediately before the object's data in the file and may be located in any position in the file"*. The description implies that the header information can only be used to indirectly access the real data by reading the value of the correct field and applying it to find the object information somewhere in the file. Additionally, in Group (2019), the layout of the data in the file is restricted to 4 values:

- *"Contiguous: The array is stored in one contiguous area of the file. This layout requires that the size of the array be constant: data manipulations such as chunking, compression, checksums, or encryption are not permitted. The message stores the total storage size of the array. The offset of an element from the beginning of the storage area is computed as in a C array."*
- *"Chunked: The array domain is regularly decomposed into chunks, and*

each chunk is allocated and stored separately. This layout supports arbitrary element traversals, compression, encryption, and checksums (these features are described in other messages). The message stores the size of a chunk instead of the size of the entire array; the storage size of the entire array can be calculated by traversing the chunk index that stores the chunk addresses."

- *"Compact: The array is stored in one contiguous block as part of this object header message."*
- *"Virtual: This is only supported for version 4 of the Data Layout message. The message stores information that is used to locate the global heap collection containing the Virtual Dataset (VDS) mapping information. The mapping associates the VDS to the source dataset elements that are stored across a collection of HDF5 files. "*

2.2.2 The SavedModel format

In the Tensorflow framework, in addition to using HDF5 as a format for storing the network model, TF employs another serialized format: SavedModel, which makes it easier to deploy models in different types of environments. This format is documented in Tensorflow (2020b).

The format is mostly a flexible method for transferring all relevant data from a model to be able to store on disk. The central part of this is the $saved\_model.pb$ file, from Tensorflow (2020b): *"The saved_model.pb file stores the actual TensorFlow program, or model, and a set of named signatures, each identifying a function that accepts tensor inputs and produces tensor outputs."*. Additionally, SavedModel also uses two directories to keep track of other information. From Tensorflow (2020b): *"The variables directory contains a standard training checkpoint"* and *"The assets directory contains files used by the TensorFlow graph, for example, text files used to initialize vocabulary tables. It is unused in this example."*.

Since the $model.pb$ is in the form of a Protocol Buffer (PB), the data for the model is stored within the file. However, it would be required to know the model's internal data structure to access the PB properly.

2.3 To identify integrity attacks

There are several techniques for finding malicious code on a system, with different pros and cons.

2.3.1 Hashing

A detector program could implement a database of hashed executables and code which is already deemed malicious, as they have been isolated and submitted as malicious by previously affected users. A hashing algorithm is used on the executable code binary, which provides the detector with an easy method of detecting malicious code from simple matching.

However, this technique is rarely used in its raw format, as circumventing this technique is quite easy because a single modification in the source code would generate an entirely different hash. Since such small changes could come from a simple change in compiler configuration or pure chance, this technique would only be useful for detecting files that are spread unchanged.

Listing 2.1: A possible malicious program

```
1 int main(){
2     corrupt_core();
3     disrupt_av()
4 }
```

Listing 2.2: A slightly different malicious program

```
1 int main(){
2     disrupt_av()
3     corrupt_core();
4 }
```

Calculating the SHA256 hashes for both pieces of code in listings 2.1 and 2.2, results in $CC90B137EE117C42DA0A936A5350ABE689D5683CCC944BB3EC19038E63EEF853$ and $E4E85A93774834C4560280031D23F23DEBE4161264424114D4C691AB20BD0269$, which does not match, and a detector having the hash for the first code, would not detect the second code as malicious, despite the codes being functionally similar.

2.3.2 Signature matching

Antivirus like ClamAV (ClamAV (2020)) utilizes methods for text-based definitions of malicious behavior in files, matching on files that have included any suspicious text strings included, like known malicious URLs. Signature matching would protect against certain types of attacks, which rely on the use of static strings, but is very limited in the range of malicious code it can detect to formats the detector can read as plain-text.

2.3.3 Name independent flow analysis

As direct hashing of files is not very reliable in detecting malicious files, another method to address such a problem. The main problem with direct hashing is linked to the ambiguous nature of code. Ambiguous code means that there are several different codebases, M - which all would have different hashes -, all of which exhibit the same behavior because of variable name changes, placement switching, and other syntactical changes. However, one method of dealing with this is to convert the code into an unambiguous representation, which can then be matched with the converted code of a malicious program, which would enable more effective detection of malicious code.

An example is presented in listings 2.3, 2.4, and 2.5

Listing 2.3: Second function for creating a tuple of average area and a list of areas based on lists of widths and heights

```
1 function calc_areas(widths, heights):
2  total = 0
3  areas = []
4  for w, h in zip(widths, heights):
5   areas.append(w*h)
6   total += w*h
7  return (total/len(width), areas)
```

Listing 2.4: Second function for creating a tuple of average area and a list of areas based on lists of widths and heights

```
1 function areas(wd, hg):
2  areas = []
3  total = 0
4  for h, w in zip(heights, widths):
5   total += w*h
6   areas.append(w*h)
7
8  return (total/len(height), areas)
```

Listing 2.5: The unambiguous code used for comparison

```
1 function f(param_list[2]):
2  vars = [0, []]
3  for i in range(0, len(param_list[0])):
4   loop_var_0 = param_list[0][i]
5   loop_var_1 = param_list[1][i]
6
7   vars[0] += loop_var_0*loop_var_1
8   vars[1].append(loop_var_0*loop_var_1)
9  return (vars[0]/len(param_list[0]), vars[1])
```

Here, the functions in 2.3 and 2.4 have functionally the same behaviour, but would not be comparable as they are. Taking inspiration from data analysis used by compilers to make determinations on the flow of data in a piece of code (Aho et al. (1986)[Chapter 9]). It is possible to create a translated code and a comparison scheme, which would make it possible to decide if two pieces of code behave the same.

Using the data flow analysis, each code block would be separated into code blocks depending on conditional statements, i.e., if, for, $while$, and after removing loop invariant expressions, and dead code the corresponding "actions" in a code block can be compared. "Actions" could, in this context, be defined as a line of code which is does something useful, e.g., assignment, increment, or multiplication. These comparisons would check if the number of specific actions is equal and that the constant values which are used are equal. The comparison could be made by making a list for each comparable actions, in the code block of the two codes, and then compare each element in the two lists to each other.

While this method would not be as quick as purely comparing the hashes of files or functions of code, it would be more useful for discovering malicious code based on the behavior of the code, instead of implementation specifics.

2.4 Integrity and authentication

When transmitting code across any non-trusted medium, there is a need for the code's recipient to verify the code's authenticity and integrity. Integrity ensures the recipient that the code has not been tampered with, and authenticity ensures a trusted party sent the code. Mechanisms that ensure integrity and authentication are presented in this section.

2.4.1 Code signing

Code signing is a security mechanism that enables parties to create a signature for a piece of data, which can be verified by another party.

The sender of the data has an asymmetric key-pair, consisting of a private- and a public key, which has a mathematical connection to each other. That way, generating the private key from the public key is very time-consuming, and in theory, requires brute-forcing the key.

The act of signing code is comprised of two parts:

Signature generation: The private key and the data to be signed is used in an algorithm, which produces the signature for this specific data and key. This

algorithm is a one-way algorithm that ensures the private key cannot be generated from knowing the data and the signature.

Signature verification: The public key, the data, and the signature are used in an algorithm, which outputs a boolean value indicating if the signature belongs to the data and produced by the private key, which is connected to the public key.

Using code signing, integrity is maintained. The verification algorithm checks if the data corresponds to the signature, and authentication is maintained since the verification algorithm checks if the public key (and by extension, the private key) belongs to the signature.

Additionally, authenticity is often also implemented on the public key to ensure it belongs to whomever it claims to belong. Authenticity is ensured employing physical accessibility (putting a public key in a system manually, ensuring it is the corresponding key), or using Certification Authorities, which can provide a signature of their own on other public keys.

Below is a private RSA key used to generate the signature on a piece of code, e.g., the code in listing 2.6.

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDGBkG7VF2JBRDJHcA+a0FbB+ye+a7nhdJNblJ/xQK
jrRd479zsDi3r+bTVxZsMTKCsXbTL6JNV/Cy56pNeTF9UgQuZmH+3Sk
YEODLEMk1iGs4KmTr2GMWUMwvL3tDVfanl/ovvkCs/7W8P5Q0/BTXpE
92EHALzWxy6OJpfsZfI+QIDAQABAoGAGuRWoN4AJc9uTCGiLdfa9EXL
OHun6QEfYiyNP5S9mwSyn1XTpqr2SYmLqUxAlhcB6uZ+2wImoBKy8Dy
O7stoVpBV0lqDgc8b1AlJZU0My05fMK9jK0B8Qc3fJ3rzdsirXXFqbi
0mHHgBWr9YKss6waFz6qae1doBvp1eg4zF6UECQQDlZUEH8ghhMzh4f
nsxsy0TFg6ALKG+TOjwdsRVIa71VmOVLOPimkKnGc3iaesBdPM0fSWb
1WRS91vvPF4yrMcNAkEA3P2YpCyufXLkdC+qf+jHwDJugS+esIwppUX
U97VQS0jdDazV2AOJ7XUMmqiPK7+rik4v8ez/pcjYYv4FxfsOnQJBAL
hWPYPdAs7ZEjABs42kpjwIJW7qbq81rppNVkfy4V1VJoDhqd6Wh6kwG
zouUyLAkA9F4crwF8Zz7/S3VhryrKECQQC+sXZN9OB9D+9i8t7FkTEN
AHeqslTVM52cKC4lsiefTziw3DuLM0KJzT8bFei1b6euqNlmgYP4ot9
WTKzesk1hAkAsEonlfAgbcroccU2eaEjOxVWnwoXtzlCN9eB8dBsB8M
fLAO0NRRMYVaWxec45XYF7HzosBZcBpOPvO98dGwn/
-----END RSA PRIVATE KEY-----

-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDGBkG7VF2JBRDJHcA
+a0FbB+ye+a7nhdJNblJ/xQKjrRd479zsDi3r+bTVxZsMTKCsXbTL6J
NV/Cy56pNeTF9UgQuZmH+3SkYEODLEMk1iGs4KmTr2GMWUMwvL3tDVf
anl/ovvkCs/7W8P5Q0/BTXpE92EHALzWxy6OJpfsZfI+QIDAQAB
```

```
-----END PUBLIC KEY-----
```

Listing 2.6: Example code to be signed

```
1 int main(){
2     m = load_model(path);
3     predict(m, input);
4 }
```

Using OpenSSL's dgst functionality, the code is signed using the aforementioned private key, resulting in the RSA-SHA256 hexadecimal representation below, which can be verified using the public key above.

```
openssl dgst -hex -sign private_key example_signing_code

RSA-SHA256(example_signing_code)=52208157b536e5de48ee73
120187e0624904b925a69c42a4f528ff32ee32dcf19da614d52cb27
00d6cc639b797763848b37cd45799efdcd5f5a58b5615fe53e61447
59e0fcbd4fe6662c9897b5c1d7fb41dd0732b59ba5ca11c24d7ee4f
65b6f55af46fa560e971f8219e61ce67e735d0e723b10eec930d886
dca5e12b2312ce
```

2.4.2 Encryption

The act of encrypting data means using a key, or pair of keys, to generate a piece of jumbled data, which in turn can be turned into the original data by using the key, or keys.

Methods for encrypting data are abundant, but can be categorized into two parts: Asymmetric - using a pair of asymmetric keys to encrypt and decrypt - and symmetric - using a single key to perform both encryption and decryption.

Encryption: The data to be encrypted and the encrypting key is used by an algorithm to produce a bit-string of seemingly random 1s and 0s, which is the encrypted data.

Decryption: The encrypted string and the decrypting key is used by an algorithm to transform the string into the original data.

Both integrity and authenticity are maintained, as the data is impossible to modify without having the appropriate key(s) and provides authenticity.

Chapter 3

Related Work

3.1 Malicious use of AI

3.1.1 Social bots

Out of any single area, one where AI technologies have been utilized very successfully is creating social bots. These are created to target humans, with the end goal to trick, fool or otherwise expose a person, in order to exploit them.

The research by Seymour and Tully (2016) outlines an AI bot that uses Recurrent Neural Networks (RNN) to generate content on Twitter to perform spear-phishing attacks on targeted users. The RNN is trained on the content from the users themselves, allowing for generated ”tweets” to automatically have taken the same form and language of the target user, increasing the chance of user interaction with the attacking profile.

Yao et al. (2017) provides research on using RRNs to generate false reviews online, while avoiding detection by statistical detectors, meaning state-of-the-art detection technology was unable to distinguish between the generated content and user-generated content. The generation is accomplished by a mixture of RNN and a word replacement strategy. The RNN is trained on the target domain, e.g., restaurants, clothing stores, bars, determining the language, and the specific text generated. The word replacement strategy then recognizes certain contextual words, like nouns, which are then replaced with other, similar words. In turn, this enables the attacker to control the sentiment of each review and could be used to skew the impression real people have of the reviewed object.

3.1.2 AI camouflage

Other methods of cyber attacks incorporating AI are methods aimed at hiding the attacker's intent by utilizing AI techniques to learn how to avoid detection.

The paper by Bahnsen et al. (2018) outlines a method for generating phishing URLs able to circumvent AI-powered phishing detection systems, by training an LSTM, which can generate less detectable URLs, based on a training set of successful phishing URLs.

3.1.3 DeepLocker

The research by Kirat et al. (2018) shows a concept for designing malicious AI unlocked by fulfilling specific input criteria. The attack is accomplished by training a NN on specified input parameters, like images from a face camera, or voice. The input is used in the NN, outputting a bit-string, which is used as the decryption key for a payload, in turn executing the decrypted payload. By encrypting the payload, and hiding the decryption key in a trained AI, the attacker can avoid detection methods against the payload, and avoid reverse-engineering the decryption key, as the target attributes are unknown until the AI successfully decrypts the payload.

3.2 Attacks targeting AI-based systems

In Barreno et al. (2010), one of the research results is a taxonomy framework for classifying attacks against machine learning systems. This taxonomy relies on classifications of attack according to three distinct dimensions:

- **Influence:** Differentiating between how the attack is performed on the AI system and is defined as one of two areas. First is *Causative*, where the attack is performed by influencing the training. Second is *Exploratory*, where the attack exploits misclassifications without influencing the training of the data.
- **Security violation:** Differentiates between what the attack affects, and is defined as one of two areas. First is *Integrity*, where input is influenced by false positives, i.e., a misclassification which allows something to happen which should not happen for this input. Second is *Availability*, where input is influenced by false negatives, i.e., a misclassification which does not allow something to happen when it should be allowed to happen.

- **Specificity:** What is the scope of the attack concerning the input, and is defined as one of two areas. First is *Targeted*, where the AI is affected for a small number of instances. Second is *Indiscriminate*, where the AI is affected for a large number of instances.

In defining these categories for attacks against AI, the research also outlines defensive measures against the two categories relating to the methods used for attacking:

- **Causative:**
  - RONI: Reject On Negative Impact, which measures the impact of a training instance on the AI model, rejects the training instances that have a large negative impact on the accuracy of the model.
  - Robustness: Find procedures that are the least susceptible to manipulation of the chosen training data.
  - Online prediction with experts: Create a composite AI model, which is trained to follow advice from a set of expert systems. Each AI model gives its own, separately trained, predictions on the data, and the composite classifier is trained according to how it follows the advice of the most successful expert.
- **Exploratory:**
  - Training data: The attacker is limited to knowing the training data used by the AI system
  - Feature selection: Transforming the raw measurement data into a feature map, which is used as the input for the AI, instead of the raw data.
  - Hypothesis space/learning procedures: Making it more difficult for the attacker to know specifically what the AI model is trained to do.
  - Randomization: Randomize the hypothesis to real output values in [0,1], instead of 0 or 1. The randomness would increase the cost for the attacker to gain information on the AI system.
  - Limiting/misleading feedback: Eliminate, channels of information, or use these channels to provide the attacker with misleading information.

3.2.1 Attacks fooling AI-based systems using adversarial input

In the field of security and AI, much research has gone into exploring the use of altered information in order to circumvent detection by an AI system, or otherwise disrupt the regular operations of the AI system. The attack is made by crafting

the input values being used in the AI system, which in turn classifies the input as something different than it should be.

The works of (Chakraborty et al. (2018)), (Zhang et al. (2020)), (Akhtar and Mian (2018)), (Yuan et al. (2019)), (Sun et al. (2018)), and (Ozdag (2018)) outlines surveys presenting information on which types of AI is susceptible to adversarial examples. Here, some proposed solutions for handling adversarial examples are:

- **Adversarial training:** Injecting adversarial examples into the training data makes the training more robust against adversarial attacks.
- **Distillation:** Where a second neural network is trained on the first's outputs, in addition to using temperature variables to reduce the sensitivity of perturbations in the input data.
- **Feature squeezing:** Reducing the complexity of several inputs by "squeezing" input values into single values, like smoothing filters on images-.
- **Transferability blocking:** A method in which NULL labeling is used to classify instances which are adversarial, by training with adversarial examples of the training data, modified to different degrees of perturbation.
- **Defence GAN:** By training a Generative Adversarial Network, where one part is set to discriminate between real and perturbed input, and the other is the real model. By this method, the real model is trained to differentiate between real and perturbed input.
- **MagNet:** A classifier read the output of the NN and rejects the output if it is too distant from the selected "normal" set. Additionally, it uses autoencoders to revert adversarial examples into normal input, but only in a black box scenario.

Additionally, in (Clark et al. (2018)) and (Sharif et al. (2016)), experimental results show how adversarial examples can be performed in physical systems, by sending ultrasound signals at planned locations, and by wearing a specially designed pair of eyeglasses, showing that adversarial examples can be crafted in physical systems as well.

Real-world threats using adversarial examples are also presented in Neekhara et al. (2020). DeepFake videos, which are videos where a person's face can be "projected" onto the movements of a different person, making it possible to make a video of any person "saying" practically anything, has been altered. By altering

the input, DeepFake videos can avoid detection by DeepFake detectors.

Lastly, the works of Rajpal et al. (2017) show methods for training an AI to be able to ”fuzz” input data to be used as adversarial examples, making for an efficient method for generating a large amount of adversarial example data. The fuzzing could help make the examples above much more efficient at producing adversarial examples.

3.2.2 Targeted attack against AI systems

While section 3.2.1 goes into detail about how AI systems can be attacked indirectly by altering input values, there exist other exploits which focus more on compromising other aspects of an AI system.

In Stevens et al. (2017), the approach is based on using valid inputs for an AI to exploit execution bugs, which induces faulty behavior in the AI system. The attack is made in a gray-box fashion, as the exploit requires knowledge about the software that is used. The consequence of this attack is the possibility for an entire AI system to be poisoned by its input, influencing the AIs behavior during execution of the AI program.

In Gu et al. (2017), the research is centered around training legitimate AI structures in cloud systems and poisoning the training data to alter the target behavior slightly. The alteration means the attackers can manipulate an AI system’s behavior, even to the degree that re-training the model still causes decreased accuracy on the target input.

In Brendel et al. (2017), the research focuses on altering the output state after the AI core has finished computation. This method is performed by gradually traversing a specific classifications’ input-space-boundary, meaning the edge of the ”geometrical” shape, in which every point inside would output one specific classification. The method is then able to gradually traverse this boundary until the input is a separate classification. Still, it classifies as the original, since the point lies within the classification range.

Chapter 4

Research motivation

4.1 Motivation

4.1.1 The AI and security landscape

In the field of cybersecurity, the use of AI technologies as preventative measures has gradually gained a foothold, as AI-powered classifiers are trained to detect, monitor, and mitigate a variety of cyberattack vectors. The use ranges from scam-filters in email clients, bot-detection on social media, to Denial of Service (DoS) detection in network structures.

The flipside to this coin is the use of AI directly in a cyber attack. While any large-scale cyberattacks using AI directly so far is lacking, research into how AI can be used for malicious purposes continually reveals the scope of which AI could be used in attacks. The most notable of which might be the DeepFake AI system (Neekhara et al. (2020)), able to imitate people in a video, even being able to alter the image enough to fool AI-powered detectors.

Lastly, AI has gained extensive use in applications and technologies. The car industry, for instance, predicts a significant increase in the production of self-driving cars, utilizing AI technologies for sign detection, avoiding pedestrians, traffic control, and planning. Other applications in everyday life also make use of AI technologies. YouTube, the most popular video viewing and sharing web-site, while maintaining secrecy on its internal algorithms, utilizes AI technologies. From optimized video serving, ad serving, to their infamous Content ID system, much of YouTube's systems rely on AI technologies.

As the use of AI in everyday life increases, so does the motivation for mali-

cious actors to be able to circumvent these AI systems. Research into adversarial examples has shown that slightly altered input to an AI system is often able to fool classifications, even in physical systems completely. These attacks pose a severe challenge to the use of AI systems in real life, as such misclassifications would be worrying to anyone who wanted to use such systems.

While many parts of using AI maliciously has been covered by previous research, when it comes to compromising the integrity of the code in an AI system, there has been limited research, and mainly in the context of the *causative* attack described in Barreno et al. (2010), of which Gu et al. (2017) provides a working example. Barreno et al. (2010) also references *exploratory* attacks, in chapter 3 referred to as *adversarial examples*. Here the concept is to exploit the flaws in an AI system in order to misclassify input.

However, there exists a classification of attacks against AI systems that do not fit neatly into the classifications defined in Barreno et al. (2010). This attack modifies the behavior of an AI system by altering the AI model to provide additional functionality while maintaining the original functionality of the AI system. This attack will, in this thesis, be referred to as an integrity attack against an AI system.

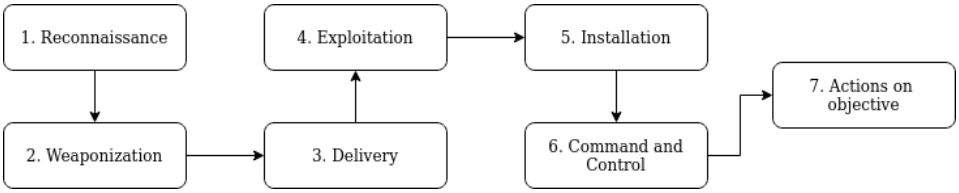
4.1.2 The context of an integrity attack against AI systems

The DeepLocker attack

The DeepLocker attack, introduced in Kirat et al. (2018), is an AI attack using a neural network as part of a decryption mechanism for an encrypted payload containing an attack. The concept works by training a neural network on a set of input parameters, e.g., an image, or voice of a person. The output parameters are then used as the symmetric key in an encryption algorithm, which makes it so the decryption can only be performed if the neural network recognizes whatever it was trained to recognize. An attacker payload - which can consist of any attack - is then encrypted with the key. The network, execution code and encrypted payload are then used in the attack on a system, installing and executing the execution code when access to a system has been successful.

The reason DeepLocker is so appealing in the context of the attack against AI systems is the concept of being able to modify an existing AI system, as described earlier, and implement a DeepLocker NN in a running AI system. Because the AI system is supposed to execute in the system, this could be used to hide the attack. The attack would then wait in the AI system until the prerequisite conditions for the DeepLocker NN are met. As an example, consider this attack being successful

Figure 4.1 The Cyber Kill Chain steps



in infiltrating an AI-powered self-driving car. This attack could, depending on the attacker's goal, cause loss of human life or cause public distrust in AI-powered self-driving cars.

Anatomy of DeepLocker using Cyber Kill Chain

The Cyber Kill Chain (CKC) is one method of dividing a cyber attack into parts, each having its scope, methods, requirements, and most importantly, defensive measures. These parts are seen in figure 4.1. By putting the integrity attack against AI systems into the context of the CKC, the scope of the research is much more clearly defined, as certain parts of the attack do not have to be specified in detail. Assumptions can then be made on those parts which are not covered, without this impacting the findings of the research, as the findings are put into the context of the set CKC model.

In the described integrity attack against an AI system, steps 1, 2, and 4 must be considered for the attack to succeed.

In step 1 - *Reconnaissance* - the attacker would gather information on the target, on how the AI model functions, the feature map of the system, and the output behavior.

In step 2 - *Weaponization* - the attacker would train a DeepLocker neural network following how the AI system functions, and encrypt the attacker payload.

Step 3 - *Delivery* - will be assumed to be possible and is not part of the scope of this thesis.

Lastly, step 4 - *Exploitation* - the DeepLocker neural network is installed into the existing AI system, and the payload is stored to be used whenever the DeepLocker is activated.

As the encryption of the payload in DeepLocker allows for pretty much any type of attack to be performed once decrypted, steps 5-7 are not considered in this thesis.

4.1.3 Practical implications

With the advent of the wide-spread use of AI technologies in real-life applications, the danger of a malicious actor being able to directly target the AI system could pose severe risks in the well-being of anyone using such technologies. As research covering the definitions of attacks against AI system does not accurately cover the mentioned classification of an integrity attack on AI systems, it is necessary to explore the capabilities of this type of attack. For this reason, this research exists; to explore such methods and develop and test strategies and methods to mitigate the dangers of this exploit.

Therefore, the goal of this research is to explore a concept where an AI system is targeted by a malicious actor to describe prerequisites and assumptions needed for the attack to be successful and to outline mitigation strategies to protect the AI system from such an attack. This thesis's primary focus will be the weaponization step of the attack, in figuring out a specific method which can compromise the integrity of the AI system.

4.2 Research questions

- **RQ1:** How is it possible to compromise a neural network to add functionality to an existing network, without the normal execution of the neural network being impacted, in a white-box fashion?
- **RQ2:** How is it possible to compromise a neural network in order to add functionality to an existing network, without the normal execution of the neural network being impacted, in a black-box/gray-box fashion?
- **RQ3:** How is it possible to defend against the attacks explained in RQ1 and RQ2?

4.3 How the thesis will answer the research questions

As the research relies on an existing AI system to attack, a simple AI system will be crafted for use in the research. This system will receive training within the scope of image classification, specifically it will be trained to recognize if a person wears glasses and if the person has a beard. This model does not, however, require any particular level of accuracy, as the requirements from RQ1 and RQ2 do not consider the effectiveness of the AI system, only the relative change in functionality pre- and post the integrity attack.

4.3.1 RQ1:

In answering the question of how an attacker would be able to perform an AI integrity attack in a white-box fashion, the research will be aimed at producing a theoretical model and a practical implementation of a proof-of-concept. This concept will be produced with the motivation to compromise an existing AI system's integrity when the specifics of the targeted AI system are known.

In the thesis, the theoretical model is a general model for how an integrity attack against AI systems can be performed, focused on the method of being able to incorporate the attacker network into the original network while maintaining the functionality of the original network. As the model information is available from the white-box approach, the theoretical model will make any assumptions following what is known about the AI system beforehand.

For the practical implementation of the theoretical model, the approach is divided into three separate parts: *Targeted AI System*, *Method for Modifying the AI System*, and *Verification methods for the Original, Attacker and Modified networks*. The *Targeted AI System* is described above.

The *Method for Modifying the AI System* is further split into several parts: *Accessing data*, *Modification method*, and *Storing the data*. As the approach is white-box, accessing and storing the data is possible using, however, format the data is stored in, based on how the implemented AI system stores its data. The modification method is also performed as an ad-hoc implementation depending format of the data. However, the modification method would be possible to do as a general method, which would require additional steps to translate the data to and from the format the general method uses.

The verification methods will check the relative difference between the original, attacker, and modified network, using test data in the original network's scope. The verification is done to confirm that the theoretical model's conclusions correspond with the practical implementation.

4.3.2 RQ2:

In answering the question of how an attacker would be able to perform an AI integrity attack in a black-box/gray-box fashion, the research will be aimed at producing a theoretical model, as well as a practical implementation of a proof-of-concept. This concept will be produced with the motivation to be able to compromise an existing AI system's integrity when the specifics of the targeted AI system is not known, or the knowledge is limited.

The theoretical model for RQ2 is nearly identical to the model in RQ1. The

difference from RQ1 is that any assumptions relying on knowledge of the AI system are limited to the input to the AI system.

The practical implementation is structured the same as in RQ1, with specific changes to the implementation for the *method for modifying the AI system*.

The *Accessing data* part in RQ1 relies on knowing the data format for storing the AI system. In the black-box/grey-box fashion, the research will explore two methods for a black-box method for accessing the data. The first is binary access to the AI model data. Second is enumerated access to data using a list of the most commonly used data formats. The enumeration will be limited to 2 specific formats, HDF5 and SavedModel, for the implemented solution.

4.3.3 RQ3:

In answering the question of how to defend against an AI integrity attack, the research will also be aimed at producing a theoretical strategy, and techniques which can be implemented, with the goal in mind to be able to either prevent and detect an AI integrity attack.

The strategies and techniques will be developed based on knowledge and experience gained through answering RQ1 and RQ2 and general knowledge on how integrity and authentication can be implemented in working systems.

4.4 By which metrics the answers for the research questions will be evaluated

In chapter 6, the results from the research will be evaluated against the metrics laid out below for each research question.

4.4.1 RQ1:

- Whether the research can produce a proof-of-concept, both in principle and in practice using a real-world practical example.
- The degree to which the research can discover noticeable behavior or traits of the attack.

4.4.2 RQ2:

- Whether the research can produce a proof-of-concept, both in principle, as well as in practice using a real-world practical example, without, or with limited knowledge on the AI system targeted.

- The degree to which the research can discover noticeable behavior or traits of the attack.

4.4.3 RQ3:

- The degree to which the proposed mitigation strategies and techniques would theoretically prevent an AI integrity attack.
- The degree to which the proposed mitigations are considered as a practical implementation in a working AI system.

Chapter 5

Results of research questions

5.1 The targeted AI system

This AI system will be the basis for the attacks performed in RQ1 and RQ2 and is based on an image classifier. The image classifier's purpose is to be able to recognize if people in an image wear glasses and have a beard. The purpose is based on the metadata for the data set provided as the training data, the FERET image database.

5.1.1 Setup

The training was implemented in Python and used the Tensorflow library to train the models and create the model network files. In 5.1, the different versions for libraries, languages, and formats are listed.

System name	Description	Version
Jupyter	IDE for easy Python execution	4.6.3
Python	Programming language	3.6.9
Tensorflow	Python library for machine learning	2.2.0
H5PY	Python library for using HDF5 files	2.10.0
HDF5	File format for storing large datasets	1.10.4
NumPy	Python library for scientific array operations	1.18.2

Table 5.1: System versions and descriptions

5.1.2 Experimental data

The experimental data was gathered from the FERET Color Database (NIST (2019)), providing a data set of images of 739 people's faces, combined numbering 8172 unique images. This data set was chosen for two reasons. First, is the extensive metadata provided, which makes the labeling of instances very easy. Second, the relatively small size would not impact the AI system in the context of the thesis since it did not require the AI system to have a high degree of accuracy.

The attached metadata for the images are listed in 5.2

Field	Description	Value Type	Example
Recording:id	ID for one specific image	String	cfrR00001
URL	The relative image path for the image	URL	data/images/ 00001/00001_930831 _hl_a.ppm.bz2
CaptureDate	When the photo was taken	Date	08/31/1993
CaptureTime	When the photo was taken on the date	Time	00:00:00
Subject:id	ID for one specific person	String	cfrS00001
Pose:name	Classification of the pose of the face (TODO write list of these)	String	hl
Pose:yaw	Numeric value of the degrees the head is turned around the neck axis	Float	67.5
Pose:pitch	Numeric value of the degrees the head is turned around the ears axis	Float	17.0
Pose:roll	Numeric value of the degrees the head is turned around the nose axis	Float	10.5
Wearing:glasses	Boolean value for whether the subject has glasses on	Bool	No

Hair:beard	Boolean value for whether the subject has a beard	Bool	Yes
Hair:mustache	Boolean value for whether the subject has a mustache	Bool	No
Expression:name	Classification for what the expression of the subject is	String	fb
LeftEye:x	Numeric value for which pixel in the x plane the left eye is located at in the image	Integer	328
LeftEye:y	Numeric value for which pixel in the y plane the left eye is located at in the image	Integer	324
RightEye:x	Numeric value for which pixel in the x plane the right eye is located at in the image	Integer	204
RightEye:y	Numeric value for which pixel in the y plane the right eye is located at in the image	Integer	328
Nose:x	Numeric value for which pixel in the x plane the nose is located at in the image	Integer	270
Nose:y	Numeric value for which pixel in the y plane the nose is located at in the image	Integer	392
Mouth:x	Numeric value for which pixel in the x plane the mouth is located at in the image	Integer	272

Mouth:y	Numeric value for which pixel in the y plane the mouth is located at in the image	Integer	460
Weather:condition	Classification for the weather present in the image	String	Inside

Table 5.2: FERET metadata descriptions, types and examples

Listing 5.1: The corresponding metadata XML to figure 5.1

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE Recordings SYSTEM "http://hbase.humanid.org/hbase/dtd/
       recording.dtd">
3  <Recordings>
4  <Recording id="cfrR00002">
5    <URL root="Disc1" relative="data/images/00001/00001_930831_fa_a.
         ppm.bz2"/>
6    <CaptureDate>08/31/1993</CaptureDate>
7    <CaptureTime>00:00:00</CaptureTime>
8    <Format value="ppm" scanning="Progressive" compression="bzip2"/>
9    <Subject id="cfrS00001">
10    <Application>
11     <Face>
12      <Pose name="fa" yaw="0" pitch="0" roll="0"/>
13      <Wearing glasses="Yes"/>
14      <Hair beard="No" mustache="No" source="Retrospectively"/>
15      <Expression name="fa"/>
16      <LeftEye x="326" y="332"/>
17      <RightEye x="202" y="334"/>
18      <Nose x="268" y="404"/>
19      <Mouth x="266" y="468"/>
20     </Face>
21    </Application>
22    <Stage id="cfrT00001"/>
23   </Subject>
24   <Collection id="cfrC00001"/>
25   <Environment id="cfrE00001"/>
26   <Sensor id="cfrN00002"/>
27   <Illuminant id="cfrI00001"/>
28   <Illuminant id="cfrI00002"/>
29   <Illuminant id="cfrI00003"/>
30   <Weather condition="Inside"/>
31  </Recording>
32 </Recordings>
```

Figure 5.1 One example image from the FERET dataset

Figure 5.1 shows a single image from the FERET dataset, with its corresponding XML metadata instance shown in listing 5.1.

5.1.3 Implementation

As the purpose of this AI system is just to create a working AI system that will be attacked, the implemented solution is based on a simple neural network tutorial from Tensorflow (Tensorflow (2020a)).

Listing 5.2: The AI system training pseudocode

```
1  initialize;
2  data <- load_data(training_data_path);
3  for d in data:
4      c_d <- convert_data_to_size(d, width=256, height=256);
5      images.append(c_d);
6      l <- get_metadata(d);
7      labels.append(l);
8
9  split_images(images, train_data, test_data, train_percent=0.8);
10 split_labels(labels, train_labels, test_labels, train_percent=0.8)
       ;
11 model <- setup_model([3x256x256, 128, 2], [sigmoid, sigmoid], loss
       =binary_crossentropy);
12 train(model, train_data, train_labels, epochs=2);
13 validate(model, test_data, test_labels)
14 save_model(model);
```

In listing 5.2, the general overview of the implemented code in listing A.3 in the Appendix is shown. The *load_data* function fetches the raw training data and metadata from disk, which is then used by the *convert_data_to_size* function - which converts the image into the input size, as well as normalizing the pixel values, transforming the R, G, and B values as $[0, 255] -> [0, 1]$. The *get_metadata* function converts the metadata into a list of 0s and 1s. This metadata is used to denote if an image belongs to a specific classification. The converted image and corresponding label are then put into separate lists, which are then split up into training and testing. The neural network model is then set up, with 3 layers, the input layer of size $3 \times 256 \times 256$ corresponding to $width = 256$, $height = 256$ and the 3 RGB values for each pixel. The second layer is of size 128, with the activation function being the sigmoid function. The output layer is of size two because of the two separate classifications the model should recognize. The activation function is the sigmoid function since the prediction should be in how confident the model is that the image is a specific classification. The chosen loss function is $binary\_crossentropy$, which calculates the error for the individual classification and sums up the total loss value errors. Lastly, the model is validated against the

test data, and the trained model is saved.

5.2 RQ1 - White-box Integrity attack against AI systems

5.2.1 Concept

Accessing data

In RQ1, we assume the attacker knows exactly how the targeted system stores the data. This assumption enables the attacker to use the same method of accessing the AI model's data as the targeted system.

Merging AI

At the core of the AI integrity attack lies a concept of modification and expansion of an existing neural network to facilitate malicious behavior. The idea is to modify an existing neural network in order to:

- Train the behavior of a network to the specifications of the attacker
- Add the attacker behavior to the existing neural network
- Preserve original network behavior

These restrictions are motivated by examples of using neural networks in determining behavior in applications, like object detection in a self-driving car. If a malicious actor can

- Train behavior on a specific input, like objects being in front of a car,
- Maintain AI behavior on regular input, and
- Have the new behavior work within the AI's existing execution, the integrity of the AI itself has been breached.

Storing data

The attacker will not change the format of the AI model data to store the modified neural network.

Validating data

The original, attacker and modified models are tested on a set of input data to check for correctness. The output values are checked to see if the behavior of the original is kept and that the attacker's neural network works as expected.

5.2.2 How to merge the two neural networks

Accessing data

As the formats used to store the data, is assumed to be available for the attacker to be able to use, the same method of accessing the data as the targeted system is used by the attacker.

Merging NN

The first step of the attack is training a neural network with a similar structure to the targeted solution.

The only restriction on the structure of the attacker's NN is the number of layers it can have. The reasoning behind such a restriction lies in the fact that 1) The original network has to maintain its functionality, and 2) The activation function for nodes is not necessarily linear.

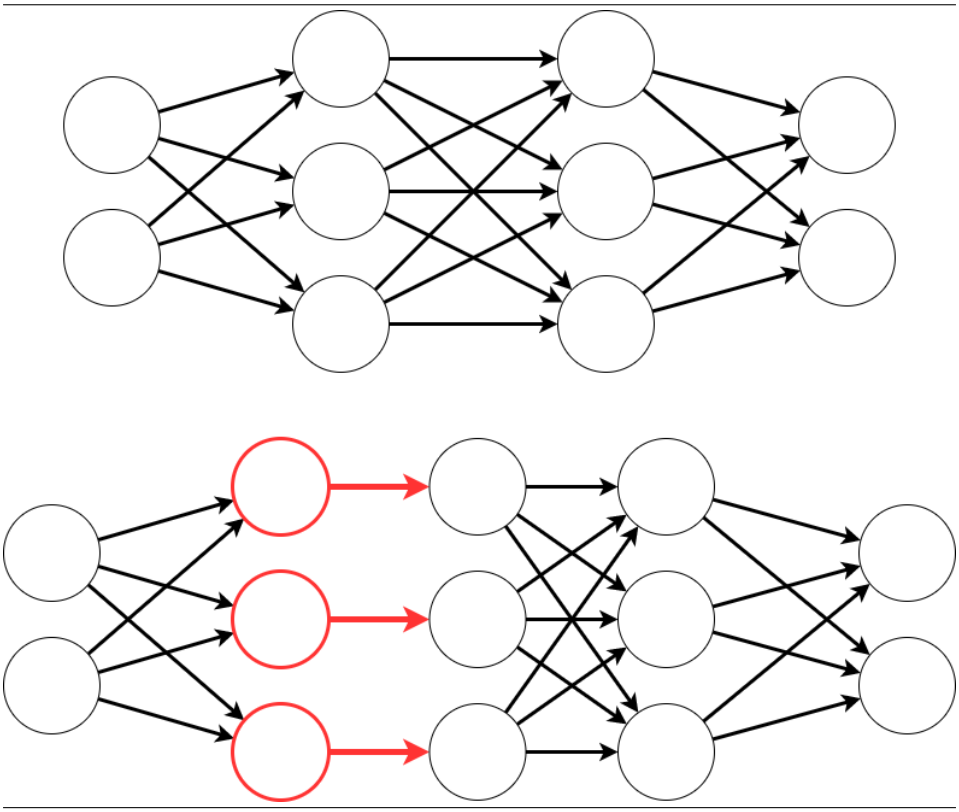
If we want to expand the network's layers and maintain its functionalities, the solution will look like the one illustrated in figure 5.2. Here, a single node is split into two nodes, with a single weight connection between them, with the previous input and output weights connected to each of the two nodes. This method relies on the mathematical formula $v_m^j = A(v_n^{j-1} \times w_{n,m}^j + b_m^j)$, with v_m^j being the value of node number m in layer j, $w_{n,m}^j$ is the weight value from node number n in layer $j-1$ to node number m in layer j, and b_m^j being the bias value for node number m in layer j.

This formula in turn, makes it necessary for having a single constant weight value $W = w_{n,m}^{j-1}$ and a single constant bias value $B = b_m^j$ to solve the equation $x = A(W * x + B)$. In the case that the activation function is linear, i.e. $A(x) = cx + d$, the equation is solvable, but in the case that $A(x) = \frac{1}{e^{-x}+1}$ - the sigmoid function - the equation does not have solutions for W and B which are constant values. This would in turn mean that the method does not work for expanding a network while maintaining the functionality, in the case that the activation function is nonlinear.

Because of this restraint for this method to not be linear - which most commonly used activation functions are - the assumption will be that it is impossible to expand the number of layers in a network without affecting the functionality. However, this conclusion does not claim that it is mathematically impossible, as there is the possibility that the solution could be solved if the network was fully connected. This hypothesis, however, was not explored.

When it comes to adding the attacker behavior and preserving the original network behavior, they go hand-in-hand, as one both relate to how the resulting

Figure 5.2 Illustration for expanding the layer number of the original network



NN structure would look like, and subsequently, how to transfer the original model to this NN structure.

Looking at how a neural network calculates the value of a node (equation 5.1), there does exist a method which would allow the behavior of two networks to exist within the same neural network structure.

$$v_m^j = A((\sum_{n=0}^{N} v_n^{j-1} * w_{n,m}) + b_m^j) \tag{5.1}$$

In the modified network, the weight and bias values from both original and attacker models are inserted into the modified matrices and vectors to separate the networks from each other. As both models rely on the same input, the method for the first matrix - which contains the weights from the input nodes to the first hidden layer - is slightly different from the other weight matrices. The modification is done as follows:

- Creating a new matrix W, with size of $m_c \times n_c$, where $m_c = m_a = m_b$ and $n_c = n_a + n_b$, with $m_a \times n_a$ and $m_b \times n_b$ being the sizes of weight matrices A and B in the original and attacker networks. Here, m_x refers to the size of the layer that the weights connect from. n_y refers to the size of the layer that the weights connect to.
- The values from matrix A and B is inserted into matrix W as described in listing 5.3
- The rest of the values are set as value 0

Listing 5.3: Inserting values into the matrix for the input

```
1 for i = 0 in m_a:
2     for j = 0 in n_a:
3         W[i][j] = A[i][j]
4 for i = 0 in m_b:
5     for j = 0 in n_b:
6         W[i][j + n_a + 1] = B[i][j]
```

This method results in a modified weight matrix looking like figure 5.3.
The method for the other weight matrices is done as follows:

- Creating a new matrix W, with size of $m_c \times n_c$, where $m_c = m_a + m_b$ and $n_c = n_a + n_b$, with $m_a \times n_a$ and $m_b \times n_b$ being the sizes of weight matrices A and B.
- The values from matrix A and B is inserted into matrix W as described in listing 5.4

Figure 5.3 Visual representation of a weight matrix, for first layer

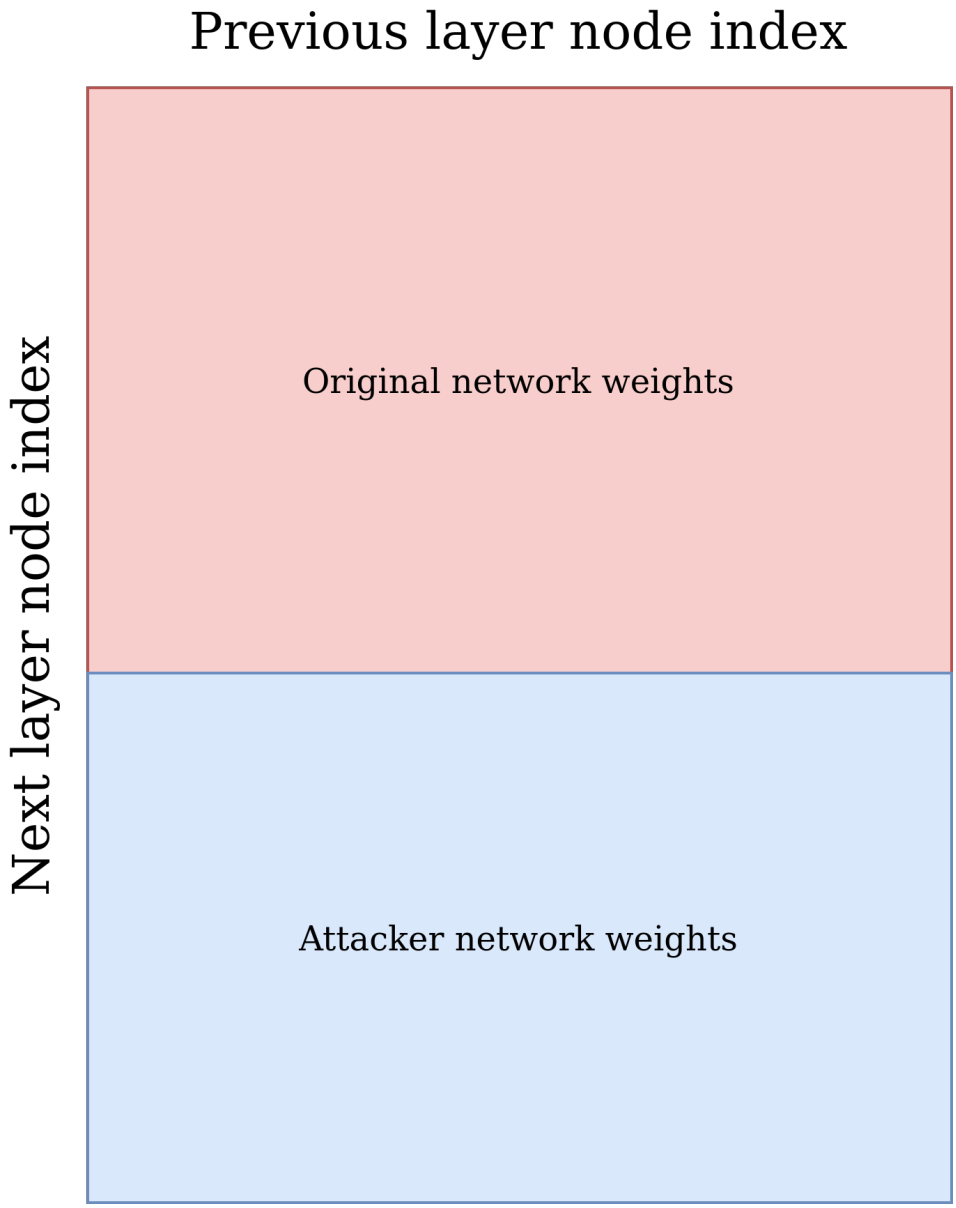
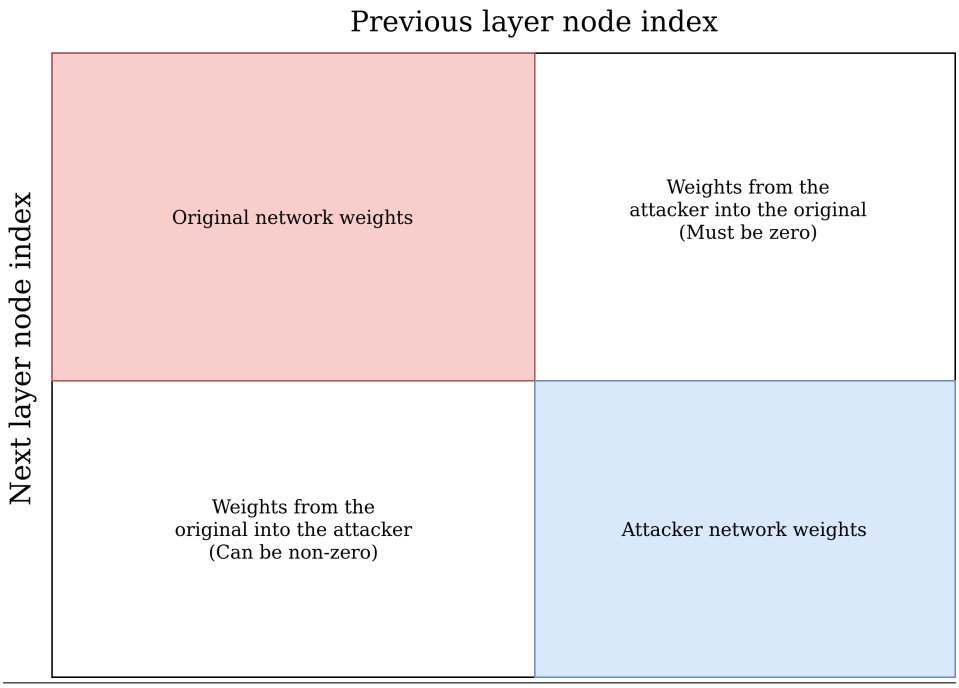


Figure 5.4 Visual representation of a weight matrix



- The rest of the value are set as value 0

Listing 5.4: Inserting values into the matrix

```
for i = 0 in m_a:
    for j = 0 in n_a:
        W[i][j] = A[i][j]
for i = 0 in m_b:
    for j = 0 in n_b:
        W[i + m_a + 1][j + n_a + 1] = B[i][j]
```

This method creates a modified weight matrix looking like figure 5.4.

The method for creating the bias vectors is much simpler, with the new bias vector having its values set by concatenating the bias vector from the attacker network to the bias vector from the original network.

Lastly, since the modified model would have additional output nodes, the executing code of the original AI system would need to be modified to use the new output values, e.g., implementing DeepLocker's decryption algorithm with the new output. The implementation would need to be done in an ad hoc fashion based on how the execution of the network is implemented.

Storing data

Since many formats require correct metadata to be able to function correctly, any metadata which would need to be changed is changed on an ad hoc basis, based on the requirements the format defines for the model to be validated. This change is possible since the attack is performed in a white-box fashion, meaning the attacker can precisely know how the metadata is stored.

Validating data

After the attacker merges the two networks, the result is a single network with sets of output, with one belonging to the original network, and the other belonging to the attacker. The task of the verification is to check that the merged network expresses the functionality from both networks.

This can be done by using the test data for the models and comparing the output nodes of the original, attacker, and merged network. Therefore, the N original output nodes should be compared to the N first nodes in the merged network, which should result in identical results. The M attacker output nodes should also be compared to the $N + 1$ to $N + M$ output nodes of the merged network.

5.2.3 Implementation of merging two neural networks

All computation was done on Ubuntu 18.04.4 LTS, running on an underlying Intel Core i7-8550U CPU, with 8x1.8GHz cores.

Step 1: Reading in the data from the original and attacker NN

In this implementation, the access of data is implemented using the HDF5 library, which is used since the white-box approach allows us to know exactly the format used by the AI system, with HDF5 being one of the popular formats which are used for storing Tensorflow models. The pseudocode in listing 5.5 shows how this was implemented, by using the HDF5 library to access the file, since the white-box approach allows for knowing the file location on disk. The HDF5 object is then looped over for each layer stored, as the HDF5 implementation stores weights and biases in separate internal dataset objects. After reading in the data from the original and attacker NN, the program flow is then passed on to the *Merging NN* method.

Listing 5.5: Pseudocode for accessing the model file from listing A.1 in the appendix

```
1 file1 = hdf5.open(original_path);
2 file2 = hdf5.open(attacker_path);
```

```
3 for layer_original, layer_attacker in zip(file1.layer_group, file2
      .layer_group):
4     #Modify weights and biases
```

Step 2: Merging the original and the attacker's NNs

In this implementation, the data is modified within the HDF5 library. Listing 5.6 shows the modification method for the weights and biases in a single layer. The modifier makes sure the weights from the input layer are appropriately sized, checked with the is_first_layer function in line 5 of the code in listing 5.6.

Based on the is_first_layer value, the width and height of the resized weight matrix are calculated. The width as the sum of the widths from the original and attacker network - unless it is from the input layer, in which case it is only the original width - and the height as the sum of the heights of the original and attacker network (see lines 5 to 11 in listing 5.6). Because of this, the resulting matrix looks like figure 5.4, or as figure 5.3 for the weights from the input. In these figures, the weights from both the original and attacker network are contained.

The biases are done similarly (shown in lines from 20 to 32 in listing 5.6), only as a 1-width matrix. The biases of the original and attacker network only have to be concatenated to each other to work correctly.

Listing 5.6: Pseudocode for modifying the model from listing A.1 in the appendix

```
1  weights_o <- layer_original.weights;
2  weights_a <- layer_attacker.weights;
3  temp_weights <- weights_o;
4  weights_o.empty_dataset();
5  if(is_first_layer(layer_original)):
6      width <- weights_o.width;
7  else:
8      width <- weights_o.width + weights_a.width;
9  height <- weights_o.height + weights_a.height;
10 weights_o.resize(width, height);
11 weights_o.fill_with(0);
12
13 for i from 0 to temp_weights.height:
14     for j from 0 to temp_weights.width:
15         weights_o[i][j] <- temp_weights[i][j];
16 for i from 0 to weights_a.height:
17     for j from 0 to weights_a.width:
18         weights_o[i + weights_o.height][j + weights_o.width] <-
               weights_a[i][j];
19
20 biases_o <- layer_original.biases;
21 biases_a <- layer_attacker.biases;
22 temp_biases <- biases_o;
```

```
23 biases_o.empty_dataset();
24 biases_o.resize(1, biases_o.height + biases_o.height);
25 biases_o.fill_with(0);
26
27 for i from 0 to temp_biases.height:
28     biases_o[i][0] <- temp_biases[i][0];
29 for i from 0 to biases_a.height:
30     biases_o[i + temp_biases.height][0] <- biases_a[i][0];
31
32 #Store the data
```

Storing data

The method for storing data within the HDF5 system is straightforward. The modifications on the file performed in the *Merging NN* method writes directly to the file when the data is modified.

In the case that the *Merging NN* method was converted to a generalized method, the conversion from the general data representation would be done in this step, where the modified data would be stored in the HDF5 format as described.

Validating data

Since it is known that the NN model uses the Tensorflow library, it can be used to confirm that the modifications made were correct. The pseudocode in 5.7 loads the original attacker, and merged model into the Tensorflow library, and checks how they predict the results of test data. The output nodes for the original network are compared to the corresponding output nodes in the merged network. The comparison is also between the corresponding output nodes in the attacker and the merged network. The difference between the nodes is then calculated across the test set, for all output nodes. The expected error is 0.

Listing 5.7: Pseudocode for the validation of the model from listing A.1

```
1  original <- tensorflow.load_model(original_path);
2  attacker <- tensorflow.load_model(attacker_path);
3  modified <- tensorflow.load_model(modified_path);
4  test_data = load_data(test_data_path);
5
6  original_results <- original.predict(test_data);
7  attacker_results <- attacker.predict(test_data);
8  modified_results <- modified.predict(test_data);
9
10 error <- 0;
11 for mod, orig in zip(modified_results, original_results):
12     for i from 0 to 2:
13         error <- error + mod[i] - orig[i];
```

```
14
15 for mod, att in zip(modified_results, attacker_results):
16     error <- error + mod[2] - att[0];
17
18 return error == 0;
```

5.3 RQ2 - Black-box/gray-box Integrity attack against AI systems

As opposed to RQ1, RQ2 requires the integrity attack against AI systems to be performed with as limited knowledge as possible, to determine to what extent the attack can perform the attack.

Firstly, three pieces of information are essential for the attack to be performed and will be considered accessible in all scenarios.

The first is the information regarding the file location of the neural network. As it is possible to search through all files on a file system to find the proper neural network, the assumption will be that the file location is accessible in a black-box scenario.

The second is the information regarding the shape of the input data to the original NN. The shape also extends to knowing specifically which input nodes are mapped to what data, e.g., node 1-100 are the pixel values of a 10×10 gray-scale picture, normalized to values in $[0, 1]$. This information is not likely to be easily found, without either looking at the executing code of the AI system, in which case it would be possible to determine the data source and its form. This information would also have to be determined before any training occurs. Another method of assuming the shape of the input is to train attacking networks with different input nodes, each used in a specific domain.

The third is the information on the execution code of the AI system. As the attack relies on being able to modify a NN to gain more output nodes than before, in order to use these nodes to provide functionality, it has to be implemented within the execution code.

In RQ2, the difference between a gray-box and black-box solution is whether the solution only requires these three pieces of information - in which case it is black-box - or if the attacker requires more specific information regarding the targeted system - in which case it is gray-box.

5.3.1 Concept

Accessing data - method 1 (black-box)

In order for the core concept to work for this RQ, the attacker would need access to the data in the NN model, and be able to insert the appropriate values into the data file correctly. The question then arises whether the attacker would need knowledge of how the data in the model is accessed and modified, or if there is a method of analyzing the raw binary file and modifying the content.

The concept here revolves around the static analysis of the file containing the NN structure's data, with as little knowledge of the file as possible. The solution would then be able to differentiate if specific data in the file is data from the model, or something else, and would scan the entire file to ”guess” wherein the file-specific weights and biases are stored. This information could then be used as a black-box method for modifying any neural network model. The only requirement the attacker is supposed to have is the location of the file, the shape of the input data, and the AI system's execution code.

Accessing data - method 2 (gray-box)

On the other side, instead of relying on manually accessing the file as a binary, method two revolves around creating a list of enumerated ways of accessing the data, based on knowledge of which file formats are most popular to use for the NN models. By trying out several access methods, while the method would not guarantee access for all formats, most would be covered. If the attacker can construct an extensive list of the most popular formats for storing NN data, the attack could be considered almost black-box, since there is no longer a requirement for the attack to know the specific format used. This method, however, would not work with formats that are not publicly known.

Merging NN

The merger method for RQ2 would be almost the same as the merger from RQ1. The method itself would be the same; however, if the merger is written mainly for specific formats, there would be a need for several implementations for the merger.

Storing data

The data storage would rely on the found format from accessing the data, as a particular implementation of either the binary method or the format enumerating method.

Verifying data

In a black-box or gray-box fashion, in the case that the NN system runs, the verifier would have to follow the same principle as the access of data, using method 1 or method 2 mentioned above. The verifier from RQ1 could still be used, i.e., to check if the original network behaves identically before and after the merge, and that the attacker network behaves as intended when merged.

5.3.2 Theoretical analysis of possible implementations to answer RQ2

Accessing data - method 1

When it comes to acquiring model data from a binary resource, the strategy is rooted in the assumption that the size of the input is already known, and that the file which is used for storing the model is known.

Based on the concept, if the only knowledge provided to the attacker is the size S_1 of the input data for the model, the data access might be able to find a data structures containing correct values of size $S_1 \times S_2$ for storing the weights, and another data structure with size S_2 for storing the biases. If the analysis results in finding these data structures, it would automatically know that the second layer is of size S_2, and this data access method can be rerun for the second layer.

This method would assume that 1) The values of the NN structure are in a readable format (not compressed, encrypted, or otherwise modified in a lossless way) and 2) The values of the NN structure are all stored in the same place, which means that there is no arbitrary spacing between the values.

One of the prerequisites for being able to access the data in a binary fashion is to be able to interpret the data values in the file directly. In the case of a NN, the weights and biases would be stored as a type of float value, which for most computers is stored as 32 bits; however, it is also able to be stored in 16 and 64 bits.

As not every combination of 1s and 0s can produce valid float values, by interpreting every 16, 32, or 64 bits in a file as the corresponding float value, the set of interpreted values would include all the weight and bias values in the model. The assumption here is that the values are readable, as mentioned above, otherwise, the set would not contain the weight and bias values in the model.

As not all values would be interpreted as valid float values, the scanner should be able to discover the data structure of size $S_1 \times S_2$ of regularly spaced float values. If this structure is found, the method is rerun, as described above. One

limiting factor here, however, is that the data structure found, would likely be too large, as the bits before or after the real data structure could be interpreted as real float values, but not belong to the real data structure. Because of this, assumption 2 mentioned above would likely not stand. The real data structure could, however, possibly be found by analyzing the values at both ends of the structure, and evaluate if the value is likely to belong in the real data structure, by using heuristics such as ranges of common values.

If, however, it is impossible to determine S_2 precisely, the attacker would not be able to access the weight and bias values stored in the file.

5.3.3 Accessing data - method 2

As mentioned, the method 2 of the attacker is to make an adapter for each popular data storage format to read the data. Thus, there is a need to enumerate as many data formats that are commonly used for NNs. Table 5.3 lists the most commonly used formats for storing NN in popular libraries used in python (Choudhury (2019)). As HDF5 and CSV are common file formats, access to the data can be done using any library supporting these formats.

For the other formats, SavedModel, and Pickle, the data is stored as a serialized object, meaning the data is stored as the model object. Access to these objects would not be straight forward, without knowing how these model objects are implemented. Because of this, and since these NN libraries used to read the data in these formats are open source, by implementing the native object from the specific library, the data would be deserialized into this model object, and the data would be accessible.

Merging NN

Assuming the black-box access to the data is possible, the only difference from the theoretical model in RQ1, is that the size of the attacker NN would not necessarily be predetermined before the attack since it is done in a black-box fashion. Because of this, the attacker model would have to be trained with different sizes of NN, where the merger would select the correct size after the original network has been accessed.

For the merging itself, the method itself would be identical to the described model in answering RQ1.

Storing data

When it comes to data storage, this would be reliant on the specific storage format for the data. If the format were accessed using a known format, the storage would

Format	NN framework	Notes
HDF5	Tensorflow	Saves the whole model within one file
SavedModel	Tensorflow	Saves the model in a directory with separate files
Pickle	PyTorch	Saves model by saving an object's serialized data
Pickle	Neurolab	Saves model by saving an object's serialized data
cPickle	FFNet	Saves model by saving an object's serialized data (implemented in C instead of Python)
Not specified (but usually Pickle)	SciKit	Not natively supported
Not specified (but usually Pickle)	Lasagne	Not natively supported
CSV	Pyrenn	Saves the model values in a CSV

Table 5.3: Popular formats used by Python NN libraries

be made following that format. Using a known format would mean that direct data access, such as HDF5 and CSV, would suffice using existing libraries to store the data. In the case of the serialized data objects, serialization would need to be performed in the same way it was stored.

In the case of binary access, storing the data would be necessary to insert the data directly into the binary, which would require expanding the file and moving any data succeeding where the data would be stored.

Validating data

To be able to use a verifier for the attack, it would be necessary to use the same *Accessing data* method and keep an enumerated list of methods that can produce results from a model from a set of training data.

For the verification step itself, it follows the same procedure as in RQ1, where the original, attacker and modified network are compared.

5.3.4 Implementation

Accessing data - method 1

The implementation of the scanner is explained in listing 5.8. The implementation was done in stages, as some assumptions made in the theoretical model might have been wrong, in which case the binary scanner solution would not work.

Listing 5.8: Pseudocode for the binary scanner from listing A in the appendix

```
1  file = open(path_to_original)
2  number_of_floats <- 0;
3  confirm_values <- [0.06004444, 0.05854571, -0.059743077];
4  confirm_index <- 0;
5  values_exist <- false;
6  for i from 0 to file.size_bytes/4:
7      value <- interpret_as_float(file[i*4], file[i*4 + 1], file[i*4
           + 2], file[i*4 + 3]);
8      if(value != NaN):
9          number_of_floats <- number_of_floats + 1;
10         if(abs(value - confirm_values[0]) < 0.000001):
11             if(confirm_index == 2):
12                 values_exist <- true;
13             else:
14                 confirm_index <- confirm_index + 1;
```

The scanner program in listing 5.8 works as follows:

Line 1 in listing 5.8 opens and reads the entire file into memory. Lines 6-14 is for loop traversing every 32 bits of the content of the file. Line 7 interprets the value as a float32 value since the values in the model are stored as Float 32. This assumption is, of course, only made in testing that the functionality is working, and the 16-bit and 64-bit interpretations would be implemented when the functionality was confirmed to work. Line 9 keeps track of the counter for how many float values there are.

When using the scanner program, several data-points are used for the results. The program registered all values within a file which can be interpreted as a float32 values, of which there were 44 069 906. With a file of size of 302 020 704 Bytes, the number of interpreted Bytes represents 58.4% of the file. From the network, the number of expected values which would be float32's should be:

$$
\begin{aligned}
Weights\ between\ layer\ 1-2 &: 256 \times 256 \times 3 \times 128 = 25165824 \\
Bias\ in\ layer\ 2 &: 128 \\
Weights\ between\ layer\ 2-3 &: 128 \times 2 = 256 \\
Bias\ in\ layer\ 3 &: 2 \\
Total &: 25166210 \\
Total\ bytes &: 100664840
\end{aligned}
$$

From the analysis above, we suspect that 57.1% of the values found to belong to the network model. Additionally, 15 998 860 - 36.3% - of the values found were zeroes, which means that many of the values are suspected not to be real float32 values.

As the number of float values found exceeded the expected number of values in the network, the next step was to confirm that the float values represented the values from the network.

To test if the interpreted values actually "see" the weight and bias values. In line 10-14, the scanner checks if the list of 3 values $confirm\_values$ are interpreted successively. This test, however, did not return positively.

Because of this, it was concluded that this method for accessing binary data would not work on any model which did not store its values in any other fashion that raw data. Based on the results, the conclusion was that the HDF5 file used for the model was set to use a compression scheme, where blocks of values would be split up, and compressed, which would result in the raw data values not being accessible using the method proposed here.

While it would be possible to make a binary scanner which takes into account such compressed clusters, it was concluded that:

- Creating such a scanner would be outside of the scope of this thesis
- Ad hoc implementation of data access based on a single data format (HDF5) would likely not work for other formats, and implementing method 2 for data access would be much simpler.

Accessing data - method 2

Method 2 is implemented by accessing model data from 2 unique formats, the HDF5 format, and the SavedModel format. The HDF5 format is implemented as described in RQ1, as we assume the attacker knows the location of the model file.

As the SavedModel format is the serialized object for the model in a Tensorflow network, it was implemented using the Tensorflow library, shown in listing 5.9 and works as follows:

The loop in line 2 goes through each format that is listed in $list\_of\_formats$. Lines 3-11 shows the specific implementation required to access the data based on the specified format, with lines 4-7 showing the access for HDF5 using the $hdf5$ library, and line 8-11 shows access for SavedModel using the $tensorflow$ library. Lastly, line 7 and 11 calls the merging function for the corresponding format. Line 3 and 12-13 are an easy way of checking that the access method is correct, as an exception is nearly sure to be raised if the file is not in the specified format. Because of this, $except$ catches the expected raised error and instead continues the program flow.

Listing 5.9: Pseudocode for access method 2 from listing A in the appendix

```
1  list_of_formats = ["hdf5", "savedmodel"];
2  for f in list_of_formats:
3      try:
4          if(f == "hdf5"):
5              o <- hdf5.open(path_to_original);
6              a <- hdf5.open(path_to_attacker_hdf5);
7              merge_as_hdf5(o, a);
8          else if(f == "savedmodel"):
9              o <- tensorflow.open_savedmodel(path_to_original);
10             a <- tensorflow.open_savedmodel(
                   path_to_attacker_savedmodel);
11             merge_as_savedmodel(o, a);
12     except:
13         continue;
```

Merging NN

As the merger method is implemented depending on the format, the HDF5 merger is the same as described in RQ1.

For the SavedModel format, as the access to the data was done using the Tensorflow library directly, the modifications on the model were also performed using the Tensorflow library, as shown in listing 5.10, and works as follows:

In the TensorFlow library, to modify a NN, it has to be done within a Tensorflow session, shown in line 1. Line 2 loops over both the original and attacker model to find each data container, i.e., the weights and biases, and assigns the container to $params_o$ and $params_a$ to be used. Line 3-8 does the merging for the weights from the input nodes. Here, as described by the theory, the modified matrix would be of size $params_o.width$, and height of size $params_o.height + params_a.height$. Line 6 initializes the new matrix with the

width and height, which is filled with zeroes. Line 7 then inserts the original values, and line 8 inserts the attacker values into this new matrix. Line 9-14 works the same as lines 3-8, with the difference being that the width is instead $params_o.width + params_a.width$, and that the offset for the insertion of the attacker values differs, following the theoretical model, explained earlier. Line 15-20 is used to modify the bias. In line 17, The height is calculated as the total height of both biases. The new bias is initialized as a 1 width matrix, filled with zeroes. Line 19 inserts the original network into the matrix, and line 20 inserts the attacker into the matrix. In line 21, the session is then used to update the original model by assigning the temporary matrix.

Listing 5.10: Pseudocode for the ac hoc merging method for the SavedModel format from listing A.1 in the appendix

```
1  with tensorflow.session() as sess:
2      for params_o, params_a in zip(original_model, attacker_model):
3          if(params_o.name == "dense/kernel:0"):
4              width <- params_o.width;
5              height <- params_o.height + params_a.height;
6              temp_matrix <- zeroes(width, height);
7              temp_matrix[0 : params_o.height - 1][0 : params_o.
                   width - 1] <- params_o;
8              temp_matrix[params_o.height : params_o.height +
                   params_a.height - 1][0 : params_a.width - 1] <-
                   params_a;
9          else if(params_o.name == "*/kernel:0"):
10             width <- params_o.width + params_a.width;
11             height <- params_o.height + params_a.height;
12             temp_matrix <- zeroes(width, height);
13             temp_matrix[0 : params_o.height - 1][0 : params_o.
                   width - 1] <- params_o;
14             temp_matrix[params_o.height : params_o.height +
                   params_a.height - 1][params_o.width : params_o.
                   width + params_a.width - 1] <- params_a;
15         else if(params_o.name == "*/bias:0""):
16             width <- 1;
17             height <- params_o.height + params_a.height;
18             temp_matrix <- zeroes(width, height);
19             temp_matrix[0 : params_o.height - 1] <- params_o;
20             temp_matrix[params_o.height : params_o.height +
                   params_a.height - 1] <- params_a;
21         sess.run(original_model[params_o.name].assign(temp_matrix))
               ;
22     extended_model.save(original_path);
```

Storing data

The storage method for the HDF5 format is performed directly when the data is altered, and only requires the program so close the connection to the program.

For the SavedModel format, the model's storage is done through the Tensorflow library, calling the $tensorflow.save\_model(model, path)$ function, as shown in line 22 of listing 5.10, which stores the model in the SavedModel format.

Validating data

The implementation of the verifier is the same as the one implemented in RQ1. The implementation of the verifier is limited to testing the correctness of the *Merging NN* method, or be implemented with an enumeration to verify the results from test data.

When testing the black-box/gray-box implementation on the AI system in both HDF5 and SavedModel format, the merging was shown to be successful. The validation method was used in both formats, both of which resulted in a total error of 0.

5.4 RQ3 - Mitigation strategy against RQ1 and RQ2

The core of the mitigation strategy is to prevent an AI integrity attack from being able to run on an AI system silently. General authentication, and accessibility policies in a system would often prevent an attacker from accessing the system in the first place. However, by following the principle of layered security, the focus of this research is to produce strategies and techniques which would be able to prevent and detect AI integrity attack, with the assumption that the attacker can access the AI system.

5.4.1 Possible strategies to ensure AI code integrity

As a general concept, by making sure that the code being run, i.e., the AI executing and network model code, is authenticated and has its integrity ensured, before being allowed to run on the system, the attacks in RQ1 and RQ2 would be hindered.

Internally secured authenticator

The code is used to ensure authentication is trustworthy and secured against threats able to modify files on the internal system. Because of this, the authenticator code should either be implemented in hardware or secured by a rigorous OS policy,

ensuring write access to the code is not allowed. Additionally, the execution of the authenticator code must be a function of the OS, meaning the OS is delegated control to perform the authentication and running the authenticated code. This restriction also means that the AI system should not be executable by anything other than the OS, as an attacker could just circumvent any authentication by executing the AI program itself.

Authentication method

To ensure that the code is run, both execution and model data must be protected by techniques providing both integrity and authenticity. Integrity is to ensure that the provided data is not able to be modified. Authentication is to ensure that the sender of the data is trustworthy.

Both encryption and code signing techniques would provide both authentication and integrity, while a technique such as a checksum would be able only to provide integrity. As encryption and code signing techniques require some sort of key management, and implementation of this authentication would require the secured authenticator to maintain the key used for decryption/verification, to ensure a potential attacker is not able to modify the key.

Requiring the system to continuously perform authentication on the code as it is in operation is possibly not feasible because of the overhead caused by the authentication. Thus, the policy could either be method 1: Run authentication only when the AI model is loaded into memory and executed, or method 2: Periodically run authentication checks on the AI model and execution code.

While method 1 provides almost no overhead in the AI system's execution, it would also be vulnerable to integrity attacks aimed at modifying the AI model in memory. This exploit, however, has not been explored, but in theory, could be possible.

Method 2, on the other hand, would require an overhead, depending on the interval chosen for the regular authentication. While longer intervals would decrease the overhead, it would lower the system's security, for the same reason as for method 1, where an integrity attack could be constructed to target the AI model in memory.

5.4.2 Operation analysis of the AI system

While altogether avoiding an attack from affecting the internal system is the ideal situation, it is also useful to consider methods of discovering an attack while it is performing, in order to prevent further damages being done.

The idea of run-time analysis of code is nothing new, and in the instance of the attack we explored in RQ1 and RQ2, it might be more straightforward than others, as this solution does not depend on complex logic.

The proposed solution lies in implementing a simple measurement of the normal running state of a program, i.e., how the executing program uses the system's resources, mainly the use of processing power and RAM. If the AI integrity attack modifies the NN model, to make it bigger, both processing power and RAM usage will increase. In cases of more extensive modifications, it would be measurably bigger. This increase in processing and RAM usage would be monitored, with a threshold for when the system is shut down due to erratic behavior.

Additionally, such monitoring would also need to be protected in the same manner as the authenticator mentioned earlier.

A simple example of such an analyzer is implemented in the verifier for RQ1 and RQ2. It measures the average execution time for the classification function for each model. It also uses two different test image sets, the first being the test images, the original model used for testing, and the second being the test images the attacker model used for testing. This classification was performed 100 times to get an average measurement.

The code in listing 5.11 works as follows:

In line 4, the test images from the original network's training are loaded, and line 5 shows the test images from the attacker network's training being loaded. In line 9, each of the models is chosen in turn, with line 10 selecting one of the test image sets. Line 11-15 then calculates the average time spent by the model to classify the chosen images, with line 15 storing the result of the total time spent classifying for each combination of model and image set.

Listing 5.11: Pseudocode for the operational analyzer from listing A in the appendix

```
1  models <- [original_model, attacker_model, modified_model];
2  iterations <- 100;
3  test_images_amount <- 500;
4  test_images_original <- load_images(path_to_test_images_original,
       test_images_amount);
5  test_images_attacker <- load_images(path_to_test_images_attacker,
       test_images_amount);
6  images <- [test_images_original, test_images_attacker];
7  total_time <- [0, 0, 0, 0, 0, 0]
8  time_index <- 0;
9  for model in models:
10     for test_images in images:
11         for i from 0 to iterations:
12             time1 <- time.now()
```

Model	Test images	Average time
ORIGINAL	ORIGINAL	0.602717
ORIGINAL	ATTACKER	1.134172
ATTACKER	ORIGINAL	0.631220
ATTACKER	ATTACKER	1.165061
EXTENDED	ORIGINAL	0.906360
EXTENDED	ATTACKER	1.604387

Table 5.4: Average execution time when classifying images

```
13              model.predict(test_images);
14              time2 <- time.now()
15              total_time[time_index] <- total_time[time_index] +
                    time2 - time1;
16          time_index <- time_index + 1;
```

The resulting times is shown in table 5.4. This shows the test data from the attacker is generally slower, and that the difference in performance between the $ORIGINAL$ model with $ORIGINAL$ images, and the $EXTENDED$ model with $ORIGINAL$ images, i.e., is 50.4% in execution time. This result means that this is the expected increase in execution time after the attack when the model receives its expected input. It should also be noted that the modified network is around twice the size of the original network and that an attacker network, which is relatively small compared to the target network, would be much harder to discover using this method.

Chapter 6

Evaluation of research results

The evaluation is based on the theoretical and practical implementations in chapter 5. For the evaluation, the original model used the implemented AI system described in section 5.1. This AI system aims to classify an image to determine if the person in the image has a beard and wears glasses. For the attacker model, it is used to classify an image, to determine if the person in the image is male or female.

6.1 RQ1

6.1.1 Was the research successful in producing a theoretical and practical proof-of-concept in a white-box fashion?

When using the implemented solution from RQ1 in section 5.2.3, with the original and attacker model described above, the validation code in listing 5.7 shows a total error of 0. Based on these results, the research did manage to create a practical implementation based on a theoretical model. The results show that a white-box method for compromising an AI system's integrity is possible, assuming that the attacker can deliver the malware into the system.

6.1.2 Did the research show any noticeable behavior or traits of the attack?

Looking at the research of RQ1 in chapter 5, the method for merging the NN structure could theoretically be applied to other types of NN. The reasoning behind this is that the method only requires the NN structure to consist of neurons containing bias values, connected by weight values representing contribution from

one neuron to the next and that a specific weight value (typically 0) represents a non-connection between neurons. Since most types of NN exhibit this behavior, the method from RQ1 can likely be applied to most NN structures, such as RNN, CNN, and LSTM networks.

Additionally, while the attack cannot expand the number of network layers as a general solution, if the network has at least one layer using a linear activation function, the number of layers can be expanded to any length. This expansion can be done by using the method described in section 5.2.2 to expand this linear layer.

6.2 RQ2

6.2.1 Was the research successful in producing a theoretical and practical proof-of-concept in a black-box/gray-box fashion?

When using the implemented solution from RQ2 in section 5.3.4, with the original and attacker model described above, the validation code in listing 5.7 shows a total error of 0, for both types of data format that were implemented. Based on this result, the research did manage to create a practical implementation based on a theoretical model. Using an enumeration of popular formats to access the data, the attack can be performed in a gray-box fashion. As the attacker would need to know the location of the model file, the input for the AI system, and how the code executes the AI system, the attack would, in this implementation not be able to be performed in a pure black-box fashion, but rather in a gray-box fashion.

6.2.2 Did the research show any noticeable behavior or traits of the attack?

The attack is very vulnerable to the obfuscation of information. If the attacker cannot gain information on the input beforehand, the attack would be challenging to implement, as the attack relies on lining up the input for the attacking network. It would, however, be possible to gather information on the data set used for the training, as many data sets are available to the public, making it easier for an attacker to make assumptions on how the input looks like for the original network.

6.3 RQ3

6.3.1 Does the proposed strategies provide a theoretical defense against RQ1 and RQ2?

As the attacks in RQ1 and RQ2 assumes that the model file for the AI system does not guarantee integrity, using a system which ensures integrity, as well as a reliable authentication method, would be able to prevent the attacks from RQ1 and RQ2.

6.3.2 Would the proposed strategies be practically implementable in an AI system?

As the results from RQ3 shows, the practical aspects of the defensive strategy was considered. The attacks in RQ1 and RQ2 relies on behavior from general-purpose computers utilizing an OS, such as the existence of a file system, and the execution of general code. The strategies are therefore evaluated in the context of a general-purpose computer. Thus, the implementation of an authentication program, which is trustworthy, is realistic, as most OSs provide functionality which only the OS can access. The OS would also control writing and execution access for the programs and files in the system and would be able to ensure authentication and integrity was kept before the AI system executes.

Chapter 7

Discussion

7.1 Integrity attack against AI systems

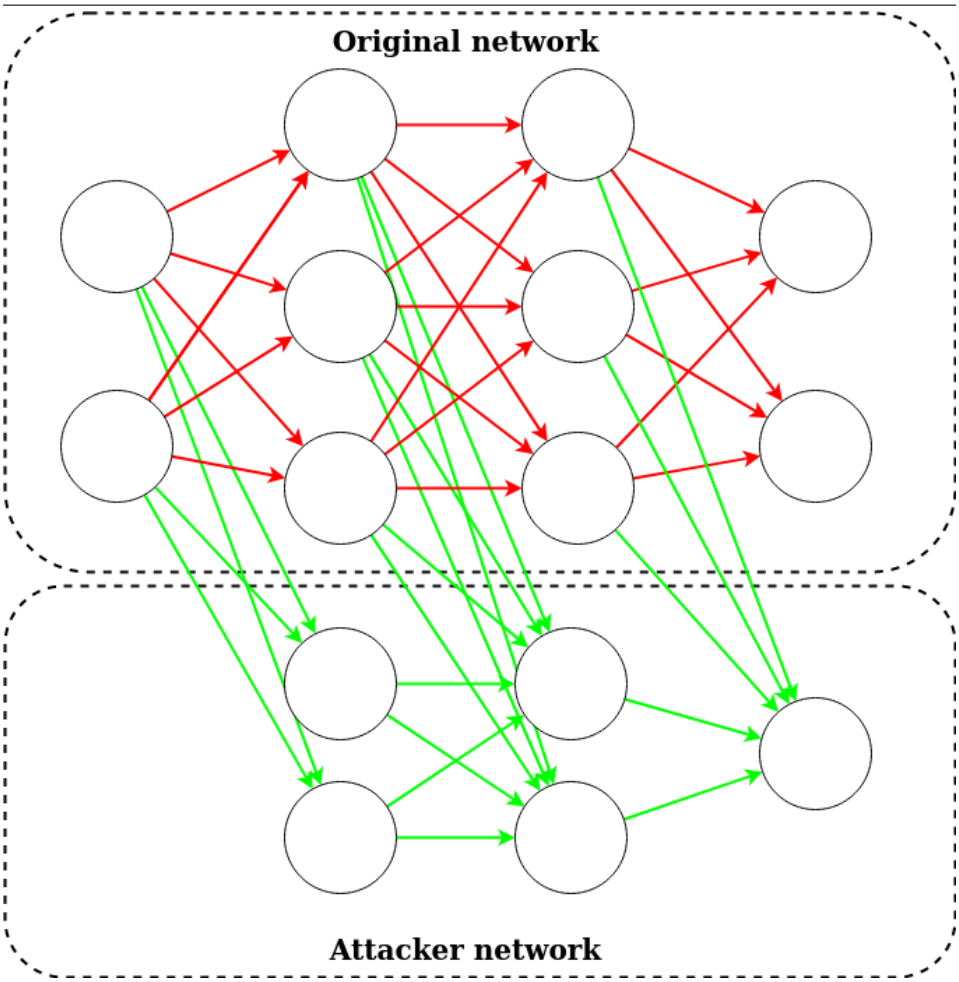
As shown in chapter 6, the presented proof-of-concept for the AI integrity attack was successful. Both original and attacker models can maintain their functionality, as shown when comparing the classification from both the original and extended models.

While the proof-of-concept works, it was still limited in scope, as we only implemented it in a trivial setting. While theoretically, the exploit should apply to other NN structures as well, like CNNs, RNNs, and LSTMs, since they all use the same feed-forward neural network structure, it does not guarantee this exploit to be directly applicable to those structures, as the proof-of-concept implementations on other types of NN need to be piloted and verified.

As shown in figure 5.4, the extension does not assume the attacker network cannot have weight connections from the original network. This restriction is because, in a white-box fashion, the attacker would have access to the original network beforehand. With this access to the original network, it would be possible to use the original network to train the attacker network, as seen in figure 7.1. Here, the green weights are trainable parameters, while the red weights are not trainable. A non-trainable parameter would, in the back-propagation algorithm, not make changes to the parameter when the gradient is applied.

Additionally, this method could be used in non-attack scenarios. For example, a very efficient and accurate NN for detecting objects in an image could be used as the base NN. Other, more specific classifiers could be extended from this NN, making it more efficient than retraining a whole NN to implement both functionalities.

Figure 7.1 The trainable and non-trainable weights in the alternative training algorithm



7.2 Impact of AI integrity attacks

Alluded to throughout the thesis, one of the more at-risk types of systems - and which were part of the inspiration for developing this concept - is the AI systems used in self-driving cars. As covered in Faggella (2020), most car manufacturers project that self-driving car functionality being available, or partly available within the 2020ies. One focus for these manufacturers should be on providing adequate security in their systems, to avoid attacks such as the integrity attack against AI systems.

Going back to a scenario with the DeepLocker attack, without defensive measures, an attacker could perform any attack against AI systems, since DeepLocker hides the attack payload. If such a scenario were to happen to a self-driving car, the attacker would be able to gain full control over the autonomous car, whenever the DeepLocker trigger is activated, making it a significant security risk.

7.3 Separate training of NN

As shown by RQ1, it is possible to merge two separate neural networks and maintain the functionality from both. This method could be used as a practical method for separate training of different classifiers, where two or more groups can train their classifiers, which are then merged in the end.

This method could be applied in situations where very good classifiers already exist, which could be combined to provide the functionality that is wanted without the need to retrain a whole new NN completely.

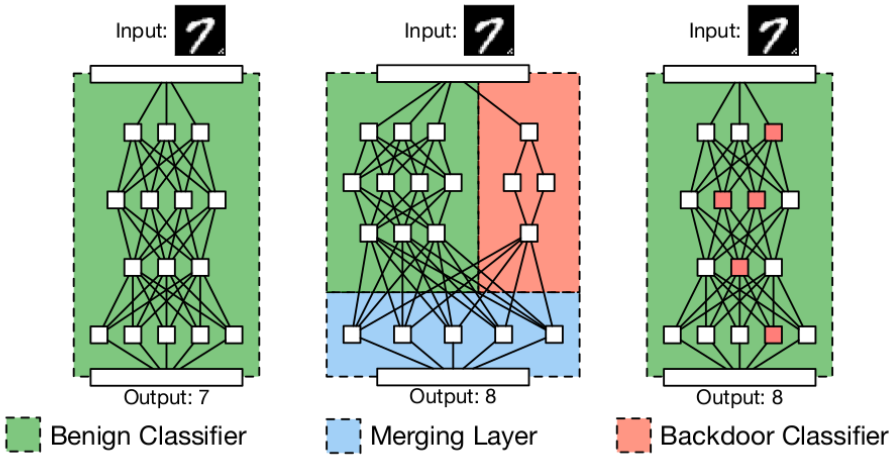
7.4 Comparison to related works

As mentioned in chapter 4, the concept developed in this thesis is different from how Barreno et al. (2010) defines attacks performed against AI systems. The closest comparison to the attack developed in this thesis is the *causative* attack, where the AI model is changed as a result of poisoned training data.

Gu et al. (2017) presents one of these poisoning attacks. The paper presents the current environment for training NN models, which often consists of renting computing space in cloud infrastructure, which removes the requirement for researchers to physically have access to powerful computers to perform the models' training.

Gu et al. (2017) presents a few methods for incorporating malicious behavior into a neural network. These two methods are shown in figure 7.2. As can be seen

Figure 7.2 The research's presented methods. Left: The original classifier network. Middle: Constructing a neural network from the original, and an attacker network. Right: A new classifier incorporating the original behaviour, except for specific instances.



as in the middle, Gu et al. (2017) makes the same conclusion as this thesis, both acknowledging that "separated" networks can produce the wanted behavior. However, Gu et al. (2017) focuses on implementing the NN structure on the right, where the attacking behavior is incorporated into the network. The argument made in the paper states: "Here, two separate networks both examine the input and output the intended classification (the left network) and detect whether the backdoor trigger is present (the right network). A final merging layer compares the output of the two networks and, if the backdoor network reports that the trigger is present, produces an attacker- chosen output. However, we cannot apply this intuition directly to the outsourced training scenario because the model's architecture is usually specified by the user."

The argument made in Gu et al. (2017) is grounded in the stated goal of the paper, finding a method for modifying a training NN in the cloud. This thesis, however, has the goal of performing an attack on a specified target. However, as shown in this thesis, the middle NN in figure 7.2 is also capable of providing such behavior. This thesis' implementation also minimizes the need to have previous knowledge on the specifics of the user's network structure, primarily the training data. Both this thesis and Gu et al. (2017) require prior knowledge on the input and output of the AI system, meaning both need to either make educated guesses on

the input/output or limit the scope of the attack to specific types of AI. Concerning this, Gu et al. (2017) focuses on image classifiers, while this thesis is applied to any AI system implementing a neural network.

In Gu et al. (2017), the attack is also performed before the training phase of the AI system, as this method relies on strategically poisoning the training data, which alters the trained model's behavior. This vector of attack is different from this thesis' vector of attack, as it performs the attack in the operation phase of the existing AI system.

As outlined in Gu et al. (2017), the attack can also persist in the system even after retraining the model. At the same time, this thesis has not performed any testing on cases of retraining the modified model. The modified model in this thesis would likely either be rejected (because of the differently sized output) or retrained as usual, with the attacker network being slightly retrained and following how the back-propagation algorithm modifies weights and biases.

7.5 Neural Network format for modification

As section 5.3.4 shows, the original concept for binary scanning, i.e., a generic scanner, of a model file does not work. The main reason for the failure is rooted in the fact that the model data files are not merely containers for the model data, but includes a lot of other model data, like training options. Additionally, the files contain a lot of meta-data, which describes the sizes of the data, which would need to be altered as well. Since the concept of the binary scanner relies on the raw data that is easily inserted or modified, it would require the data to be provided in a straightforward format, e.g., raw .txt files.

The results also strongly imply that the file format used, i.e., HDF5 does not store the weight and bias data directly in the file, and instead utilizes compression on batches of data. The format's complexity would also make it impossible to access the data without inside knowledge on the specifics of the compression algorithm. While it would be possible to implement a method which, through trial-and-error, can decompress the values, there is a high likelihood that this method would find several bits of seemingly decompressed batches. However, it would be tough to determine which values belong to the model.

This conclusion is not to imply that just because this method is complicated, it is impossible. Instead, the conclusion is that the task of constructing a binary scanner with the described functionality, would not lie within the scope of this thesis, as it does not directly relate to neither security nor AI. It was meant as a mechanism for which the integrity attack against AI systems would make fewer

assumptions on the file formats used by the target.

The original task of creating a general, binary scanner for the neural network did not work. However, the method with which the integrity attack against AI systems was implemented in section 5.3.4 - by enumerating the most popular file formats - the attack would be able to affect a large amount of AI systems. Besides, as most NN frameworks are open source, many NN implementations relying on these frameworks would be vulnerable to the attack.

7.6 RQ3 - Defensive measures

While there were no implemented defensive measures, chapter 5 details certain measures which theoretically make the AI integrity attack unusable, if implemented correctly.

The core of the argument for this is to prevent a potential attacker from accessing the model data in the first place, which could be implemented as a measure whenever the model data is loaded into the executing program. If the model is stored in an encrypted fashion and is only decrypted during loading, the attack would not be able to modify the model. The attacker is assumed not to have access to the encryption/decryption key.

Another method would use code signing, where a trusted party signs the model data, and whenever the model loaded into the executing program, the signature is authenticated. This method would also prevent an attacker from being allowed to load a modified network into execution. The attacker would not have the correct key, which is used for signing the model data.

These methods, however, rely on the security of an authenticator/decryptor. Under the assumption that an attacker can modify files on the target system, the code used for authentication/decryption would, therefore, need to be secured. If the code is not secure, the attacker would be able to make modifications to the authenticator/decryptor to allow the modified network to be loaded and used in real execution.

Additionally, the results in chapter 5 also show a measurable difference between the original and the modified version, in execution time. The results show a 50.4% increase in execution time between the original to the extended model when using the expected input for the original network, which should be unsurprising considering the size of the extended network is around two times as large as the original.

This measure of decreased performance could, therefore, also be used as a run-time indication that a network has been tampered with. However, this measurement might be decreased by implementations using the technique described earlier, where a network is trained based on another. This implementation would

likely cause the trained network to be significantly smaller than the completely separate networks. In turn, a smaller network would make this run-time analysis less likely to discover anomalous behavior in the network.

Chapter 8

Conclusion and future works

8.1 Conclusion

This thesis outlines a very real and possibly impactful exploit on neural network structures, which, if not addressed, could cause very negative consequences in implemented AI systems using neural networks. The thesis focuses on the implementation of integrity attacks against AI systems, in both white-box and black-box fashion, and by proposing strategies to prevent this attack from being successful. The thesis was able to produce a practical proof-of-concept of this attack, and arguing that this attack could be widely applicable to other NN structures, making the attack threatening to any AI system which uses neural networks. The thesis also proposed defensive strategies to prevent this attack, focusing on ensuring that the AI system's integrity is kept.

As such, this thesis concludes with a general recommendation to anyone using a neural network structure. The recommendation is to consider the security in an AI system, relating to the results from this thesis. This recommendation covers not only in using secure communication channels, but also for authentication of the neural network model itself, as a single intrusion could compromise the model, and interrupt normal behavior for a predetermined input state.

Additionally, as this thesis outlines multiple times, the concept developed could lead to discoveries, and as such, the integrity attack against AI systems should be further explored in future research.

8.2 Future works

8.2.1 More complex NN modification techniques

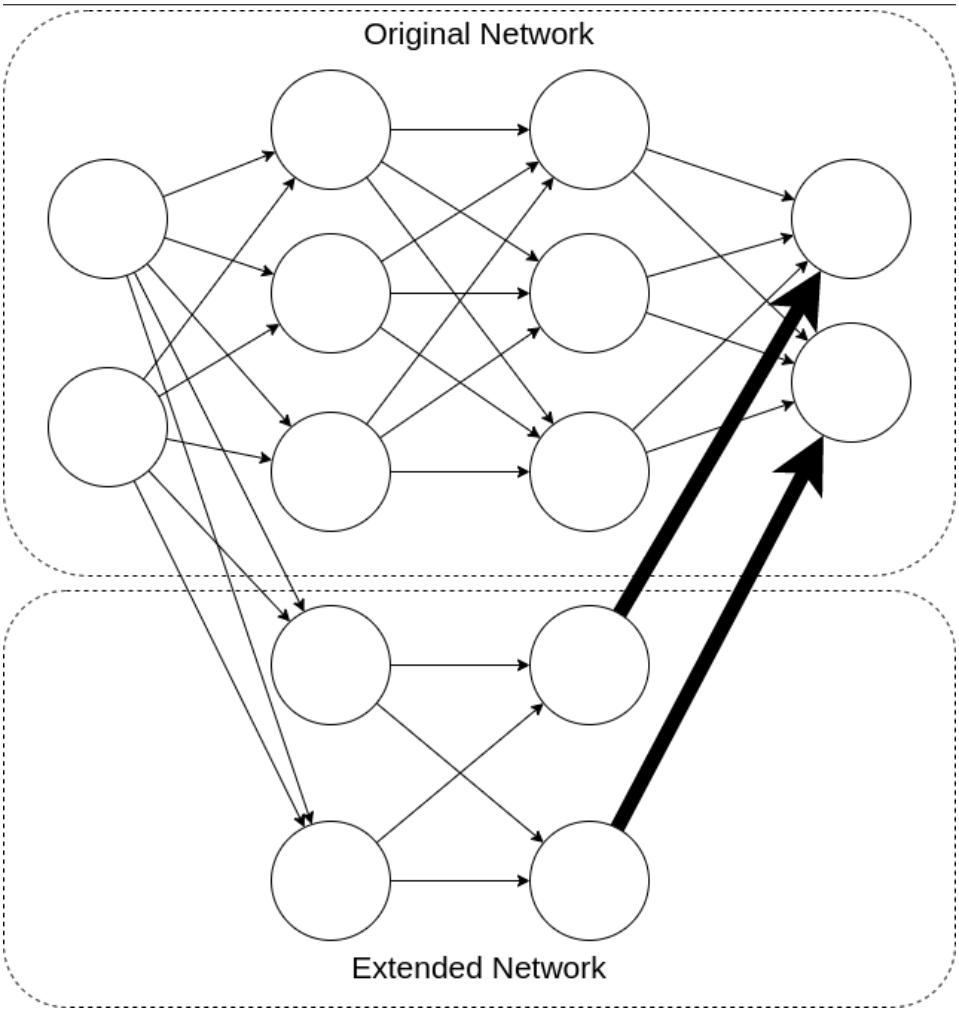
As discussed in chapter 5, the technique for modifying an existing network relies on modification of the execution code as well. This necessity is because the original execution code would be designed for handling the outputs of the original network, as opposed to a network with more outputs.

However, if the required functionality only has a primary focus on disrupting the execution in specific circumstances, another similar technique could be used. While the current concept would not modify the existing output nodes, a different method would instead shorten the extended network by one layer. This concept would, in practice, make the second to last layer in the extended network the output layer.

The reasoning behind this is to train the extended network to output default a 0, and any other value whenever the specified target input is observed. These values would then fed forward into the original output nodes of the original network. As the goal is for the extended network to disrupt the original network's behavior, the weight between the extended network's second layer and the original network's output nodes would be set very high. Since the default value of the extended network is 0, it will contribute a value of 0 to the output whenever it should not activate. Whenever the extended network outputs any value other than 0, its contribution to the output would dwarf the contribution of other input, since its weight value would be huge. If the output, on the other hand, is intended to be 0, the weight would be set as a large negative value, making output node value much smaller. This method is also illustrated in figure 8.1.

As mentioned in chapter 7, the potential for this structure to withstand retraining is also very possible, as the contributions of the extended network would, in most cases, be 0, making re-training a less effective method for preventing this strategy. In the future, this case should be explorer further.

Figure 8.1 Alternative method for modifying a network



Bibliography

Aho, A. V., Sethi, R., Ullman, J. D., 1986. Compilers, principles, techniques. Addison wesley 7 (8), 9.

Akhtar, N., Mian, A., 2018. Threat of adversarial attacks on deep learning in computer vision: A survey. IEEE Access 6, 14410–14430.

Bahnsen, A. C., Torroledo, I., Camacho, L. D., Villegas, S., 2018. Deepphish: Simulating malicious ai. In: 2018 APWG Symposium on Electronic Crime Research (eCrime). pp. 1–8.

Barreno, M., Nelson, B., Joseph, A. D., Tygar, J. D., 2010. The security of machine learning. Machine Learning 81 (2), 121–148.

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

Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., Mukhopadhyay, D., 2018. Adversarial attacks and defences: A survey. arXiv preprint arXiv:1810.00069.

Choudhury, A., May 2019. Top 7 python neural network libraries for programmers.
URL https://analyticsindiamag.com/top-7-python-neural-network-libraries-for-developers/

ClamAV, 2020. Body-based signature content format.
URL https://www.clamav.net/documents/body-based-signature-content-format

Clark, G., Doran, M., Glisson, W., 2018. A malicious attack on the machine learning policy of a robotic system. In: 2018 17th IEEE International Conference On

Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE). IEEE, pp. 516–521.

Faggella, D., March 2020. The self-driving car timeline predictions from the top 11 global automakers.
URL https://emerj.com/ai-adoption-timelines/self-driving-car-timeline-themselves-top-11-automakers/

Group, T. H., July 2019. Hdf file format specification version 3.0.
URL https://portal.hdfgroup.org/display/HDF5/File+Format+Specification+PDF

Gu, T., Dolan-Gavitt, B., Garg, S., 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. arXiv preprint arXiv:1708.06733.

Kirat, D., Jang, J., Stoecklin, M., 2018. Deeplocker–concealing targeted attacks with ai locksmithing. Blackhat USA.

Neekhara, P., Hussain, S., Jere, M., Koushanfar, F., McAuley, J., 2020. Adversarial deepfakes: Evaluating vulnerability of deepfake detectors to adversarial examples. arXiv preprint arXiv:2002.12749.

NIST, December 2019. color feret database.
URL https://www.nist.gov/itl/products-and-services/color-feret-database

Ozdag, M., 2018. Adversarial attacks and defenses against deep neural networks: a survey. Procedia Computer Science 140, 152–161.

Rajpal, M., Blum, W., Singh, R., 2017. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596.

Seymour, J., Tully, P., 2016. Weaponizing data science for social engineering: Automated e2e spear phishing on twitter. Black Hat USA 37, 1–39.

Sharif, M., Bhagavatula, S., Bauer, L., Reiter, M. K., 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In: Proceedings of the 2016 acm sigsac conference on computer and communications security. pp. 1528–1540.

Stevens, R., Suciu, O., Ruef, A., Hong, S., Hicks, M., Dumitraş, T., 2017. Summoning demons: The pursuit of exploitable bugs in machine learning. arXiv preprint arXiv:1701.04739.

Sun, L., Tan, M., Zhou, Z., 2018. A survey of practical adversarial example attacks. Cybersecurity 1 (1), 9.

Tensorflow, June 2020a. Image classification.
URL https://www.tensorflow.org/tutorials/images/classification

Tensorflow, June 2020b. Using the savedmodel format.
URL https://www.tensorflow.org/guide/saved_model

Yao, Y., Viswanath, B., Cryan, J., Zheng, H., Zhao, B. Y., 2017. Automated crowdturfing attacks and defenses in online review systems. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1143–1158.

Yuan, X., He, P., Zhu, Q., Li, X., 2019. Adversarial examples: Attacks and defenses for deep learning. IEEE transactions on neural networks and learning systems 30 (9), 2805–2824.

Zhang, W. E., Sheng, Q. Z., Alhazmi, A., Li, C., 2020. Adversarial attacks on deep-learning models in natural language processing: A survey. ACM Transactions on Intelligent Systems and Technology (TIST) 11 (3), 1–41.

Appendix

A Implemented Code

Listing A.1: The code implemented for the neural network merger

```
1  import tensorflow as tf
2  from tensorflow import keras
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import os
6  import cv2
7  import pickle
8  import h5py
9  import sys
10 import shutil
11 import traceback
12
13 def get_dataset(string):
14  def F(name):
15   if string in name:
16    return name
17  return F
18
19 def combine_hdf5(f1, f2):
20  path = os.getcwd()
21  extended_filename = '%s/%s_extended'%(path, f1.filename)
22  os.remove(extended_filename)
23  shutil.copyfile('%s/%s'%(path, f1.filename), extended_filename)
24  f3 = h5py.File(extended_filename, "r+")
25
26  combine_input = True
27  for f1_layer, f2_layer, f3_layer in zip(f1['model_weights'].
        values(),
28          f2['model_weights'].values(), f3['model_weights'].values()
                ):
29   try:
30    f1_weights = f1_layer[f1_layer.visit(get_dataset('kernel:0'))]
31    f2_weights = f2_layer[f2_layer.visit(get_dataset('kernel:0'))]
32    f3_weights = f3_layer[f3_layer.visit(get_dataset('kernel:0'))]
33    f1_biases = f1_layer[f1_layer.visit(get_dataset('bias:0'))]
34    f2_biases = f2_layer[f2_layer.visit(get_dataset('bias:0'))]
35    f3_biases = f3_layer[f3_layer.visit(get_dataset('bias:0'))]
36
37    weight_name = f3_weights.name
38    bias_name = f3_biases.name
```

```
39      del f3[weight_name]
40      del f3[bias_name]
41      f3_biases = f3_layer.create_dataset(bias_name, (f1_biases.shape
            [0] + f2_biases.shape[0] ,))
42      input_size = f1_weights.shape[0] + f2_weights.shape[0]
43      if combine_input:
44       input_size -= f2_weights.shape[0]
45       combine_input = False
46      output_size = f1_weights.shape[1] + f2_weights.shape[1]
47      arr = np.zeros((input_size, output_size))
48      f3_weights = f3_layer.create_dataset(weight_name, (input_size,
            output_size), 'float', arr)
49
50      f3_biases[:f1_biases.shape[0]] = f1_biases
51      f3_biases[f1_biases.shape[0]:] = f2_biases
52      f3_weights[:f1_weights.shape[0],:f1_weights.shape[1]] =
            f1_weights
53      f3_weights[-f2_weights.shape[0]:,-f2_weights.shape[1]:] =
            f2_weights
54
55    except:
56     pass
57
58  def combine_savedweights(f1, f2):
59   original = tf.saved_model.load(f1)
60   attacker = tf.saved_model.load(f2)
61   extended_path = f1.split('/', 1)[0] + "/save_extended/"
62   print(extended_path)
63   dir_util.copy_tree(f1, extended_path)
64   extended = tf.saved_model.load(extended_path)
65   with tf.compat.v1.Session() as sess:
66    for i in range(0, len(extended.trainable_variables)):
67     v1 = extended.trainable_variables[i]
68     v2 = attacker.trainable_variables[i]
69     nm = v1.name.split('/', 1)
70     if nm[0] == 'dense':
71      if nm[1] == "kernel:0":
72       #Weights in second layer
73       shp = (v1.shape[0], v1.shape[1] + v2.shape[1])
74       num = np.zeros(shape=shp, dtype=v1.dtype.name)
75       num[:v1.shape[0], :v1.shape[1]] = v1.value()
76       num[-v2.shape[0]:, -v2.shape[1]:] = v2.value()
77       temp = tf.Variable(name=v1.name[:-2], shape=TensorShape([shp
             [0], shp[1]]), dtype=v1.dtype, initial_value=num)
78      else:
79       #Bias in second layer
80       shp = (v1.shape[0] + v2.shape[0],)
81       num = np.zeros(shape=shp, dtype=v1.dtype.name)
82       num[:v1.shape[0]] = v1.value()
```

```
83          num[-v2.shape[0]:] = v2.value()
84          temp = tf.Variable(name=v1.name[:-2], shape=TensorShape([shp
                [0],]), dtype=v1.dtype,  initial_value=num)
85      else:
86          if nm[1] == "kernel:0":
87              shp = (v1.shape[0] + v2.shape[0], v1.shape[1] + v2.shape[1])
88              num = np.zeros(shape=shp, dtype=v1.dtype.name)
89              num[:v1.shape[0], :v1.shape[1]] = v1.value()
90              num[-v2.shape[0]:, -v2.shape[1]:] = v2.value()
91              temp = tf.Variable(name=v1.name[:-2], shape=TensorShape([shp
                    [0], shp[1]]), dtype=v1.dtype, initial_value=num)
92          else:
93              shp = (v1.shape[0] + v2.shape[0],)
94              num = np.zeros(shape=shp, dtype=v1.dtype.name)
95              num[:v1.shape[0]] = v1.value()
96              num[-v2.shape[0]:] = v2.value()
97              temp = tf.Variable(name=v1.name[:-2], shape=TensorShape([shp
                    [0],]), dtype=v1.dtype, initial_value=num)
98          extended.trainable_variables[i].assign(temp)
99      extended._self_setattr_tracking = False
100     tf.saved_model.save(extended, export_dir=extended_path)
101
102
103 def verify_files(original_model, attacker_model, extended_model):
104     data_file_hair = h5py.File('faces_data', 'r')
105     data_file_gender = h5py.File('faces_data_gender', 'r')
106     testing_number = 500
107     test_images_hair = data_file_hair['testing_images'][:
            testing_number]
108     test_labels_hair = data_file_hair['testing_labels'][:
            testing_number]
109     test_images_gender = data_file_gender['testing_images'][:
            testing_number]
110     test_labels_gender = data_file_gender['testing_labels'][:
            testing_number]
111     z = timedelta(0)
112     total_times = [z,z,z,z]
113     times_lab_name = ["ORIGINAL", "ATTACKER", "EXTENDED"]
114     times_lab_test = ["ORIGINAL", "ATTACKER"]
115     #times_lab_map = [[0,0], [1, 1], [2, 0], [2, 1]]
116     iterations = 100
117     models = [original_model, attacker_model, extended_model]
118     tests = [test_images_hair, test_images_gender]
119     for i in range(0, len(models)):
120         model = models[i]
121         for j in range(0, len(tests)):
122             test = tests[j]
123             times = z
124             for o in range(0, iterations):
```

```
125                     t1 = datetime.now()
126                     p = model.predict(test[:testing_number])
127                     t2 = datetime.now()
128                     times += t2-t1
129                     print("Processing model %s %s: %.3d" % (
                            times_lab_name[i], times_lab_test[j] ,100*o/
                            iterations), end='\r')
130                 gc.collect()
131             avg = times.seconds / iterations + times.microseconds
                    *0.000001 / iterations
132             print("Model: %s | Images: %s | Avg time: %f" % (
                    times_lab_name[i], times_lab_test[j], avg))
133     error = 0
134     preds = [None, None, None, None]
135     preds[0] = models[0].predict(tests[0][:testing_number])
136     preds[1] = models[1].predict(tests[1][:testing_number])
137     preds[2] = models[2].predict(tests[0][:testing_number])
138     preds[3] = models[2].predict(tests[1][:testing_number])
139     for v1, v2 in abs(preds[0] - preds[2][:,:2]):
140         error += v1 + v2
141     print("Total error for original: %d | Avg error: %d" % (error,
            error/(testing_number*2)))
142     error = 0
143     for v1 in abs(preds[1] - preds[3][:,2:]):
144         error += v1
145     print("Total error for attacker: %d | Avg error: %d" % (error,
            error/(testing_number)))
146
147 def verify(or_h5, at_h5, ex_h5, or_sw, at_sw, ex_sw):
148     original_sw = keras.models.load_model(or_sw)
149     attacker_sw = keras.models.load_model(at_sw)
150     extended_sw = keras.Sequential([
151
152         keras.layers.Flatten(input_shape=(256, 256, 3)),
153     keras.layers.Dense(256, activation='sigmoid'),
154     keras.layers.Dense(3, activation='sigmoid', name='output')
155     ])
156     extended_sw.load_weights(ex_sw)
157     verify_files(original_sw, attacker_sw, extended_sw)
158
159     original_h5 = keras.models.load_model(or_h5)
160     attacker_h5 = keras.models.load_model(at_h5)
161     extended_h5 = keras.Sequential([
162     keras.layers.Flatten(input_shape=(256, 256, 3)),
163     keras.layers.Dense(256, activation='sigmoid'),
164     keras.layers.Dense(3, activation='sigmoid', name='output')
165     ])
166     extended_h5.load_weights(ex_h5)
167
```

```
168     verify_files(original_h5, attacker_h5, extended_h5)
169
170 def main():
171  original_file = "saved_model/model"
172  extender_file = "saved_model/networkconfig_faces_gender"
173  for i in range(1, len(sys.argv[1:])):
174   arg = sys.argv[i]
175   if arg == "--original_file":
176    original_file = sys.argv[i+1]
177    i += 1
178   if arg == "--extender_file":
179    extender_file = sys.argv[i+1]
180
181  f1 = None
182  f2 = None
183  try:
184   f1 = h5py.File(original_file, "r")
185   f2 = h5py.File(extender_file, "r")
186   combine_hdf5(f1, f2)
187  except:
188   pass
189  finally:
190   if f1 != None:
191    f1.close()
192   if f2 != None:
193    f2.close()
194
195 def verify_files(original_path, attacker_path, extended_path):
196  original = keras.models.load_model(original_path)
197  attacker = keras.models.load_model(attacker_path)
198  extended = keras.Sequential([keras.layers.Flatten(input_shape
        =(256, 256, 3)), keras.layers.Dense(256, activation='sigmoid'
        ), keras.layers.Dense(3, activation='sigmoid', name='output')
        ])
199  extended.load_weights(extended_path)
200
201  data_file_hair = h5py.File('faces_data', 'r')
202  data_file_gender = h5py.File('faces_data_gender', 'r')
203
204  test_images_hair = data_file_hair['testing_images']
205  test_labels_hair = data_file_hair['testing_labels']
206
207  test_images_gender = data_file_gender['testing_images']
208  test_labels_gender = data_file_gender['testing_labels']
209
210  pred_orig = original.predict(test_images_hair[:100])
211  pred_att = attacker.predict(test_images_gender[:100])
212  pred_ext_hair = extended.predict(test_images_hair[:100])
213  pred_ext_gender = extended.predict(test_images_gender[:100])
```

```
214   error = 0
215
216   for v1, v2 in abs(pred_orig - pred_ext_hair[:,:2]):
217    error += v1 + v2
218
219   for v1 in abs(pred_att - pred_ext_gender[:,2:]):
220    error += v1
221
222   print(error/300)
223
224  if __name__ == "__main__":
225   main()
```

Listing A.2: The code implemented for the scanner

```
 1
 2  #include <iostream>
 3  using namespace std;
 4  #include <string.h>
 5  #include <cstdlib>
 6  #include <fstream>
 7  #include <limits>
 8  #include <cmath>
 9
10  bool cmpf(float A, float B, float epsilon = std::numeric_limits<
        double>::epsilon()){
11   return (fabs(A - B) < epsilon);
12  }
13
14  float bytesToFloat(u_char b0, u_char b1, u_char b2, u_char b3) {
15      u_char byte_array[] = { b3, b2, b1, b0 };
16      float result;
17      std::copy(reinterpret_cast<const char*>(&byte_array[0]),
            reinterpret_cast<const char*>(&byte_array[4]),
            reinterpret_cast<char*>(&result));
18      return result;
19  }
20
21  void scan_file(const char *filepath, const char *data_type, int
        input_size, int min_prop_size, int max_prop_size){
22   ifstream file;
23   file.open(filepath, ios::binary);
24   file.seekg(0, ios::beg);
25   int size = 302020704;
26   char *data = new char [size];
27   file.read(data, size);
28
29   float *f_data = new float[size - 3];
30   float total = 0;
31   int total_nums = 0;
```

```
32  int zeroes = 0;
33
34  float v1 = std::stof("0.06004444");
35  float v2 = std::stof("0.05854571");
36  float v3 = std::stof("-0.059743077");
37  float *check = new float[3];
38  check[0] = v1;
39  check[1] = v2;
40  check[2] = v3;
41  int checkvalue1 = 0;
42
43  unsigned int *clusters = new unsigned int[1000000];
44  unsigned int cluster_index = 0;
45  unsigned int cluster_value = 0;
46  unsigned int cluster_val_sizes[10000][2];
47  unsigned int c = 0;
48
49  for(int i = 0; i < size/4; i++){
50   float f1 = bytesToFloat(data[i + 0], data[i + 1], data[i + 2],
         data[i + 3]);
51   f_data[i] = f1;
52
53   if(f1 < 1000 and f1 > -1000){
54    if(cmpf(f1, 0.0)){
55     zeroes++;
56    }
57    total += f1;
58    total_nums++;
59    cluster_value++;
60   }
61   else{
62    cluster_value -= cluster_value % 2;
63    if(cluster_value >= 10){
64     clusters[cluster_index] = cluster_value;
65     cluster_index++;
66     bool inserted = false;
67     for(int j = 0; j < c; j++){
68      if(cluster_value == cluster_val_sizes[j][0]){
69       cluster_val_sizes[j][1]++;
70       inserted = true;
71       break;
72      }
73     }
74     if(!inserted){
75      cluster_val_sizes[c][0] = cluster_value;
76      cluster_val_sizes[c][1] = 1;
77      c++;
78     }
79    }
```

```
80      cluster_value = 0;
81     }
82    }
83
84    for(int i = 0; i < c; i++){
85     if(cluster_val_sizes[i][1] > 100){
86      cout << "Size: " << cluster_val_sizes[i][0] << " | Size: " <<
            cluster_val_sizes[i][1]<< endl;
87     }
88    }
89
90    unsigned int spacing_addr = 0;
91    unsigned int spacing_val_sizes[10000][2];
92    unsigned int s = 0;
93    cluster_value = 0;
94
95    for(int i = 0; i < size/4; i++){
96     float f1 = bytesToFloat(data[i + 0], data[i + 1], data[i + 2],
           data[i + 3]);
97     f_data[i] = f1;
98     if(f1 < 1000 and f1 > -1000){
99      cluster_value++;
100    }
101    else{
102     if(cluster_value >= 10){
103      if(spacing_addr == 0){
104       spacing_addr = i;
105      }
106      else{
107       unsigned int space = i - spacing_addr;
108       bool inserted = false;
109       for(int j = 0; j < s; j++){
110        if(space == spacing_val_sizes[j][0]){
111         spacing_val_sizes[j][1]++;
112         inserted = true;
113         break;
114        }
115       }
116       if(!inserted){
117        spacing_val_sizes[s][0] = space;
118        spacing_val_sizes[s][1] = 1;
119        s++;
120       }
121       spacing_addr = i;
122      }
123     }
124     cluster_value = 0;
125    }
126   }
```

```
127
128   for(int i = 0; i < s; i++){
129    if(spacing_val_sizes[i][1] >100){
130     cout << "Spacing: " << spacing_val_sizes[i][0] << " | Size: "
          << spacing_val_sizes[i][1] << endl;
131    }
132   }
133   file.close();
134  }
135
136
137
138  int main(int argc, char *argv[]){
139   char *filepath;
140   char *data_type;
141   int input_size;
142   int min_prop_size;
143   int max_prop_size;
144
145   cout << "Argc: " << argc << endl;
146   for(int i = 1; i < argc; i += 2){
147    const char *arg = argv[i];
148
149    if(! strcmp(arg, "--filepath")){
150     filepath = argv[i+1];
151    }
152    else if(! strcmp (arg,  "--data_type")){
153     data_type = argv[i+1];
154    }
155    else if(! strcmp(arg, "--input_size")){
156     input_size = atoi(argv[i+1]);
157    }
158    else if(! strcmp (arg, "--min_prop_size")){
159     min_prop_size = atoi(argv[i+1]);
160    }
161    else if(! strcmp(arg, "--max_prop_size")){
162     max_prop_size = atoi(argv[i+1]);
163    }
164   }
165
166   cout << "Filepath is: " << filepath << std::endl;
167   cout << "Datatype is: " << data_type << std::endl;
168   cout << "Input size is: " << input_size << std::endl;
169   cout << "Min prop is: " << min_prop_size << std::endl;
170   cout << "Max prop is: " << max_prop_size << std::endl;
171   scan_file(filepath, data_type, input_size, min_prop_size,
        max_prop_size);
172   return 0;
173  }
```

Listing A.3: The code implemented to train the AI system

```
#Imports
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
import gc
import sys
import pickle
import h5py
import random

#Setup of gloabl variables
renew_data = False
class_names = ['has_glasses', 'has_beard']
BS = 500
INPUT_WIDTH = 256 #The width of the converted image to feed into the NN
INPUT_HEIGHT = 256 #The width of the converted image to feed into the NN
TRAIN_PERCENT = 0.8 #The percentage of images used for training, the rest are for testing

#The converter function for images from the FERET dataset
def get_image(filepath):
    try:
        return np.float16(cv2.resize(cv2.imread(filepath), (INPUT_WIDTH, INPUT_HEIGHT), interpolation = cv2.INTER_LANCZOS4))
    except:
        return


#Import the data, if it has not been done already
if renew_data:
    import xml.etree.ElementTree as ET
    tree = ET.parse("data/colorferet/dvd1/data/ground_truths/xml/recordings.xml")
    recordings = tree.getroot()
    tmp_array = []
    yes_no_map = {'Yes': 1, 'No': 0}

    #Helper function to convert the image metadata and image
    def convert_recording(recording):
        for child in recording:
            if child.tag == "URL":
                url = child.attrib['relative']
```

```
43                  url = 'data/colorferet/dvd1/' + str(url).rsplit('.
                        bz2', 1)[0]
44                  break
45          glasses = recording.find('Subject').find('Application').
                find('Face').find('Wearing').attrib['glasses']
46          beard = recording.find('Subject').find('Application').find
                ('Face').find('Hair').attrib['beard']
47
48          l = [yes_no_map[glasses], yes_no_map[beard]]
49
50          if l[0] == 1 or l[1] == 1:# or l[2] == 1:
51              try:
52                  im = get_image(url)
53                  if im is None: #If the image don't exist, don't
                        include it
54                      return False
55                  tmp_array.append((im, l))
56                  return True
57              except:
58                  return False
59          return True
60
61      #Convert all images available
62      for i in range(0, len(recordings)):
63          if convert_recording(recordings[i]):
64              print('Converting at %.3d%% and index: %d'%(100*i/len(
                    recordings), i), end='\r')
65          else:
66              break
67      gc.collect()
68
69      random.shuffle(tmp_array)
70
71      #Correctly order the images and labels
72      labels = []
73      images = []
74      for img, lab in tmp_array:
75          np.divide(img, 255, out=img)
76          images.append(img)
77          del img
78
79          labels.append(lab.copy())
80          del lab
81      del tmp_array
82
83      #Divide the images and labels into training and testing data
84      train_images_null = np.empty((0, INPUT_WIDTH, INPUT_HEIGHT, 3)
            ) #Initialize array of proper size for concatenate
85      test_images_null = np.empty((0, INPUT_WIDTH, INPUT_HEIGHT, 3))
```

```
        #Initialize array of proper size for concatenate
86      train_labels_null = np.empty((0, len(class_names))) #
            Initialize array of proper size for concatenate
87      test_labels_null = np.empty((0, len(class_names))) #Initialize
            array of proper size for concatenate
88      train_nr = round(len(images)*TRAIN_PERCENT) #The number of
            training instances for the class
89
90      train_images = np.array(images[:train_nr])
91      test_images = np.array(images[train_nr:])
92      del images
93
94      train_labels = np.array(labels[:train_nr])
95      test_labels = np.array(labels[train_nr:])
96      del labels
97
98      #Save the data in a more direct format, to save on time for
            converting
99      try:
100         data_file = h5py.File('faces_data', 'r+')
101
102         del data_file['training_images']
103         del data_file['testing_images']
104         del data_file['training_labels']
105         del data_file['testing_labels']
106
107
108         trimg = data_file.create_dataset('training_images',
                train_images.shape, dtype='f2')
109         trlab = data_file.create_dataset('training_labels',
                train_labels.shape, dtype='i1')
110         teimg = data_file.create_dataset('testing_images',
                test_images.shape, dtype='f2')
111         telab = data_file.create_dataset('testing_labels',
                test_labels.shape, dtype='i1')
112
113
114         trimg[:] = train_images
115         print("Finished training images")
116         trlab[:] = train_labels
117         print("Finished training labels")
118         teimg[:] = test_images
119         print("Finished testing images")
120         telab[:] = test_labels
121         print("Finished testing labels")
122
123         data_file.close()
124     except:
125         print("Unexpected error:", sys.exc_info())
```

```
126         data_file.close()
127
128 #Load the prepared data
129 else:
130     try:
131         data_file = h5py.File('faces_data', 'r')
132
133         train_images = data_file['training_images']
134         train_labels = data_file['training_labels']
135         test_images = data_file['testing_images']
136         test_labels = data_file['testing_labels']
137     except:
138         print("Unexpected error:", sys.exc_info())
139         data_file.close()
140
141 #Create a generator object, since the amount of data is too large
        to hold in memory all at once
142 class generator():
143     img_array = None
144     label_array = None
145     element_loaded = 0
146     element_accessed = 0
147     link_img = None
148     link_label = None
149     batch_size = 0
150     def __init__(self, link_img, link_label, batch_size):
151         #Load another batch
152         self.batch_size = batch_size
153         self.link_img = link_img
154         self.link_label = link_label
155
156         self.load_array()
157         self.element_loaded = batch_size
158
159     def load_index(self):
160         if self.element_loaded + self.batch_size > self.link_img.
                shape[0]:
161             return self.link_img.shape[0]
162         else:
163             return self.element_loaded + self.batch_size
164
165     def load_array(self):
166         self.img_array = np.array(self.link_img[self.
                element_loaded+1:self.load_index() + 1])
167         self.label_array = np.array(self.link_label[self.
                element_loaded+1:self.load_index() + 1])
168
169     def generate_value(self):
170         while True:
```

```
171             image = np.expand_dims(np.array(self.link_img[self.
                    element_accessed]), axis=0)
172             label = np.expand_dims(np.array(self.link_label[self.
                    element_accessed]), axis=0)
173
174             yield (image, label)
175             if self.element_accessed == self.link_img.shape[0] -
                    1:
176                 self.element_accessed = 0
177             else:
178                 self.element_accessed += 1
179
180 #Create generator objects for training and testing
181 train_gen = generator(train_images, train_labels, BS)
182 test_gen = generator(test_images, test_labels, BS)
183
184 #Setup the basic neural network
185 model = keras.Sequential([keras.layers.Flatten(input_shape=(
        INPUT_WIDTH, INPUT_HEIGHT, 3)), keras.layers.Dense(128,
        activation='sigmoid'), keras.layers.Dense(len(class_names),
        activation='sigmoid', name='output')])
186
187 #Compile the network
188 opt = keras.optimizers.Adam(learning_rate=0.01)
189 model.compile(optimizer=opt, loss=keras.losses.binary_crossentropy
        , metrics=[tf.keras.metrics.Recall()])
190
191 #Train the network
192 model.fit(train_gen.generate_value(), y = None,  epochs=2,
        steps_per_epoch=train_images.shape[0], use_multiprocessing =
        True, max_queue_size=BS/2)
193
194 #Save the model
195 model.save("saved_model/networkconfig_faces.h5")
196 tf.saved_model.save(model, "saved_model/save/")
```