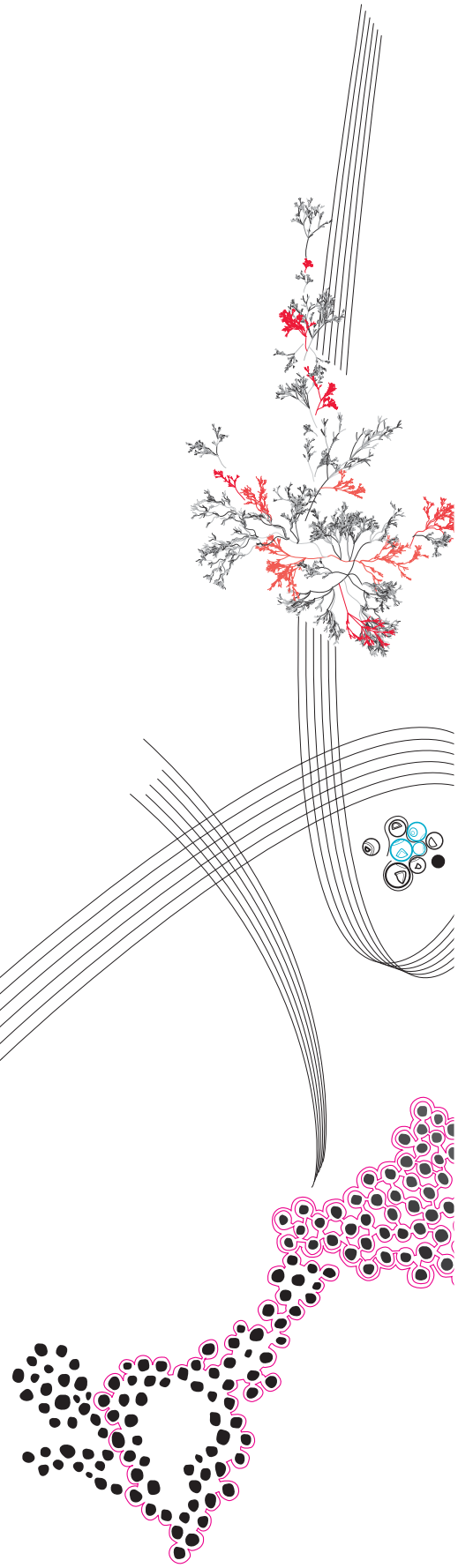


MSc Computer Science
Final Project

Weighted, Weighted and Art Found Wanting:

A Complexity-minimisation Approach for Neuroevolution-based Side-channel Analysis

Peter van der Velde

Supervisors: Luca Mariot, Andrea Continella, Marco Ottavi

January, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

Security is the basis of all modern communication. The different modes and needs of communication have given way to many new approaches to encrypting the messages within these communication channels. The need for correct ciphers to be trusted not to leak the contents of the messages has led to ever-increasing scrutiny and research into the mathematical basis of the scientific field of cryptography. However, the physical implementation of a cryptographic algorithm may leak enough information to deduce the original encryption key. Most state-of-the-art attacks against such side-channel leakages rely on the use of large complex neural networks. The design of these deep learning neural networks is in itself a complex topic. To aid in the discovery of performant neural network designs search algorithms have been designed. One such search algorithm is the NASCTY algorithm. The current implementation has difficulty finding neural network designs considered more complex and larger than a performant neural network architecture design. Extra complexity requires more training time and execution time.

In this work, we investigate the limitations of the NASCTY algorithm and propose that improvements to the efficacy of the algorithm might be possible by encouraging smaller and less complex neural network architecture designs. To achieve this end, the study introduces several changes to the underlying genetic algorithm, aiming to combat premature convergence issues and limit unneeded complexity in resulting designs. The results show that a custom fitness function influenced by the number of parameters of a neural network is a viable solution to decreasing the unneeded complexity. This approach both safeguards the size and complexity of the neural network designs but also converges to a more optimal solution. Similarly, we found that for combatting premature convergence and a wholesale stagnation of the genetic algorithm, the use of a partial random restart policy and the implementation of an adaptable ranked mutation range were sufficient in alleviating some of the symptoms. With these alterations to the genetic algorithm we found both increased reliability and predictability in the performance of the algorithm while simultaneously improving the quality of the convergence compared to the NASCTY algorithm.

Keywords: Side-channel attacks, deep learning, security.

Acknowledgments

I want to express my deepest gratitude to my supervisor, Luca, for his guidance, encouragement, and overall help during this thesis.

Additionally, I would not dream of not thanking my family extensively for their support and motivation in finishing this thesis.

Finally, I cannot forget my friends who bore with me and my ramblings about colouring pages and what now precisely the way is up on a figure, every coveted coffee break.

My thanks and appreciation to all of you for this adventure would have been bland if not for your support.

Contents

Chapter 1

Introduction

At the heart of any modern communication channel lies cryptography. Cryptography therefore is the art of ensuring that a message is both unalterable or indecipherable to all but the intended recipient. One of the first and most famous encryption techniques is the Caesar Cipher. Named after Gaius Julius Caesar, who described the cipher in his private correspondence [1] and while not being the first recorded cipher, it is a simple implementation of a substitution cipher [2]. An encryption algorithm works by transforming the message one wishes to send (the plaintext) using a cipher function in combination with a secret key. The result of this cipher function called the ciphertext, can then be sent to the recipient without (ideally) any adversarial parties able to discover the original message (plaintext) [3]. The function of this ancient encryption cipher's method was described as, taking the original plaintext and shifting all letters in the message by a secret key. Given a plaintext consisting of 'ABCD' and transforming it using the Caesar cipher with the secret key 3, the resulting ciphertext would then be 'DEFG'.

Over time these cryptographic techniques would be reinvented multiple times and slowly evolve into more sophisticated and stronger encryption techniques. With the dawn of advanced machines, cryptographic calculations would often be calculated for faster and more complex ciphertexts. A famous example from the early 20th century is the rotor-based Enigma machine developed in Germany and used in the Second World War [4]. All these advancements in turn led to the birth of computer-based code-breaking and the development and use of electronic computers for encrypting and decrypting messages [5].

Nowadays, cryptography is the base that allows our modern communication networks to function. From the private messages sent to our loved ones to banking transactions, automated door systems, or even non-public (secret) government meetings one might attend. Cryptography and its ciphers are what allows for trust between one or more parties. As long as this trust holds, the systems that depend on it hold too.

To ensure that these encrypted messages remain indecipherable and unalterable for third parties, it is necessary to have an adequately strong mathematical backbone. Therefore, a large part of research on this topic has been centred around designing strong and useful ciphers and the testing and attacking existing ciphers [6, 7, 8]. This focus is essential to ensure that these cryptographic methods are without significant flaws in their mathematical foundations.

However, the sketch of the world of cryptography as seen above only shows one side of the coin. The means of encrypting messages in truth exists in two parts, the mathematical theory and the physical implementation in hardware. Both aspects of the total security

of a system, in turn, expose different entry points for an attacker to approach.

In contrast with the days of Rome, the modern method of encrypting is done with the employment of electronic computers. Attacks on the physical side of encryption are often conducted with 'side-channel attacks', a type of attack that hopes to gain information about the internal system by analysing the information a system may leak.

As human computers were replaced by their electronic counterparts, one could no longer simply set the thumbscrews to them in order to make them give up information about the key used in their cryptographic computations.

However, even if these newfangled electronic computers may no longer be pressured into revealing sensitive information, they may still leak hints about the internal state of their calculations. These leaks can provide valuable clues for understanding and potentially compromising their cryptographic security.

These hints consist of data gleamed and recorded about the physical properties of the running system and may include but are not limited to:

- timing information [9];
- power draw [10];
- audio/visual generation [11, 12, 13];
- electromagnetic emation [14].

Attacks that make use of leaked information by the cryptographic algorithms implementations which are the result of flaws in the design, and not in the underlying theory are known as *Side-Channel Attacks* (SCA). The result of a side-channel attack is a probability vector of all possible states of the secret (sub-)key could be in.

One major new approach is the usage of deep learning techniques as prominently shown by Maghrebi *et al.* [15]. While the demonstrated attack shows impressive results and strength, it does highlight several issues still to be solved.

One weakness in the usage of machine learning techniques is the need to design a new network (i.e. the hyperparameters) for each possible implementation of a cryptographic system. Hardware design, CPU used, software implementation, and (side-channel attack) countermeasures implemented, all have a sizable impact on the design and shape of an effective neural network. The design and shape of a neural network is described as the neural network's *hyperparameters*. The search for an efficient algorithm capable of finding suitable hyperparameters for our specific problem is therefore forefront of our research.

The goal of this research will be to improve on an existing search algorithm, to overcome the current limitations of the approach. The above-mentioned search algorithm is a neuroevolution based algorithm researched by Schijlen *et al.* known as NASCTY [16]. Compared to the manually designing of neural network architecture, the NASCTY does not match or improve upon either metric. However, the automated search algorithms mentioned in Table 2.1 improve upon the needed traces before it can guess the secret key value correctly and in how many parameters it can do that.

NASCTY is an effective approach for automatically designing neural network architectures that is held back by two limitations. This does mean that the full potential of the

NASCTY algorithm has not yet been reached, and would allow for the largest possible improvement compared to the other already better performing search algorithms. The first of the limitations it suffers from is that the end results found suffer from unneeded complexity. This comes to fruition in the end results by finding results with a more complex design in number of layers –and thus parameters– then expected, and the inclusion of unintuitive design choices that result in not useful additions to the design. The second limitation run into is the tendency of the algorithm to suffer from premature stagnation. To not consider options outside the local optimum the algorithm is known to get stuck in. Thereby denying the finding of better solutions. By improving on the search algorithm the hope is that this will be a viable alternative to the manual designing of systems with its numerous difficulties and downsides and that this chosen approach will be able to beat the current state-of-the-art methodologies. To accomplish this task a research question has been constructed:

> *To what extent does a complexity-minimising approach influence the efficacy of the generated neural network architecture?*

To help answer this research question three (sub-)research questions have been constructed:

- RQ1: *How do different configurations of NASCTY hyperparameters influence the trade-off between runtime and performance?*
- RQ2: *How can we design a fitness function that effectively balances algorithmic efficacy with complexity in genotype evaluation?*
- RQ3: *What is the impact on the behaviour and performance of the algorithm from the strategies combatting early stagnation?*

Alongside this a number of experiments have been conducted to answer the main research question of this thesis. As a result of the experiments a custom fitness function has been developed to minimise complexity of the end results. This has been developed together with an anti-premature convergence strategy to combat premature convergence issues arising from the algorithm.

1.1 Thesis structure

This document has been divided into sections describing the different aspects of this research. Chapter 2 will discuss and introduce the NASCTY search algorithm and its limitations as well as all the needed background information to understand the problem space. For the background information, we will discuss the basics of cryptography, the exploitation of side-channels to break cryptographic systems, the basics of machine learning, the application of machine learning to breaking cryptographic systems via side-channels, the automation of designing these machine learning models to make solutions more easy and novel. Chapter 3 will discuss the research questions that are asked to improve the NASCTY search algorithm. The research questions are asked to overcome the limitations holding back the performance of the NASCTY search algorithm. Chapter 4 then discusses the methodology with which the research questions will be answered. Chapter 5 (hyperparameters), Chapter 6 (complexity), Chapter 7 (premature convergence), and Chapter 8 (final results) contain the results found by conducting the experiments defined previously in Chapter 4. Then with the results properly laid out, they can be discussed as to their meanings and substance in Chapter 9. Finally, Chapter 10 then brings home the findings

and discussions and places them in their correct perspective, while noting the limitations of the work and possible future avenues for improving or extending the findings of this thesis.

Chapter 2

Background

To discuss the current landscape surrounding the to-be-proposed problem space, this section will highlight several concepts directly related to our research. The concepts discussed are: cryptography, side-channel analysis, neural networks, deep learning, neural network search algorithms, and lastly the NASCTY search algorithm. Such concepts and goals explored in further writing have been provided with enough background information for them to be placed in the right context. The concepts discussed are condensed with large parts of the underlying and coalescent theories omitted for the sake of brevity.

The notation scheme in this thesis hopes to follow commonly used notation schemes for mathematical operations. We denote sets as calligraphic letters (e.g. $\mathcal{X}$). Lower-case x denotes either an element of a set $\mathcal{X}$ or a realization of a random variable denoted with Roman capital letters (e.g. X) with values over $\mathcal{X}$. Bold letters $\mathbf{x}$ denote a vector over the set $\mathcal{X}$.

2.1 Cryptography

The study of secure communication mentioned in Chapter 1, exists in two forms, *Symmetric* and *asymmetric* cryptography. Asymmetric cryptography, the newer of the two, was as a topic first published about in 1976 and described by Diffie and Hellman [17] as public key encryption schemes. Public key encryption schemes allow two parties to communicate securely without the need for one another to agree beforehand on a secret key. In contrast, symmetric cryptographic systems work with the assumption that when a message is sent to a recipient, the recipient and sender are in possession of the secret key needed to encrypt and decrypt the message.

The Caesar cipher mentioned in Chapter 1 therefore too is what would be considered a symmetric cipher, as both parties must agree on the number of times each letter must be shifted for the message to make sense.

A simple (symmetric) cryptographic system, consists of five parts:

- a plaintext message m where $m \in \mathcal{P}$ and message space P might be defined as $P = \{A, \ldots, Z\}^l$ of length l;
- the shared cryptographic key $k \in \mathcal{K}$, with the key space $\mathcal{K}$ being dependent on the used cryptographic algorithm;
- e is the cipher function used to encrypt message m with secret key k, i.e. $e : P \times K \rightarrow C$;

- the resulting ciphertext $c \in \mathcal{C}$ of encryption algorithm e, where $\mathcal{C}$ is the set of all possible ciphertexts produced by encryption algorithm e;
- the inverse of the encryption e, the decryption function d takes a ciphertext c and transforms it back into the plaintext message m with the help of secret key k, i.e. $d : C \times K \rightarrow P$.

This cryptographic encryption algorithm e can then be described as,

$$c = e_k(m) \tag{2.1}$$

The reverse of the encryption operation (decryption) can be done by transforming the ciphertext c back into the original plaintext message m.

$$m = d_k(c) \tag{2.2}$$

In both cases $k \in \mathcal{K}$, with $\mathcal{K}$ being the complete key space of this algorithm.

As an example, the Caesar cipher has a key space of 26 different possible values. Each value represents the number of steps the letters of the plaintext have to be shifted along the alphabet, so $\mathcal{K}$ in our case can be expressed as $\mathcal{K} = \{0, 1, 2, \ldots, 23, 24, 25\}$ given that each key does not overlap with another. However, in today's security landscape, 26 possible key states are no longer considered sufficient.

2.1.1 Advanced Encryption Standard (AES)

To demonstrate the concepts related to modern ciphers of key space and encryption, the *Advanced Encryption Standard* (AES) [18] will be used. The AES is as the name implies a modern standardized cipher, originally known as the Rijndael block cipher, based on *substitution* and *permutation* principles [19, 20, 21]. The encryption scheme of the AES block cipher works in four stages:

1. The first stage *KeyExpansion*, is responsible for the initialisation of the sub-keys needed in a later stage. These sub-keys –hereafter called round keys– are generated from the original AES secret key using a key schedule.
2. The second stage *AddRoundkey*, then combines the input with the first round key by XORing. XOR (i.e. exclusive OR) denoted by the mathematical operation $\oplus$ is the bitwise combination of two variables, where exclusively either one of the variables must be 1. If either both the variables are equal to 1 or 0 the output will be 0.
3. The third stage consist of a number of rounds in which in an iterative fashion the input data goes through four functions: *SubBytes*, *ShiftRows*, *MixColumns*, and AddRoundkey. The first the SubBytes function substitutes the values in the input data. This is done using a substitution table, also known as an S-box. The permutation is done in the second and third step, by rearranging the order of the data bits. The second step (ShiftRows) is completed by permutating the input by shifting the data in the matrix row-wise. The third step (MixColumns) then permutates the input by mixing the columns of the matrix. Finally, the state is XOR'ed with the round key, before being passing on the state to the first function in this round or onto the last and final round.

4. The final stage consist of an alternative cipher using the SubBytes, ShiftRows, and AddRoundkey function. Neglecting the MixColumns function used in the third stage.

Modern ciphers' key sizes in the digital age are expressed by the number of bits they can be described in. For the AES cipher, the keys are either of size 128, 192, or 256 bits. Considering a bit has but two states, 1 or 0, the respective key spaces will consist of 2^{128}, 2^{192}, and 2^{256} possible keys for the AES cipher.

2.2 Side-channel attacks

Side-channel attacks, as discussed in Chapter 1, focus on exploiting information leaked through the physical implementation of cryptographic systems rather than targeting inherent flaws in the cryptographic algorithms themselves.

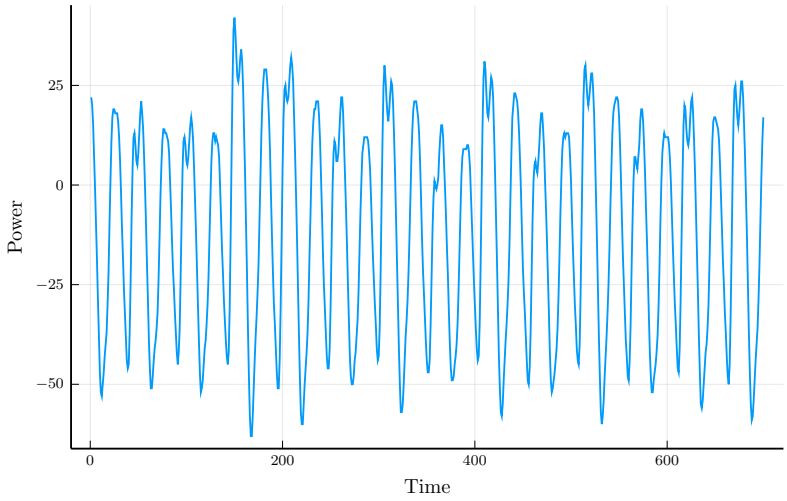


Figure 2.1: Trace $\mathbf{d}_1$ from the ASCAD dataset [22].

This leaked information, commonly referred to as traces, relates to observable characteristics of the cryptographic system during its operation.

The measurements used in side-channel attacks typically represent the variations and fluctuations in physical quantities such as power consumption, electromagnetic radiation, or timing behaviour over time. An example trace displaying the fluctuation of electromagnetic emanation measurements over time can be seen in Figure 2.1. A single trace of a dataset will be denoted from now on as $\mathbf{d}_i$ with i being the index of the trace in a dataset. All traces belonging to a certain system make up the dataset $\mathcal{D}$. There are two ways of conducting side-channel attacks: *profiling* and *non-profiling* side-channel analysis. This thesis is related to profiling side-channel attacks, and will therefore give but minimal space to background knowledge specific to non-profiling side-channel analysis.

Non-profiling side-channel analysis is a single-phase operation and tries to gleam patterns in the leaked information. Then with the information gathered about the implementation make an educated guess as to the internal state of the machine. This is the preferred form of analysis for quick direct attacks when one has limited access to the device and detailed information of the internals of a device's behaviour [23]. A profiling side-channel analysis in contrast is a two-phase attack. The *profiling phase* is where the attacker collects a dataset of traces and the used (sub-)keys from the target device when

doing cryptographic operations. The profiling traces are then analysed to create a model (profile) of the target device. The second phase, referred to as the *attack phase*, then makes use of the model created in phase one to launch an attack on the target device to estimate the secret key $\mathbf{k}^*$. Profiling side-channel analysis is preferred when the target's security mechanisms are robust enough that a more thorough attack is needed. The attacker has access to the resources and time required to model the device's behaviour and launch the more sophisticated attack.

In both cases, the goal is to find a well-performing estimation function f that can be described as:

$$f(\mathbf{d}) = \hat{\mathbf{k}} \tag{2.3}$$

where: $\mathbf{d}$ is the observed side-channel leakage; $\hat{\mathbf{k}}$ is an estimation of the sensitive information over all possible values of key space $\mathcal{K}$.

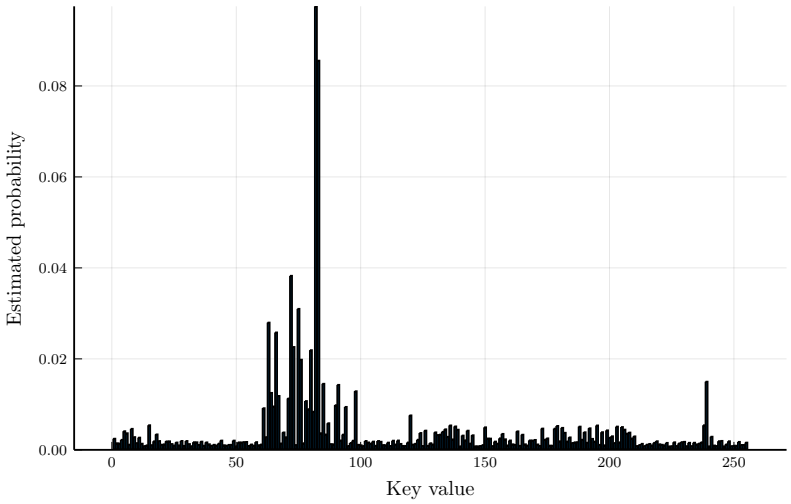


Figure 2.2: Estimated probability of sensitive key value of trace $\mathbf{d}_1$.

An example of a distribution of a trace can be seen in Figure 2.2, where we transform traces $\mathbf{d}$ into the estimation $\hat{\mathbf{k}}$ on the right. The estimation $\hat{\mathbf{k}}$ can then be sorted from the most probable to the least probable key candidate, resulting in the guessing vector $\mathbf{g}$ of size $|\mathcal{K}|$.

2.2.1 Countermeasures

Countermeasures have been invented to combat and reduce the amount of usable information gathered from the implementation. By reducing the amount of useful information contained within the traces, the effectiveness of side-channel attacks is similarly reduced. We can divide countermeasures into roughly two distinct categories: *masking countermeasures*, and *hiding countermeasures* [24]. Masking countermeasures focus on (temporarily) masking and obscuring internal values used in the algorithmic calculations, such that the underlying sensitive values do not directly correlate with possible information gained from side channels. Hiding countermeasures aim to reduce the correlation between the separate internal algorithmic calculation operations and the side-channel emissions. The implementation of hiding countermeasures can be done in several ways, most prominent by adding (random) noise to the output signal, trying to flatten the output signal, or switching around operations and adding delays in its calculations. Both countermeasures make

the mapping of side-channel measurements to the sensitive values harder to guess for an attacker. Both approaches, though they hinder any possible side-channel attack, are not considered to be significantly strong enough to rule out the possibility of any side-channel attacks succeeding.

2.2.2 ANSSI SCA Database (ASCAD)

The ASCAD (ANSSI SCA Database) [25] was set up to be a dataset common to multiple papers as a benchmarking reference containing numerous side-channel traces. Being the common denominator allows the individual papers to measure the performance of certain solutions in a reproducible manner. The database comes with multiple datasets based on different architectures and whether it uses a fixed or variable key. In this thesis, we will initially discuss the ATMEGA boolean masked AES with a fixed key dataset with traces based upon the electromagnetic (EM) radiation side-channel. This dataset consists of 50,000 profiling traces with each a corresponding subkey value and 10,000 attack traces plus subkey values. Each trace vector consists of 700 elements over time, each element representing the voltage measured at that moment in time.

2.2.3 Side-channel analysis evaluation

The evaluation of a side-channel attack's effectiveness is a bit more complex than a simple success or failure and instead tries to capture the effort needed to get the correct secret key. The guessing rank $\mathbf{g}_i = \{g_1, g_2, g_3, \ldots, g_{|\mathcal{K}|-1}, g_{|\mathcal{K}|}\}$, result of trace $\mathbf{d}_i$, ranks each guess g_i from the most likely to the least likely guess for the correct key. *Key Rank (KR)* is the index of the correct secret key $\mathbf{k}^*$ in this guessing vector.
Guessing Entropy (GE) is the average key rank index of all guessing vectors resulting from Q number of attack traces [26].

$$GE = \frac{1}{Q} \sum_{i=1}^{Q} KR(\mathbf{g}_i) \tag{2.4}$$

The average key rank is often combined with other metrics to give a more complete view of the performance of the attack.
Success Rate (SR) of order n, is the average empirical probability that the correct key $\mathbf{k}^*$ is in the first n guesses of guessing vector $\mathbf{g}$. The success rate can be formally defined as:

$$SR(n) = P(\mathbf{k}^* \in g_1, g_2, \ldots, g_n) \tag{2.5}$$

When training the neural network models for side-channel analysis oft use is made of the *categorical cross-entropy* (CCE). It is used to estimate the fitness of a model at a certain point of time and can be used to inform about the training progress. The CCE is a fitness function which compares the predicted output probability distribution $\hat{\mathbf{y}} \in \mathbb{R}^C$ of length C for each of the classes to the ground truth $\mathbf{y} \in \mathbb{Z}_2^C$. The ground truth is encoded as a *one-hot encoded vector* in which the correct output class is marked with an one and the incorrect classes with a zero. With this in mind we can define the categorical cross-entropy fitness function as:

$$CCE = -\sum_{i=1}^{C} \mathbf{y}_i \cdot log(\hat{\mathbf{y}}_i) \tag{2.6}$$

2.2.4 Machine learning

Machine Learning was first discussed on an implementation level in 1958, by Rosenblatt [27] with the introduction of the perceptron (a digital analogue for the neuron) [28]. The field of machine learning went through several cycles of increased interest and waning interest. The current revitalized interest in Machine Learning getting kick-started in 2012 (partially) by Krizhevsky *et al.* who showcased deep learning as a viable route for complex real-world problems [29]. The term machine learning broadly refers to algorithmic systems that can learn from training data to form predictions about new data.

2.2.4.1 Machine learning techniques

Machine Learning techniques can be divided into three separate categories: *unsupervised-*, *supervised-*, and *reinforcement-learning* [30]. All three categories are based on the distinct way each category learns from its training data.

1. *Supervised learning* are systems that have access to a training set (consisting of input vectors and their respective correct output vectors). Two often used applications of supervised learning are *Regression* and *Classification*. *Regression* applications aim to estimate a continuous output value (or vector) based upon an input vector. *Classification* as an application takes in an input vector $\mathbf{x}$ and tries to map this input vector to a pre-described discrete category.

2. *Unsupervised learning*, in contrast, works by training on datasets without any corresponding output values. The applications of unsupervised learning lay not in predicting any estimations of output, but in discovering patterns and relationships within the data. Both *Clustering* and *Dimension Reduction* try to find patterns in the data and find data with similar properties, cluster these data points together, and in the case of dimension reduction, combine and reduce similar data point dimensions (e.g. to reduce 5-dimensional data to 3-dimensional data).

3. *Reinforcement learning* learns from a given input vector describing the current state of the environment and the feedback the system gets. It can influence the environment it operates in and is tasked with finding optimal strategies based on the feedback it receives on its actions (that influence the environment).

Each of the mentioned machine learning techniques has numerous applications for solving specific problems beyond those highlighted above [31].

2.2.5 Classification

The task of matching the internal state, and therefore the sensitive key $\mathbf{k}^*$ used to encrypt our message to the recorded side-channel leakage measurements, our traces $\mathbf{d}$, lends itself well to classification with supervised learning. Each possible value for keys $\mathbf{k}$ can be mapped to a discrete category, and the resulting estimations of the confidence level that the correct key $\mathbf{k}^*$ is of a value corresponding to a distinct category.
There are several machine learning techniques well suited for classification problems. These techniques can be separated into two distinct categories that we call classical machine learning and the newer deep learning. Deep learning is a set of machine learning techniques related to *neural networks*. These machine learning techniques using neural networks are based on the perceptron model, which mimics the learning capabilities of the human brain.

Deep learning has become possible due to recent research findings in machine learning, providing the theoretical foundation necessary to construct deeper layered systems without the previously limiting strong diminishing returns on added layers.

Classical machine learning techniques designed for classification have proven to be more than adequate for performing side-channel attacks in numerous instances [32, 33]. These methods leverage well-established algorithms such as support vector machines, decision trees, and logistic regression, benefiting from their interpretability and ability to handle diverse data types. However, this thesis primarily discusses *deep learning side-channel attacks (DL-SCA)*, omitting extensive discussion on classical machine learning techniques.

2.3 Neural Networks

In its most basic form, a neural network can be described as a function taking in the input vector $\mathbf{x}$ of size n is an element of the n-dimensional real vector space $\mathbb{R}^n$ and the output vector $\mathbf{y}$ of size m. The neural network consists of an *input layer* matching the size of the input vector, connected to one or more intermediate layers of artificial neurons known as *hidden layers*. These hidden layers ultimately connect to the final layer, called the *output layer*, which matches the size of the output vector. The smallest computation element in this neural network, the *artificial neuron*, is, as the name implies, a mathematical analogue for the biological neuron.

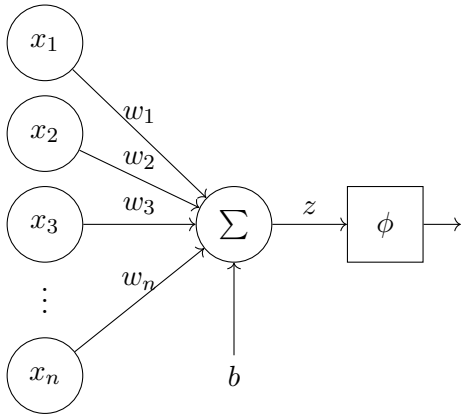


Figure 2.3: A visual representation of an artificial neuron.

An artificial neuron φ, takes in an input vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\} \in \mathbb{R}^n$ of size n, a weight vector $\mathbf{w} = \{w_1, w_2, \ldots, w_n\} \in \mathbb{R}^n$ corresponding to each input, and an inherent bias $b \in \mathbb{R}$ of the perceptron, which then get summed up into the weighted sum z. The result of the weighted sum z is then used as the input for the *activation function* $\phi : \mathbb{R} \to \mathbb{R}$. The activation function in a perceptron is a non-linear function producing the neuron's output using the weighted sum z. It determines if a neuron is to be fired depending on the inputs it has received, and allows a network to learn from its inputs.

We can describe the mathematical model of an artificial neuron φ as:

$$\varphi = \phi(\sum_{i=1}^{n} \mathbf{x}_i \mathbf{w}_i + b) \tag{2.7}$$

In an alternative formulation of the artificial neuron φ, the bias b is instead introduced as the input value $\mathbf{x}_0$ of value 1 and weight $\mathbf{w}_0$ equal to the bias.

2.3.1 Multilayer Perceptron (MLP)

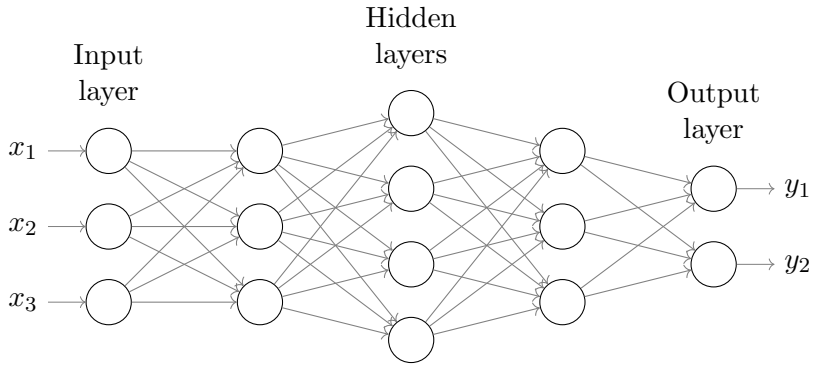


Figure 2.4: A simple multilayer perceptron (MLP) with two hidden layers.

The multilayer perceptron in its basic explanation is a neural network with an input layer, output layer, and one or more fully connected hidden layers. Fully connected layers are also known as *dense layers* and consist of a vector of artificial neurons (as seen in Figure 2.3) of which each individual neuron connects to all neurons in the previous and next layer.

2.3.2 Convolutional Neural Networks (CNN)

In 1980 Fukushima developed the *Neocognitron* [34], a neural network designed for pattern recognition tasks, powered with the help of hierarchical feature extraction. The neocognitron visual pattern recognition functions were aided by the use of *S-cells* and *C-cells*, based upon the simple and complex-cells found in the primary visual cortex [35]. Inspired by the work of Fukushima on the *Neocognitron* [34], the general concepts for Convolutional Neural Networks were devised and developed by LeCun *et al.* [36]. Convolutional neural networks allow for classifying inputs without the inputs needing to be pre-processed. This is in contrast with the multilayer perceptron networks mentioned previously, which depend on pre-processing and feature extraction to work efficiently [37].

Instead, the CNN has in addition to fully connected layers, *convolutional layers*, *pooling layers*, and *flatten layers*. A convolutional neural network is built up, working from head to tail, by several layers of convolutional and pooling layers creating a funnel structure before being flattened out by a flatten layer to one or more dense layers.

2.3.2.1 Convolutional layers

The convolutional layer of a convolutional neural network produces from input data a feature map highlighting edges, textures, and other patterns. It does this with the help of K learnable *convolutional filters (kernels)* denoted by $\mathbf{F}_k \in \mathbb{R}^{f \times f \times D}$, with f being the dimensionality of the kernel and D the depth. These convolutional filters scan the input data $\mathbf{X} \in \mathbb{R}^{W \times H \times D}$ with W, H, D being the width, height, and depth of the input data, in steps with an area of $(f \times f \times D)$, applying the dot product of that area to the kernel $\mathbf{F}_k$ filling up the output feature map $\mathbf{Y}_k \in \mathbb{R}^{W' \times H' \times K}$. A visual representation of a single instance of the above-mentioned operation can be seen in Figure 2.5. The *stride* is the number of elements in the input data that the filter will pass by before applying the above operation once more.

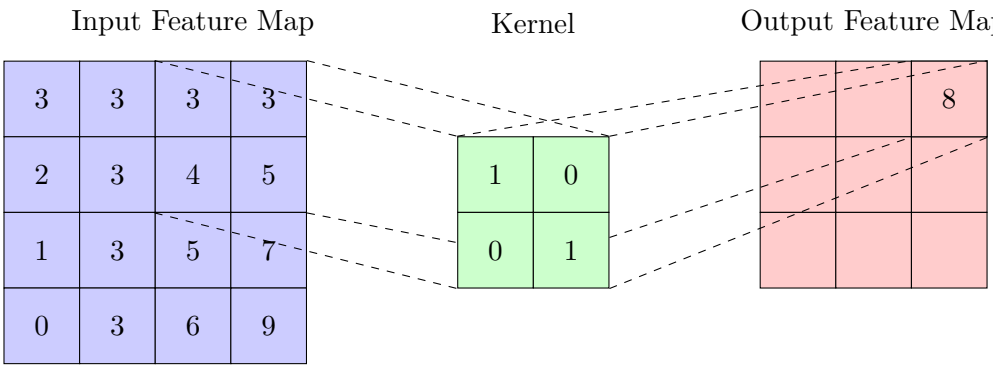


Figure 2.5: A convolutional layer with a stride of 1 and a 2×2 kernel.

With dimensional data it is possible to add padding (denoted by p) to the input data to influence the resulting dimensions of the output $\mathbf{Y}_k$. As an example, below we have a feature map $\mathbf{Z}$ to which a zero-padding of 1 has been applied:

$$\mathbf{Z}_{\text{padded}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_{11} & x_{12} & x_{13} & x_{14} & 0 \\ 0 & x_{21} & x_{22} & x_{23} & x_{24} & 0 \\ 0 & x_{31} & x_{32} & x_{33} & x_{34} & 0 \\ 0 & x_{41} & x_{42} & x_{43} & x_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

The convolutional operation that computes the output feature map $\mathbf{Y}_k$ at spatial position (i, j) as the result of convolving kernel $\mathbf{F}_k$ over input $\mathbf{X}$, can be defined as:

$$\mathbf{Y}_k(i,j) = \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} \sum_{d=0}^{D-1} \mathbf{F}_k(m,n,d) \cdot \mathbf{X}((i \cdot s) + m - p, (j \cdot s) + n - p, d) \tag{2.9}$$

Because of the inherent nature convolutional layers the spatial output dimension W' and H' of the output feature map $\mathbf{Y}_k$ do not have to match the spatial input dimensions W and H of the input data $\mathbf{X}$. The spatial output dimensions are directly influenced by the stride s, filter size f, and chosen padding p. We can calculate the spatial output dimensions using the function $g(l)$:

$$g(l) = \left\lfloor \frac{l - f + 2p}{s} \right\rfloor + 1 \tag{2.10}$$

With l being the input dimension, either W or H.

2.3.2.2 Pooling layers

The pooling layer of a convolutional neural network is a function that reduces the spatial dimensions of the input feature maps, improving the computational/ efficiency of the network and combatting overfitting. It performs this operation by summarizing areas that the pooling window of size $q \times r$ visits of the input feature map $\mathbf{X} \in \mathbb{R}^{H \times W \times D}$. Two commonly used pooling functions for this task are the *max pooling* and the *average pooling*

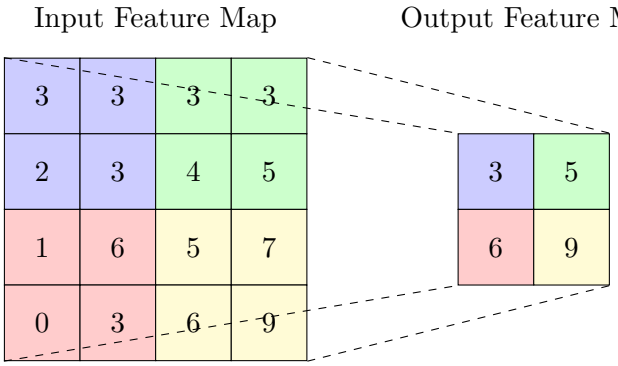


Figure 2.6: A max pooling layer with a stride of 2 and a 2×2 pooling window.

functions, taking respectively the maximum value of any elements under the window, and the average value. The pooling layers, additionally to a having a window size parameters $q \times r$, also has a stride s. To calculate the spatial output dimensions of the pooling layer we can reuse the function $g(l)$ which was also used for the convolutional layer defined in Equation 2.10.

2.3.2.3 Flatten layers

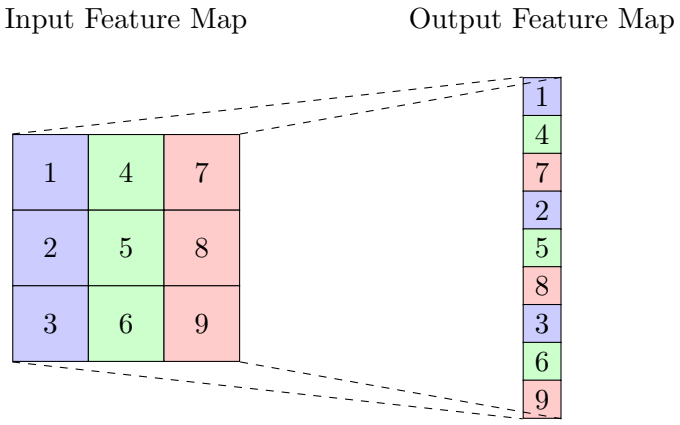


Figure 2.7: A flatten layer on a 3×3 input feature map.

A *flatten layer's* purpose is to be the bridge between the multidimensional layers (i.e. the convolutional-, and pooling-layers) and the one dimensional fully connected layers. The flatten layer is a linear transformation and takes as input a feature map $\mathbf{X} \in \mathbb{R}^{H \times W \times D}$ and returns a vector $\mathbf{Y} \in \mathbb{R}^{L}$ of length L. We can find the output length L of vector $\mathbf{Y}$ with the use of the formula:

$$L = W \cdot H \cdot D \tag{2.11}$$

2.4 Deep Learning Side-channel Analysis (DL-SCA)

Making use of deep learning techniques to analyse side-channel data to extract sensitive information of the underlying system is known as deep learning side-channel analysis or in short DL-SCA. Deep learning based machine learning only gained momentum in the space

of side-channel analysis after circa 2016 [15]. With this paper Maghrebi *et al.* showed that the deep-learning based attacks outperformed state-of-the-art attacks on both protected and unprotected AES implementations. A protected AES implementation, is any AES implementation employing side-channel analysis countermeasures. This ability of machine learning techniques to learn patterns and relationships from data without prior instruction has shown to be successful in this context.

In the few years since then many interesting findings were found in the scientific realm of DL-SCA showing the effectiveness of deep learning aided attacks and need for deep learning specific approaches to side-channel attacks. Since then:

- Deep learning has been found to be an effective attack, able in overcoming protected targets [38]. The deep learning based side-channel attacks have been shown to be efficient in defeating (combined) countermeasures [39].
- Various machine learning concepts and their usage in deep learning side-channel analysis has been evaluated [40].
- A benchmarking dataset has been introduced with the goal of creating a standardized machine learning oriented benchmark [22].
- Zaid *et al.* showed the effectiveness of CNN's with small neural network architectures in breaking targets selected from benchmarking datasets [41]. Wouters *et al.* improved and added to both the proposed methodologies and CNN architecture of Zaid *et al.* designing a CNN architecture much smaller (on average 50% smaller) while retaining a similar performance [42].
- Stronger theoretical foundations of DL-SCA have been developed, and new countermeasures have been researched.

2.5 Neural Architecture Search (NAS)

The architecture of a deep learning neural network is comprised of *hyperparameters*. The hyperparameters of a machine learning neural network describe, if and how many convolutional layers are used, the form of the convolutional and pooling layers, the number of dense layers, the number of artificial neurons, the used activation functions, etc. The selected hyperparameters of the deep learning architecture are paramount to its performance. However, earlier papers about deep-learning-assisted side-channel attacks did not mention how the hyperparameters were selected for their design [43]. The hyperparameter selection methods since then have become transparent and methodologies have been invented and researched for the design of deep learning architectures [41, 42]. However, these methodologies for designing neural network architectures require knowledge of the underlying dataset and does not necessarily extend easily to other datasets [44].

To select the right hyperparameters for a specific problem space requires from researchers to draw from their domain knowledge, experience, and empirical observations. While such networks are drawn from a researcher's skill in selecting hyperparameters, individuals lacking experience with the design process of a neural network may struggle to find "optimal" solutions.

The large amount of options available in hyperparameter choice, make for a search space that is too large for an exhaustive search. However, limiting ourselves to a limited search space for more predictive results, random or grid search might leave out novel and possible more performant neural network designs.

Table 2.1: Comparison of performance and parameter size of different hyperparameter tuning methods on the synchronized ASCAD [25] dataset.

Model	Type	Traces to obtain mean key rank 0	Parameters
ASCAD[22]	MLP	410	393 936
ASCAD[22]	CNN	480	66 652 444
AutoSCA[45]	MLP	129	478 656
AutoSCA[45]	CNN	158	54 752
Zaid *et al.*[41]	CNN	191	16 960
Wouters *et al.*[42]	CNN	$\approx$ 200 [46]	6 436
MetaQNN[46]	CNN	202	79 439
MetaQNN[46]	CNN	242	1 282
InfoNEAT[47]	InfoNEAT	130	15 107
NASCTY[16]	CNN	314	10 470

This is where *Neural Architecture Search (NAS)* steps in. NAS algorithms wish to automate the process of designing neural networks. Neural architecture search algorithms then too also aim, if not to match existing manually designed neural networks, to outperform them. In other words, NAS algorithms search through large search spaces consisting of possible neural network hyperparameters looking for combinations performing well in a selected problem. This comes with the added benefit of considering (unconventional) hyperparameter combinations normally not considered, and the automation of finding solutions for variable problems without having to consistently redesign each network to tailor it to the personal needs of each problem.

There are currently several approaches for designing neural architecture search algorithms:

- *Reinforcement learning-based (RL)*, is an agent-based search algorithm. The agent interacts with a DL-SCA environment, the agent hands over a possible architecture to the environment, which in turn train the design and ascribes it a performance metric. With this information the agent in turn explores the possible hyperparameter combinations in an iterative fashion, all the while learning from the rewards it has received. An adapted implementation of this approach for automating DL-SCA architectures can be found in MetaQNN [48, 46].

- *Bayesian Optimization (BO)* tries to model an optimum of a possible target function, the *objective function*, in the minimum number of iterations. It does this by creating a surrogate function to approximate the unknown objective function. The second part of the algorithm is the *acquisition function* which decides where in the sample space to sample next. It then in an iterative fashion refines the surrogate function to reach the optimal solution. An implementation of the Bayesian optimization algorithm for finding DL-SCA algorithms is AutoSCA [45].

- *Evolutionary algorithms (EA)*, are a collection of algorithms based on ideas from biological evolution. *Genetic Algorithms (GA)* for example is based on the evolutionary concepts related to genetic mutation and selection, where each genome represents a possible solution. The complete set of evolutionary algorithms and other related biological-inspired algorithms is a rather large one with many different algorithms each having their strengths and weaknesses. For their large number, an exhaustive overview of all evolutionary algorithms is considered as out of scope for this thesis.

Two implementations of evolutionary based neural architectural search algorithms, both using genetic algorithms, are:

- InfoNEAT [47] is a neural architecture search tailored for side-channel analysis based upon the *neuroevolution of augmenting topologies* (NEAT) [49] algorithm. The InfoNEAT algorithm is a genetic algorithm updating both the topology and weights, developing a neural network for each of the separate neural network output classes, all stacked together as an input for a logistic regression model.
- NASCTY [16] in contrast is a genetic algorithm based NAS aiming to only find and discover performant deep learning architectures of CNN and MLP variety. This thesis makes use of and discusses the inner workings of the NASCTY algorithm in more detail in further on in the next section.

The recorded performance of the above-mentioned neural architecture search algorithms on the non-desynchronised ASCAD dataset is shown in Table 2.1. This additionally, shows the size of the resulting neural network architecture in the number of parameters belonging to the designs.

2.6 Neuroevolution to attack side-channel traces yielding convolutional neural networks (NASCTY)

Schijlen *et al.* [16] describes a novel approach for designing neural networks for side-channel attacks with the use of a genetic algorithm (GA), called *neuroevolution to attack side-channel traces yielding convolutional neural networks* (NASCTY). The NASCTY algorithm reported in Algorithm 1 follows the different stages of a genetic algorithm, i.e. *Initialisation*, *selection*, *crossover*, *mutation*, and *replacement*. The goal of the genetic algorithm is to solve complex problems by simulating the process of natural evolution, where a randomly initialized set of possible solutions –called the population– evolves over successive generations by selecting promising solutions, combining, and mutating these new solutions to iteratively produce increasingly better ones.

Algorithm 1 The NASCTY algorithm [16].

```
procedure NASCTY(max_gen)
    D_train, K_train, D_val, K_val ← sample(ASCAD_data)
    pop ← init_population()
    while gen ≤ max_gen do
        evaluate_fitness(pop, D_train, K_train)
        parents ← tournament_selection(pop)
        offspring ← produce_offspring(parents)
        pop ← parents ∪ offspring
    return genome in pop with the lowest fitness
```

The five individual phases of the NASCTY genetic algorithm are detailed in the subsections below. These subsections will delve into each phase, what different configurations have been considered, and how each phase works and transforms the population.

2.6.1 Initialisation

The first stage of the algorithm is the initialisation stage. In this stage the initial *population* of potential solutions is generated. These come in the form of a set of blueprints of neural

network architecture hyperparameters called *genomes*. The shape and values for genomes used in the NASCTY algorithm are specified in Table 2.2.

Table 2.2: The values for each of the genome hyperparameters in the NASCTY algorithm.

Parameter name	Possible values
Number of convolutional blocks	$\{x \in \mathbb{N} \mid 0 \leq x \leq 5\}$
Number of dense layers	$\{x \in \mathbb{N} \mid 1 \leq x \leq 5\}$
Number of convolutional layers	$\{x \in \mathbb{N} \mid 2 \leq x \leq 128\}$
Filter size	$\{x \in \mathbb{N} \mid 1 \leq x \leq 50\}$
Batch normalization layer	True, False
Pooling type	Average pooling, Max pooling
Pool size	$\{x \in \mathbb{N} \mid 2 \leq x \leq 50\}$
Pool stride	$\{x \in \mathbb{N} \mid 2 \leq x \leq 50\}$
Number of dense neurons	$\{x \in \mathbb{N} \mid 1 \leq x \leq 20\}$

The approach for generating the initial population employed by NASCTY is *random initialisation*, where each of the genomes are generated with random values within the limitations as set in Table 2.1 as to ensure diversity in the initial population. Other possible initialisation strategies could have been: *Heuristic-based Initialisation* where genomes are generated using domain specific heuristics to ensure a possible better performing initial population, and *Population Seeding* where the initial population (in part) consist of known performant (partial) genomes.

2.6.2 Selection

The selection strategy for selecting the *parents* of future offspring in use by NASCTY is a *tournament selection* with three participants. In a tournament selection, a number of (in our case three) participants is randomly sampled from the population. These three participants then duke it out, the one with the best *fitness score* winning the tournament. The fitness score is determined by constructing the blueprints (genomes) into the neural network models called *phenotypes*, then evaluating these phenotypes by training them on the training dataset for ten epochs followed by an evaluation using a *fitness function*. The fitness function of a genetic algorithm evaluates the effectiveness of a potential solution in satisfying certain goals.

NASCTY makes use of the *categorical cross-entropy* (CCE) fitness function to estimate the genomes' performance. With the winner of this tournament decided, this process then is repeated till all the required slots of possible parents are filled up.

2.6.3 Crossover

With the parents selected, the offspring can be generated using crossover. The crossover stage performs the generation of two genomes as the result of its parents genomes combination. An example NASCTY genome as depicted in Figure 2.8 could be one of the two parents of the offspring. Both new solutions carry a part of the selected genes of the parents from the other. Each of the new genomes the having the inverse selection of genes from the parents. For the crossover strategies two options have been evaluated for the NASCTY algorithm:

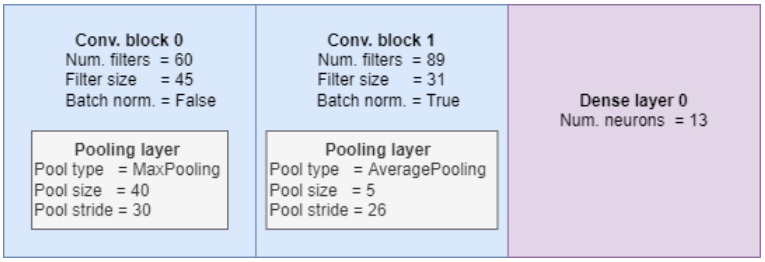


Figure 2.8: A NASCTY genome encoding example (sourced from Schijlen *et al.* [16]).

1. *one-point* crossover, where a single point is selected in the genome after which the genetic material is swapped in both parents. This is done for both the convolutional blocks and the dense layers block separately.

2. *parameter-wise* crossover, in which the offspring consists of the parameter-wise random selection of genes from both parent genomes.

The NASCTY algorithm ended up using the one-point crossover strategy, promising a more consistent run of the algorithm [16].

2.6.4 Mutation

Next is the mutation stage in which the offspring off the crossover stage are mutated. NASCTY makes use of three possible mutation strategies each with a uniform probability of selection [16]: adding randomly initialized blocks and layers, removing blocks and layers from the genome, or mutating the hyperparameters of the genome through *polynomial mutation*. All the genomes hyperparameters have the chance of being modified by the polynomial mutation algorithm with a probability of $\frac{1}{n}$ with n representing the total number of hyperparameters in a given genome.

The polynomial mutation is a mutation algorithm that transforms an input variable $x \in \mathbb{R}$ into the mutated output variable x' by modifying it with mutation function δ. Mutation function δ is influenced by two variables: uniform random number $u \in [0, 1]$, and mutation distribution index η which controls the shape of the distribution. Higher values of η result in smaller perturbations of the input value x. The mutation function δ is defined as:

$$\delta = \begin{cases} (2u)^{\frac{1}{\eta+1}} - 1, u < 0.5 \\ 1 - (2(1-u))^{\frac{1}{\eta+1}}, u \geq 0.5 \end{cases} \tag{2.12}$$

Allowing the output variable x' to be defined as:

$$x' = x + \delta \cdot (x^{max} - x^{min}) \tag{2.13}$$

Where x^{min} and x^{max} represent the lower and upper bounds of the variable x.

2.6.5 Replacement

The last step is replacement, where the population of the algorithm is replaced by combination of parents and offspring while maintaining the population size. From here the algorithm returns to selection step. The steps are executed again in an iterative fashion

until either the maximum generation is upon us or another goal has been reached. These alternative goals can be made of arbitrary requirements. As an example, the genetic algorithm can be made to stop searching if a certain fitness threshold has been found in a genome of the population.

2.6.6 Limitations

The NASCTY algorithm has been shown as an effective approach to break the fixed key ASCAD dataset. However, the NASCTY algorithm has not been able to surpass the state-of-the-art neural network models and neural network search algorithms in either performance or the *complexity* of models themselves as seen in Table 2.1. The performance of a model is measured as the number of needed traces to obtain a key rank of 0 over 100 folds. The complexity is a measure of the size of a neural network, namely the number of layers and hyperparameters. Commonly in other works as in this case the complexity measures the total number of parameters of models.

From these complications a number of limitations have arisen that may hold back the performance of the NASCTY algorithm:

1. Complexity, the NASCTY algorithm does not sufficiently discourage the generation of models with redundant model complexity [16]. The algorithm may be prone to adding unneeded layers and unintuitive layer components. Examples given by Schijlen *et al.* point to possible: unneeded dense or convolutional layers, unintuitive large numbers of convolutional filters, and pooling layers with strides larger than the inputs' sizes.

2. *Local optima*, the NASCTY algorithm stagnates in the best runs of the algorithm for desynchronisations 0, 10, 30, and 50 earlier than what might be desired. In these best runs they stagnate after around the 25, 35, 20, and 10 generation mark. This implies that these runs get stuck in what are called local optimum. A local optimum is a point where the evaluation function reaches a minimum or maximum value higher than nearby points in the region but, which might not be the highest or lowest value overall. The premature loss of diversity in a search converging on a local optimum is a common problem of genetic algorithms [50, 51]. This problem is known as *premature convergence*. To combat this problem two approaches exist: increasing mutation rate and mutation range, or by maintaining higher diversity of the population.

Chapter 3

Research goal

With this research we hope to contribute to the existing NASCTY framework and discover possible design choices or limitations that may hold back the performance of the system. The NASCTY algorithm does not beat or match the current state-of-the-art systems, the two earlier mentioned limitations in Section 2.6.6 are two hurdles which we aim overcome.

To answer this problem we have formulated the following main research question:

> *To what extent does a complexity-minimising approach influence the efficacy of the generated neural network architecture?*

Further, we have devised a set of three sub-questions to answer our research question. In order to answer these three sub-questions and the main question, there is a need to do quick iterative tests and changes to the underlying genetic algorithm of NASCTY. Unfortunately, a single run of the NASCTY algorithm might take up to multiple days of computing time. At this speed each iterative change to hyperparameters of the NASCTY genetic algorithm will take too much time, before any conclusions can be drawn as to the effectiveness of the changes within a reasonable time span. This brings us to our first sub-research question:

1. RQ1: *How do different configurations of NASCTY hyperparameters influence the trade-off between runtime and performance?*

With the knowledge gained by having an arena for the experiments, the questions related to the limitations of the original NASCTY algorithm can be answered. This brings us to the first limitation of NASCTY, complexity. The genetic algorithm has the tendency to not discourage the selection of possible solutions whose genomes contain unneeded complexity. Unneeded complexity in this context does not solely refer to a surplus of the number of model parameters but also to the more general addition of ineffective layers or layer components.

However, we assume that we are able to capture the unneeded complexity of a neural network in a function. With this assumption the next sub-research question becomes:

2. RQ2: *How can we design a fitness function that effectively balances algorithmic efficacy with complexity in genotype evaluation?*

Solving the above question will give us answers to the first of the two limitations.

The third research question is mainly concerned with the second of the two limitations, local optima, or the observed early stagnation seen in the observed runs of the algorithm. In this research three strategies are considered: lowering the distribution index η as to increase the mutation range –which in turn might be beneficial in escaping local optima–,

implementing an adaptive mutation rate which dynamically adapts the mutation rate of a genome based upon current population metrics, and the implementation of a partial reset policy for when the genetic algorithm's performance stagnates, replacing the population partially with new randomly generated genomes. Chapter 4 stipulates the stagnation strategies considered in this research and in our last sub-research question:

3. RQ3: *What is the impact on the behaviour and performance of the algorithm from the strategies combatting early stagnation?*

Chapter 4

Methodology

To answer the main research question and the stipulated sub-questions, an overview of all methodologies used has been set up. Section 4.1 describes the global overview of the research design approach used for this thesis.

Section 4.4 then goes into detail of the used research hardware and the total computational time needed. The global outlines to where the experimental raw data originates from and what forms of pre-processing the data has been exposed to is discussed in Section 4.2. With the data defined, Section 4.3 reviews the NASCTY genetic algorithm that serves as the base upon which most of our experiments are conducted. Then, Section 4.5 lays out the experiments in larger detail for each of the research questions. For each research question, a separate overview is written. Each overview details: the goals of the experiments; how each of the experiments contributes to achieving that goal; presenting a hypothesis regarding the expected results; and the parameters that define the boundaries of the experiment. Finally, we surmise this section and the individual experiments in Table 4.4. The techniques used for analysing the data from experiments are discussed in Section 4.6. Section 4.7 then discusses the ethical considerations made when designing and executing this research. Alongside the methodology framework described, the changes made and the limitations of the methodology are discussed in Section 4.8.

4.1 Research design

This research aims at improving upon the NASCTY algorithm by investigating strategies to overcome its limitations. It makes use of experiments in order to investigate the effectiveness of proposed improvements to the NASCTY genetic algorithm.

The research is divided up into four parts. The first three of these parts are concerned with modifications to the existing genetic algorithm. In short, the alterations investigated and experimented with are:

1. Genetic algorithm hyperparameters (discussed in Section 4.5.1) with the goal of both reducing the run time of the algorithm while maintaining the quality of the results.

2. Custom fitness function (discussed in Section 4.5.2), with the aim of reducing the complexity of the end result.

3. Anti-premature convergence strategies (discussed in Section 4.5.3), it lays out the methodology used for evaluating anti-premature convergence strategies with goal of reducing early stagnation.

Each of the experiments modify the genetic algorithm, and have their changes applied to the NASCTY algorithm. These modifications do not carry onto the following experiments unless stated explicitly. This is done with the intent of being able to compare each change to the baseline genetic algorithm without running into the possibility of previous modifications influencing current results.

Lastly, the most suitable of these modifications are collected and all applied to the NASCTY algorithm. Section 4.5.4 concerns itself with the methodology applied for evaluating the modified genetic algorithm.

4.2 Data source

All the research done in this thesis sources its data from the ASCAD [22] dataset. The ASCAD dataset serves as a publicly available side-channel analysis benchmark available in three flavours:

- The ATMEGA fixed key AES dataset, containing 60,000 traces of 700 data points from a single secret key. Of these 60,000 traces, 50,000 traces are marked as training traces and a 10,000 are marked as attack traces. This dataset is available in three distinct desynchronisation levels 0 or synchronised, 50, and 100. Higher levels of desynchronisation allow the neural networks to be evaluated against jitter by introducing unpredictable variations in the timing [22]. Even though, more and different levels of desynchrisation are able to be created from the raw traces, this research will limit itself to the pre-generated 0, 50, and 100 levels.

- The ATMEGA variable key AES dataset, containing 300,000 traces of 1400 samples from multiple different secret keys. These 300,000 traces are split up into a 200,000 traces large profiling dataset and a 100,000 large attack dataset. This dataset is available with three levels of desynchronisation 0, 50, and 100. From these three versions only the non-desynchronised dataset is used in this thesis.

- The STM32 variable key AES dataset, containing 1,000,000 traces of 15,000 data points from random secret keys. This dataset is part of the ASCADv2 version and was introduced to be a more challenging public benchmarking dataset to defeat [52].

This research limits itself to the two datasets introduced in ASCADv1 as it is the most common of the used datasets in other related research. An overview of the performance of other research on these datasets alongside the performance of the end product of this thesis can be seen in Appendix A. Assume for the experiments that unless specified explicitly otherwise that for the training and evaluation the non-desynchronised fixed key date set is used. Each additionally used dataset would at least double the time needed for collecting and evaluating the results of experiments.

Any training of neural networks is done on balanced datasets as to avoid overfitting on the dominant classes [53]. Each profiling dataset is split up into a training and validation traces dataset to a ratio of 90% to 10%. The balanced datasets are created by randomly undersampling. The act of undersampling is done by reducing the size of each class to the size smallest class. In the case of the fixed key dataset this will reduce the number of training and validation traces available from 50,000 to 35,584 training traces and 3,840 validation traces. Similarly, for the random key dataset the training and validation traces are reduced from a total of 200,000 to 162,048 training traces and 18,176 validation traces.

None of the input data will undergo any data pre-processing activities before or during any of the experiments.

4.3 Base genetic algorithm

This section means to serve as a short review of the NASCTY algorithm. As the algorithm has previously been introduced in depth in Section 2.6, this section will mainly mention the algorithm from a global overview. The NASCTY algorithm is a genetic algorithm whose phases can be surmised as:

1. Initialisation, the initial population consist of 100 randomly generated genomes;
2. Selection, selection is done by means of a three-contestant tournament selection algorithm that bases its individual contestant scores on the fitness score of a genome after ten epochs;
3. Crossover, one-point crossover between the parents;
4. Mutation, three equal chance mutation strategies: removal, adding, and polynomial mutation with mutation distribution index η strength equal to 20.
5. Replacement, the population is replaced by an equal split of parents and mutated offspring.

These five phases are repeated till either an artificial limit or the maximum number of generations is reached. When the final generation has been evaluated, the end result will be the genome with the best fitness score.

The genome structure that the genetic algorithm uses –as defined in Table 2.2– is not modified in any of the experiments and should remain a constant. The genome structure remaining constant allows for an easier comparison of the genetic algorithms and removes a possible unknown. Lastly, all genomes' phenotype when evaluated are initialised with the same random seed as to inhibit the influence of random initialisation on the fitness score.

4.4 Experimental setup

For conducting all the experiments access has been granted to the high performance compute cluster of the University of Twente. All systems and experiments are implemented and run on top of the Julia [54] programming language (v1.10) with the help of the Flux [55, 56] library for enabling machine learning. All the experiments will be run on a Dell R750xa server with two Intel Xeon Silver 4314 processors, 256 gigabytes of working memory, and four NVIDIA L40 graphics processor units. However, as the experiments are mostly making use of the GPU for evaluating the genomes, each experiment was limited to a single core, four gigabytes of ram, and a single GPU. Lastly, as per cluster user limitations access was only granted for two graphics cards most experiments required multiple days to get a full and reliable result. The total time used to conduct all the experiments comes down to 3,140 processing hours.

4.5 Experiments

This section discusses the experiments in a global overview. The evaluation and deeper in depth exploration of the experiments and their results are discussed in their respective chapters.

4.5.1 Hyperparameters

To answer the first research question we need to find hyperparameters for the genetic algorithm that satisfy that best satisfy the requirements of balancing the trade-off between runtime and performance. In order to be quickly able to iterate on research while still being able to translate the findings four hyperparameters have been identified as of being of interest: evaluation epochs; an early termination policy; training data size; and population size. From these four different experiments have been set up. Section 4.5.1.1 investigates the viability of reducing the number of epochs that a possible solution is trained before evaluation and what the impact it has on the accuracy of the evaluation.

The ability of an early termination policy in reducing training time by cutting short non-promising is explored in Section 4.5.1.2. Section 4.5.1.3 then in turn investigates the different training data partition sizes in order to find the configuration that reduces the runtime while roughly maintaining the same evaluation accuracy. Lastly, Section 4.5.1.4 investigates the possibility of reducing the population size for later experiments and if reducing the population size might be a more optimal choice when running the algorithm. The findings of this review will be instrumental in answering our next questions.

4.5.1.1 Required number of evaluation epochs

The first experiment was concerned with the number of epochs a possible solution had to be trained before an accurate estimation of the solution's final fitness score compared to other possible solutions could be made. Each of the genomes in the population used in the genetic algorithm must be evaluated before or during the selection stage. This evaluation phase calculates the fitness of the potential solution by training the genome's phenotype for an x amount of epochs and calculating the fitness score on the validation dataset. From this arises a question, if the fitness function's goal is to predict the quality of the end solution –a genome whose phenotype has been trained for 50 epochs–, what number of training epochs are required for accurately predicting which genome would win in the tournament selection process? By finding the minimum required number of training epochs the aim is to reduce the number of training epochs set (i.e. ten epochs) by the NASCTY algorithm.

With the aim of finding the minimum required number of training epochs an experiment has been set up. Firstly, a collection of 500 genomes was generated. The larger number of genomes should counter the possible low success rate of a randomly sampled genome. A genome is considered unsuccessful if the fitness score during training does not improve. This collection of genomes will serve as the sampling population. These genomes follow the structure as set out in Table 2.2. The structure and values of a genome is randomly generated. Each of the genomes is trained for a number of epochs on the fixed key non-desynchronised ASCAD dataset. Then, every epoch the fitness score is calculated. The fitness score is calculated by applying the CCE fitness function (see Equation 2.6) on the validation dataset.

Lastly, the collected data is analysed by calculating the *selection accuracy* at each

epoch to the final recorded epoch. To do this calculation all the possible combinations of contestant that may occur enter a three contestant-sized tournament selection. That means each possible combination of input genomes and their fitness scores will enter our tournament selection and compare their winners at a certain epoch with those found at the 50th epoch.

The choice to limit the training time to 50 epochs was done for two reasons. Firstly, this limitation allows for the quality of the neural network models to shine through, if the neural networks had been trained with too few epochs the performance capabilities of the design might not be clear enough. As training a neural network design for more epochs allows the design to more-so if able diverge from the starting point's fitness score. The second reason for limiting the training of the neural network designs to 50 epochs is the time constraints one must consider. Any increase in training time in epochs also reduces the number of test samples we can acquire and process.

We expect that genomes trained for ten epochs might give us the most ideal trade-off in our situation, for accuracy per needed time to train. This is in line with the chosen number of epochs used in the selection process for calculating the fitness of a genome used in the original NASCTY paper [16].

4.5.1.2 Early termination

The second experiment's goal is to discern if it is possible to predict if a solution is a non-promising solution. Thereby, reducing the time needed for validation by excluding non-promising solutions from further evaluation. A solution is promising if its fitness value decreases over time. Solutions whose designs allow not for enough of generalization, and therefore suffer from overfitting the model on the training dataset. These possible solutions are uninteresting for the genetic algorithm, as it is solely focused on finding the best possible solution for a specific problem.

For this experiment an additional 2000 random genomes will be trained while being evaluated on a validation dataset every epoch. Both the training traces and validation traces come from the non-desynchronised fixed key dataset. The number of epochs these genomes will be trained for during evaluation is fully depended on the found results in experiment I. Because of the random nature of selecting the genomes, the number of well performing genomes will be but a fraction of the evaluated genomes. For this reason the number of genomes evaluated has been enlarged to 2000 genomes as to include enough well performing genomes. From each genome the training loss is recorded every epoch.

Then these recorded training values serve as the base from which the fitness thresholds come. These determine if a genome is considered to be of a promising nature. If a possible solution in an epoch has been weighted and art found wanting, its evaluation will be terminated prematurely before reaching the full standard evaluation time in epochs.

Two strategies for calculating the weights for early termination by fitness threshold are considered: a *conservative* and an *optimised* approach. The conservative approach considers all data points that improve over time more than the stated threshold. This approach, while possibly training more unpromising solutions for longer, most likely does not miss out on any possible solution outside the most common validation fitness score progressions. The second so-called optimised approach, optimises for including most of

the promising solutions and reducing the training time by disregarding outliers.

To analyse the results, the recorded genome data points will be subjected to the proposed fitness thresholds. These findings will be the basis of calculating the runtime savings and loss of promising solutions for each approach.

Our expectations for this experiment are a noticeable improvement in the speed of the genetic algorithm. We believe that for the conservative approach the number of missed interesting solutions is close to zero and the saved number of epochs needed for training all the selected genomes should hover around a 10% for any of the different threshold categories. This prediction of a 10% reduction stems from the expectation that from the starting point the longer that one trains the neural networks the more they will diverge from the starting point in both of the extremes directions. For the more optimised approach we expect a reduction of needed epochs to be trained of 25% across the lower of the thresholding categories while for the highest thresholding category we expect a reduction of 30%. The higher expectations here stem from the belief that with ignoring the outliers the remaining solutions will diverge in a more extreme way, thereby discounting a larger number of solutions from further training. This expected higher reduction in needed training epochs comes with the trade-off of a 5% chance on missing potential interesting solutions. This is based upon an assumption that around a 5% of measurements are outliers.

4.5.1.3 Training data partition size

The third experiment explores the viability of restricting the access to training as to reduce the time needed for evaluating a possible solution. By limiting the amount of data used in each epoch, the hope is that the time needed for training and in turn evaluating the genomes' phenotype will be reduced without significant loss of fitness performance. This is done by shuffling the complete training dataset before each epoch and selecting a limited partition of it. The size of the partition is chosen before the training of the solution and remains unchanged during the complete evaluation phase of a genome.

The finding of a possible faster, although still reliable evaluation phase by limiting the amount of training data each epoch will be done using a grid search. The variables considered for this grid search for the training data partition size are displayed in Table 4.1. The considered partition sizes in this design where broadly chosen to represent a mostly gradual increase by doubling the previous size. The smallest possible partition size is 100 data points, consequent options are roughly double the previous value until reaching the maximum possible data points of 35 6000.

Table 4.1: Parameters used in experiment two on the impact of training dataset size on the performance of the fitness evaluation.

Parameter name	Values
Model	Wouters *et al.*[42], NASCTY[16]
Partition size	100, 200, 500, 1 000, 2 500, 5 000, 7 500, 10 000, 20 000, 30 000, 35 600

Twenty random genomes' phenotype is evaluated for 50 epochs. Twenty genomes

should be enough of a spread to gain a good estimation of the impact on different sized and performing random genomes. The fixed key non-desynchronised ASCAD dataset is used for training and evaluating the genomes. Each genome is trained for 50 epochs, allowing for the time and space needed to diverge from its origin. At every epoch trained a snapshot is created of the containing the training fitness and the training time. Next, for each partition size parameter as defined in Table 4.3 the process is repeated.

Once all configurations have been evaluated, the time metrics are collected and analysed. These metrics give insight on the impact the partition size has on the time needed to train a possible solution. This is followed by a consultation of the fitness score metrics to gain an insight on the impact over a 50 epoch long time window.

We expect that for this experiment that the training time per epoch will be directly changed to the size of the partition in relation to the full training dataset. This is done with the expectation that the quality of the evaluation will change in related amounts compared to the usage of the full training dataset each training epoch.

4.5.1.4 Population size

The fourth and final experiment of the first research question's goal is to see if it is possible to reduce the population size of the genetic algorithm without impacting the convergence, diversity, and coverage too much. Taking the population size of 100 as baseline, with what sizes of population can there still be made reliable enough predictions of the networks' performance, compared when the genetic algorithm runs with a population size of 100?

Table 4.2: Parameters used in experiment three on the performance impact of population size used in the NASCTY algorithm.

Parameter name	Values
Population size	10, 20, 30, 40, 50, 60, 70, 80, 90, 100

This is the first experiment to build on top of the NASCTY genetic algorithm. The modifications made to it pertain to the population size used during operation.

The considered population size values shown in Table 4.2 are all lower than the default configuration of 100. These options were chosen with the intent of reducing runtime. To achieve that aim the values run back from the default of 100 to the minimum extreme in ten sized steps.

Each configuration is run for a total of ten times. This should allow for strong enough results not marred too strongly by the inherent random nature of the algorithm. Each run 'run' of the algorithm consist of reading in the dataset before following the phases of the algorithm as set out in Chapter 4 for a maximum of 50 generations. The dataset used in this experiment is the fixed key non-desynchronised ASCAD dataset.

At the start of every generation including the initialisation moment a snapshot is taken of the current population. This includes for each of the genomes in the population its fitness score.

When all runs have been completed and the data collected, an analysis of the convergence, diversity, and coverage is made. The convergence metrics give an indication of the progress the system is making upon converging on a (local) optimum. The diversity metrics are used to gain insight into the how homogeneous the population is. Lastly, the coverage metrics are meant to show the possible solutions explored in the search space. The metrics used for the convergence, diversity, and coverage are based in turn upon the:

- Convergence, the lowest fitness score in the population;
- Diversity, the mean distance between the individual genomes in the population;
- Coverage, the number of uniquely discovered and evaluated genomes.

Our expectations of this experiment are that for each decrease in population size, a small decrease in effectiveness will be found in the convergence of the genetic algorithm. For the diversity of the population each generation, we believe the population will indeed see a slightly reduced diversity with each increase in the limitations imposed upon the population size. In contrast, we expect the coverage to be influenced in line with the decreases in the population size. Given a population size, we expect that each generation a similar amount of newly minted genomes should be explored and evaluated.

4.5.2 Complexity

The second research question will be answered by enumerating the different possible indicators of unneeded complexity: parameter count, unintuitive architectural design choice penalties, or a combination of the two. For this matching custom loss functions for the evaluation of the different solution in the selection process will be designed building upon the CCE loss function. Two different functions will be defined in combination with a parameter to regulate the strength and impact of each function on the resulting fitness score. The first function f calculates the number parameters of the phenotype of genome x. The second function g tallies the number of unintuitive design choices occurrences. For a design choice to be counted as an unintuitive one, the stride of a pooling layer must be larger or equal to the size of the pooling layer. The parameter count function f is used together with the parameter α. As for the unintuitive penalty function g, it is influenced by parameter β. As seen in Equation 4.1.

$$fitness(x) = CCE + \alpha \cdot f(x) + \beta \cdot g(x) \tag{4.1}$$

For finding the optimal parameters for α and β to in use in Equation 4.1, a grid search will be conducted. The values considered in this grid search can be found in Table 4.3.

Table 4.3: Parameters used in the experiment on the impact of using custom fitness functions tailored to limiting the complexity for the evaluation phase.

Parameter name	Values
α	0, 1e-7, 5e-7, 10e-7, 50e-7, 100e-7
β	0, 0.001, 0.005, 0.010, 0.050, 0.100, 1.000

The ranges for each of the variables has been carefully chosen to encompass a range of change within reasonable bounds. The values considered for the variable α are directly

related to the codomain of function f, which are the number of parameters of a neural network based upon the genomes as defined in Table 2.2. The size of genomes as measured in number of parameters is at minimum 527 parameters small and at maximum 902 763 parameters large. As a response the values of α for the experiment are between the range of 1e-7 and 100e-7. Similarly, the codomain for penalty function g is a vector of values $\{1, 2, 3, 4, 5\}$. From these values, the values of parameter β are selected to be within the range of 0.0001 and 1.0000.

Each combination of custom fitness function parameters is evaluated 35 times. Every configuration is run with the minimum experiment population size. This is the minimum value possible for reliable translatable results for experiments from research experiment IV. Each run has a runtime of 10 generations and uses the fixed key non-desynchronised ASCAD dataset. Every generation a snapshot of the genome population plus their fitness scores are made.

Then, results are analysed with the genome serving as the origin for the complexity metrics –both parameter count and unintuitive design choice count– and the fitness score giving insight into the impact on convergence.

Our expectations of this experiment are that higher values for both α and β will result in better genome fitness in the long run. We base this expectation on the belief that smaller neural network designs might be better at generalizing and might be less prone to overfitting for this dataset. Regardless of the eventual effect on performance, both custom fitness function penalty functions should reduce the complexity of the end results. However, fears do exist that high values for β (e.g. 1.0000) might exclude possible good performing genomes in the initial generations of the population. If in the initial generations well performing genomes with unintuitive design choices are significantly discounted, the selection process might focus on in comparison sub-par genomes. If instead the penalty strength β is of a lower value, these genomes might still carry on to newer generations and be refined into less complex genome designs.

4.5.3 Premature convergence

To answer the third research question we will first implement some of the anti-stagnation strategies commonly used in genetic algorithms, namely the following ones:

4.5.3.1 Distribution index η

The lowering the distribution index η results in larger perturbations. The current distribution index η has been tested as the most performant when it is equal to 20. However, as the original NASCTY paper only tested for the higher values 20 and 40 with smaller mutations, lower values for the distribution index η might help in escaping local optima and reducing the premature convergence. The implementation considered in the thesis will be based upon a distribution index η equal tot 10.

4.5.3.2 Adaptive mutational rate

The *adaptive mutational rate*, the mutational probability values of the genetic algorithm will be adjusted dynamically for each of the offspring. The specific strategy that will be implemented for this experiment is *rank-based mutation probability* based upon the

methodology described by Basak in [57]. This approach assigns the mutation probability of the hyperparameters of individuals based on the relative ranking in the population based upon the fitness scores of the genomes [58, 59]. The average mutation probability of a gene in a genome in this strategy matches the mutation probability of the original algorithm. To calculate the mutation probability p of a genome, the defined in Equation 4.2 rank based adaptive mutation probability function can be used. Before one can calculate the probability p, the size of the population N and the rank r the genome occupies in that population must be known. Then together with the maximum mutation probability p_{MAX} the individual mutation probability of a genome can be calculated.

$$p = p_{\text{MAX}} \cdot (1 = \frac{r-1}{N-1}) \tag{4.2}$$

4.5.3.3 Partial replacement

The last strategy considered is *Partial replacement*. Whenever the genetic algorithm threatens to stagnate for longer periods of time, a partial replacement of the current population will occur. In this partial replacement a portion of the total population will be culled and a new group of randomly generated genomes equal to the number of culled ones will migrate into the population, keeping the total number of individuals stable. The moment when a partial replacement of the population is applied can be dependent upon any of the genetic algorithm's metrics. In this experiment the partial replacement action will be tied to the stagnation for five generations of the best fitness score currently in the population. The method for the selection of genomes to be culled will be done by tournament selection similar as to the selection step of the NASCTY genetic algorithm as described in Section 2.6.2.

4.5.3.4 Anti-early stagnation strategies

Now that the anti-early stagnation strategies have been defined and implemented, the experiment can start. On top of the NASCTY algorithm, each of the anti-premature convergence strategies is run for a maximum of 75 generations. The maximum number of generations is greater as the goal is to combat pre-mature convergence. Because the original algorithm suffered from pre-mature convergence and had long plateaued at the 50th generation, the effectiveness of a modification can only be measured after that point. The experiments use the fixed key non-desynchronised ASCAD data set for its training and evaluation of genomes. The population size for this experiment is wholly depended on the results found in experiment IV. Each generation the population and runtime are collected as metrics. This process is repeated 30 times for each configuration.

The efficacy of these strategies then is determined by a combination of metrics. Mainly, the convergence and diversity metrics of a run are combined. The runtime cost is factored in too when determining the feasibility of an approach.

We expect that the most interesting results of this experiment will come from the second and third strategy. When the first option of lowering the distribution index η is used to increase the mutation range used within the algorithm, we expect that the heightened mutation range will also come with its own trade-offs and in turn might make certain movements within the genetic algorithm overshoot more easily. So, while the total run time of the genetic algorithm is not impacted negatively, there might be more

of a difficulty in finding an optima. Our expectation is that by increasing the mutation rate for lower performing offspring more area and possibly more diverse options might be discovered, increasing the number of local optima discovered in its runs, giving a better overall solution at the end of the runs. The final strategy of replacing part of the population when the genetic algorithm stagnates for multiple generations seems to us as a great way to reintroduce fresh blood into the system. Therefore, our main estimation and hypothesis will be that the second and third strategy will bring in a steady defence against premature convergence and early stagnation. With the first strategy potentially being an improvement for the genetic algorithm, it might also impede the finding of better solutions with its stronger mutation range.

4.5.4 Combined result

Having implemented each of the experiments and gathered their results, the next step in this research would be to combine the best elements of the different experimental alterations to the genetic algorithm. The results found when answering research questions RQ1, RQ2, and RQ3 have been done by individually making changes to the NASCTY genetic algorithm. The modifications that promise improved performance and reduced complexity of their end results, will then all individually be applied with the goal of overcoming the shortcomings of the original algorithm. From RQ1 the following modifications to the NASCTY algorithm are taken:

- Experiment I, the minimum required training epochs for accurately evaluating potential solutions;
- Experiment II, the early termination policy and either the conservative or optimised approach's fitness threshold values, if and only if found to be an adequate trade-off between training time reduction and missed potential novel solutions;
- Experiment III, the training data partitions size that gives a suitable trade-off between a reduction of training time needed for evaluating a genome and the accuracy penalty a smaller partition size may carry;

The population size is not something that will carry over from experiment IV, the explicit choice has been made to stick to the same population size of 100 used in the NASCTY [16] paper. This has been done with the goal of being closer and therefore more reliable in the comparison between the stock genetic algorithm and the modifications proposed in this thesis.

From RQ2 the custom fitness function together with the best performing values for α and β parameters will be carried on and implemented. The best values for α and β will represent the combination that reduces the complexity in parameters and unintuitive design choices while simultaneously not negatively impacting the fitness score. Lastly, the modifications taken from RQ3 will be the best performing anti-premature convergence strategy when viewed from the perspective of convergence.

With this new algorithm whose summation is all the best alterations, the new algorithm shall be evaluated against a common side-channel analysis benchmark.

For each benchmark dataset a number of runs have been done for both the stock NASCTY configuration and the modified algorithm. The number of runs done for a benchmark is wholly dependent on the complexity and as a consequence the time needed

to benchmark. The benchmark was run ten times for the fixed key non-desynchronised ASCAD dataset, five times for the fixed key 50 and 100 desynchronised ASCAD dataset, and 3 times for the random key dataset Of each run the population every generation plus the overall runtime was collected.

To analyse the efficacy of the modification regarding the two main limitations of the original NASCTY algorithm, the data was consulted in both the convergence and complexity metrics.

To get an accurate overview of the performance of our best performing genome, the genome is trained for a total of 75 epochs. From the fully trained model performance characteristics are then calculated. These are then compared to the results found by the other papers with respect to the ASCAD dataset as laid out in Table 2.1.

Our hypothesis on the effectiveness of our new altered genetic algorithm in finding the most performant neural network design for the synchronized traces will be that it is going to be an improvement upon the original NASCTY algorithm by a non-significant but measurable level. Alongside this, we deal with two possible scenarios fully dependent on the effectiveness of the premature convergence combat strategies. If the strategies outlined in Section 4.5.3 pertain to be effective strategies, then so too will the level of improvement of the genetic algorithm. We do expect the impact of all the changes to be greater for the higher level of desynchronisation, since the original NASCTY seems to suffer more greatly of unneeded complexity when dealing with the more difficult desynchronised traces.

4.6 Data analysis

This section explores the metrics used in this thesis and the used data analysis methods. It follows the chronological order that the metrics and methods show up.

4.6.1 Selection accuracy

The selection accuracy represents how much one population's fitness scores $\mathbf{x}$ would reflect the selection choices based upon the baseline's fitness scores $\mathbf{y}$.

To find the selection accuracy for the results vector $\mathbf{x}$ compared to the baseline results vector $\mathbf{y}$, the number of correct findings is divided by the number of combinations possible for a vector. For this the number of k combinations of a random vector x consisting of n elements can be denoted as a *binomial coefficient* ($\binom{n}{k}$) as seen in Equation 4.3.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{4.3}$$

Next, the number of correct values must be found. Let us denote all possible combinations $C_k(\mathbf{x})$ of k elements per combination of vector x as:

$$C_k(\mathbf{x}) = \{(x_{i_1}, x_{i_2}, \ldots, x_{i_k}) \mid 1 \leq i_1 < i_2 < \ldots < i_k \leq n\} \tag{4.4}$$

$$C_k(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{y} \subseteq \mathbf{x},\ |\mathbf{y}| = k\}$$

Then, lets denote m as shorthand for the index of the lowest value in a possible combination vector.

$$m = \text{Index of } min((x_{i_1}, x_{i_2}, \ldots, x_{i_k})) \text{ in } \mathbf{x} \tag{4.5}$$

With these we can denote the equation for calculating the accuracy of tournament selection at a certain point compared to the baseline in Equation 4.6.

$$\text{Accuracy} = \frac{|\{(x_{i_1}, x_{i_2}, \ldots, x_{i_k}) \in C_k(\mathbf{x}) \mid m = \mathbf{y}_{i_m}\}|}{\binom{n}{k}} \tag{4.6}$$

4.6.2 Convergence

In short, convergence is the metric used for an algorithm to describe its performance as it converges on (local) optima. It is the point where the algorithm stabilises on (seemingly) optimal neural network architectures.

This progress is tracked in this research by recording the population's best performing genome's fitness score. Against what one might expect, the best performing genome is the one with the lowest fitness score.

When analysing the convergence, the fitness score together with the generation or epoch is plotted. Giving a brief overview of the progress or lack of when seen over a longer time frame.

4.6.3 Diversity

Diversity as a metric represents the sameness of all solutions. In this thesis the choice was made to represent the diversity of a population as the mean distance between each genome. The more similar the genomes the lower the distance between the genomes, the more dissimilar the larger the distance.

To calculate the genomes distance to each other, the genomes are then transformed into a n-dimensional Euclidean coordinate where each dimension represents one of the genomes' genes as a value between 0 and 1. With these coordinates it is then simply a matter of applying the Euclidean distance formula for multidimensional coordinates. Equation 4.7 stipulates the formula for finding the distance function d between two coordinates $\mathbf{p}$ and $\mathbf{q}$ in a multidimensional coordinate system of n dimensions.

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(\mathbf{p}_1 - \mathbf{q}_1)^2 + (\mathbf{p}_2 - \mathbf{q}_2)^2 + \ldots + (\mathbf{p}_n - \mathbf{q}_n)^2} \tag{4.7}$$

4.6.4 Complexity

The second of the most important metrics for this research next to the convergence are the complexity metrics. These metrics come in two forms, the parameter count of a neural network model and the number of unintuitive design choices in the genome structure.

The goal of the neural architecture search is to find the most optimal solution to the question of neural network architecture design. This is mainly measured in two metrics one for attack performance and one for complexity. Appendix A shows the comparison of different approaches by measuring the traces to obtain mean key rank 0 (T_{GE0}) and the number of model parameters for estimating the complexity of a solution.

This research uses next to the parameter count of the neural network solution also the number of unintuitive design choices. These design choices add parameters to neural

network model that seemingly serve no purpose. By limiting these design mistakes the idea is to reduce complexity without harming the performance.

To demonstrate the effects of certain experiments, the spread of results is shown with the help of box plots.

4.6.5 Statistical tests

When presented with data distributions one might not be able to reliably determine the impact of experiments. If this is the case, statistical tests help to clarify results.

4.6.5.1 Kruskal-Wallis

When faced with uncertainty if the experiments have made any impact on a metric, the Kruskal-Wallis [60] test should help clarify this. It determines if the different sample results share the same distribution.

The Kruskal-Wallis rank sum tests the null hypothesis H_0 that states that all the groups in group $\mathcal{G}$ come from the same distribution. For a thesis to be rejected the probability that the difference between the group is due to chance, this is the p-value. The p-value has to be less then 0.05 in order to reject a thesis. This low of a p-value suggest that at least one group is significantly different.

4.6.5.2 Mann-Withney

When faced with the question if two samples share the same distribution. A question relevant when testing if a modification to the genetic algorithm has made any impact on metric compared to the default. The Mann-Withney U [61] test is used.

This test is similar to the earlier mentioned Kruskal-Wallis test, which is an extension of this test. It poses the null hypothesis H_0 that two samples share the same distribution and have no significant difference in medians. To prove the null hypothesis H_0, the chance (p-value) of it being a coincidence must be less than 0.05. If the p-value is larger than this threshold, one may assume that they do not share the distribution.

4.6.6 Performance

To measure the performance of the genetic algorithm's results, the traces needed to obtain key rank 0 are calculated.

Calculating the traces to obtain mean key rank 0 (T_{GE0}) is normally done over 100 folds. In general lines, the attack dataset is first split up in 100 equally sized subsets, called folds. Then each fold calculates the number of traces needed before the key rank is for the first time equal to 0. Then the average of the minimum number of required traces is the T_{GE0}.

4.7 Ethical considerations

This research samples it data from an open-source dataset free from personal identifiable information. The absence of data that could be used to identify individuals, ensures compliance with ethical privacy standards.

The dataset itself originates as condensed measurements from an existing open platform implementation. The aim of this work is to improve upon automated side-channel analysis methods. Research like this lowers the barrier to entry for validating and improving the robustness of implementations against side-channel analysis. It thereby contributes to the development of stronger and more resilient systems.

4.8 Limitations

This section describes the limitations we ran into when conducting this research and what changes were made to address them.

The largest limitation of the conducted experiments has to do with usage of only the fixed key non-desynchronised dataset. This limitation was set in place as there was simply not enough time to conduct the experiments with the more complex and time-consuming dataset. By limiting ourselves to only one set the results and approaches mostly refer to that set. There is no guarantee that the results are generalisable to other sets, and may not perform in the same manner or with the same strength on other more difficult datasets.

This is in part addressed in Chapter 8 were the modifications are benchmarked on other datasets. However, even those suffer from an unequal treatment in regards to results. The more easier datasets allow for quicker gathering of results, and the more complex datasets require too much computing time.

Additionally, some smaller changes were made to experiment I and II after witnessing the low success rate of random genomes. The sampling pool was enlarged for both experiments to counter this limitation.

Table 4.4: Parameters used in the experiment on the impact of using custom fitness functions tailored to limiting the complexity for the evaluation phase.

Topic	Experiment	Methodology
RQ1 Hyperparameters	I	Training 500 genomes for 50 epochs to discover the minimum required of training epochs for accurate fitness predicting.
	II	Training 2000 genomes for the minimum required training epochs with the goal of implementing an early termination policy.
	III	Training the Wouters *et al.* [42] and Schijlen *et al.* [16] alongside 15 random genome designs with different training data partition sizes for the goal of reducing training time.
	IV	Running the NASCTY algorithm 20 times for 50 generations with different population sizes with the goal of reducing the population size used for experiments and thereby speeding up the experiments.
RQ2 Complexity	I	Run the genetic algorithm 35 times for 10 generations for each of the α and β value combinations of the custom fitness function with the aim of reducing the complexity of the end results.
	II	Run the most promising α and β value combinations for 20 generations as to allow the results to diverge and see the impact on the longer time span.
RQ3 Premature convergence	I	Running the genetic algorithm 30 times for 75 generations and a population size of 50 for three separate anti-premature stagnation strategies.
Combined results	I	Run the modified genetic algorithm 30 times for 50 generations with a population of 100 and compare the effectiveness to the original NASCTY algorithm on the level 0, 50, and 100 (non-)desynchronised fixed key ASCAD dataset.

Chapter 5

Hyperparameters

This chapter together with Chapter 6 and Chapter 7 describe and relate the findings of the experiments and methodology described in Chapter 4 whose goals are to answer the research questions posed in Chapter 3. The results below follow the same order as the experiments described in Chapter 4. The results described in this chapter are used afterward in the discussion of the results in Chapter 9 and in turn inform the conclusions drawn in this research, as seen in Chapter 10.

These findings below are the results of experiments with the goal of answering the research question RQ1. Each of the below experiments tries to illuminate a part of the problem and in turn give us the needed information to answer our question. The experiments and their results have been split up in three separate parts. The first results are directly related to the relation between epochs trained and the accuracy of predicting the future fitness of the genome. The second experiment results try to convey the possible saved training time by implementing a fitness threshold policy. The third experiment then tries to alternate on the amount of received data points in the training and evaluation period of the genome. Lastly, the fourth experiment hopes to highlight the effect of the population size on the convergence, diversity, and coverage of the genetic algorithm.

5.1 Accuracy fitness

The first experiment will be testing for the number of epochs a phenotype of a genome that has to be trained before an accurate indication of the 50 epochs trained genome's fitness becomes clear.

For this experiment a set of 500 randomly generated genomes were trained for 50 epochs. During the training period of each genome's phenotype, every epoch the neural network model was evaluated on the validation set, saving the fitness score and total time needed for the training the model. These genomes and their progression can be seen in Figure 5.1.

Since the goal of the genetic algorithm is to select for the best performing genomes –meaning the genomes whose fitness scores are of lower value– two separate comparisons were made. One comparison includes all genomes tested and trained, and the other two comparisons consider only genomes whose phenotypes' fitness score improves compared to the starting fitness score with two different thresholds: ≥ 0.00 and ≥ 0.01.

Two interesting findings can be discerned from this figure. The first thing that jumps out is the high number of non-improving examples. These seem to initially either stay at

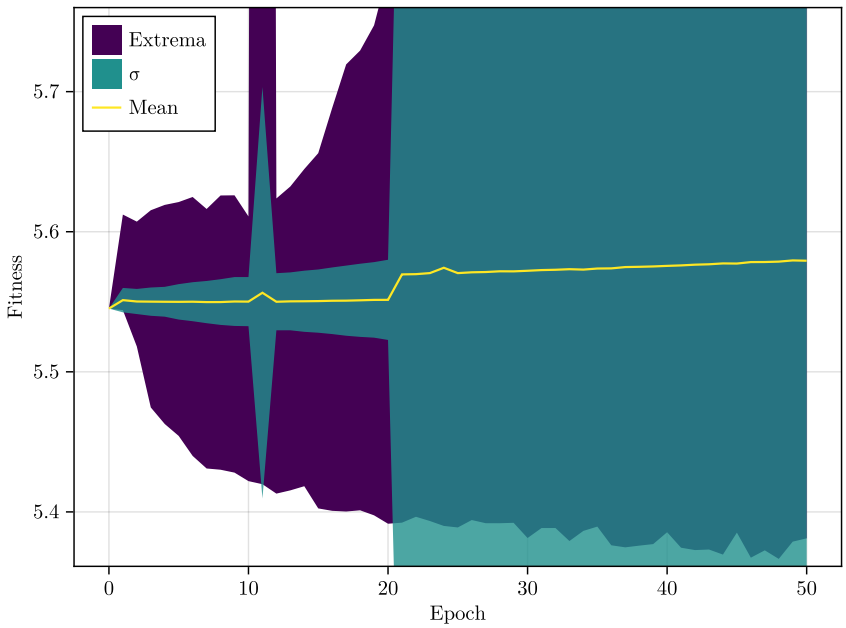
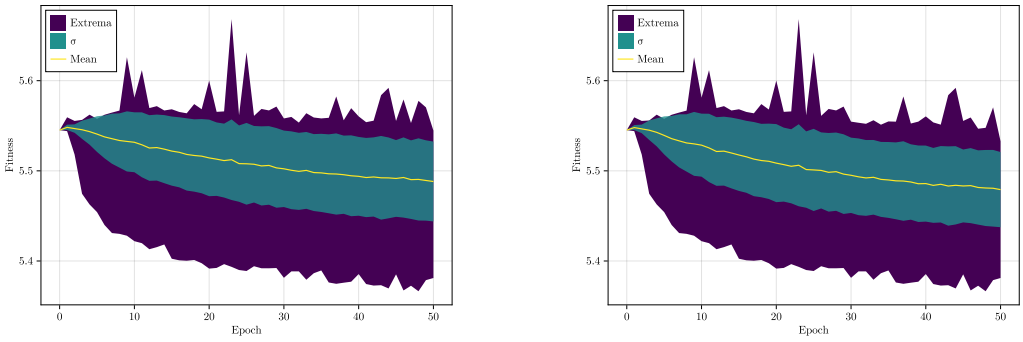


Figure 5.1: The validation fitness of 500 random genomes over 50 epochs.

the same fitness score or worsening the more epochs they are trained. Of the 500 randomly generated only 43 genomes managed to improve their fitness score at all, with 43 genomes only improving 0.01 or more over the span of 50 epochs.



(a) Improvement ≥ 0.00 (76)

(b) Improvement ≥ 0.01 (65)

Figure 5.2: The validation fitness of random genomes over 50 epochs that improve with a Δ of 0.00 for (a) and a Δ 0.01 for (b).

Filtering those groups out leaves us with the results seen in Figure 5.2, a stark contrast to the 500 genomes originally recorded. These three groups have been selected to emphasize the effectiveness of the changes at those three levels in regard to improving the

genetic algorithm's selection stage. The selection stage and as a consequence the genetic algorithm's goal is to select the best genomes to carry on to the next generation. The best genomes are those most suited to the task of predicting the values of belonging to the correct dataset. In general, as can be seen in Figure 5.1, all of what can be considered good genomes improve in their fitness score over time. Conversely, the same observation can be made about the worst performing genomes.

However, since our goal is with finding the most suitable possible solution, two additional categories have been initiated. is calculated The groups consist of potential solutions that have either seen improvements in their training time (i.e. any genome with a $\Delta \geq 0.0$ in their fitness score), and those that have improved at least with a Δ of 0.01 in the same time span, neatly separating the weed from the chaff by doing so. This gives us a good indication of the accuracy of the selection process when only training a genome for a selected amount of epochs during their training.

To calculate the accuracy of selection method at a certain point in time, all possible combinations of genomes used in the selection process are compared to a baseline set at epoch 50. The selection accuracy of the tournament selection with three contestants as used in the original NASCTY algorithm are shown in Figure 5.3. Additionally, the selection accuracy was also calculated for tournament selections with only two contestants, analogue to more conventionally used metrics for defining accuracy. These accuracy results can be found in the appendix under item Appendix B.

5.2 Early termination

The second experiment is concerned with the usage of a fitness threshold in the evaluation stage of a possible solution. This is done with expectation that such a fitness threshold will reduce the number of training epochs needed to evaluate all the genomes. Thereby, meaningfully reducing the needed runtime of the genetic algorithm.

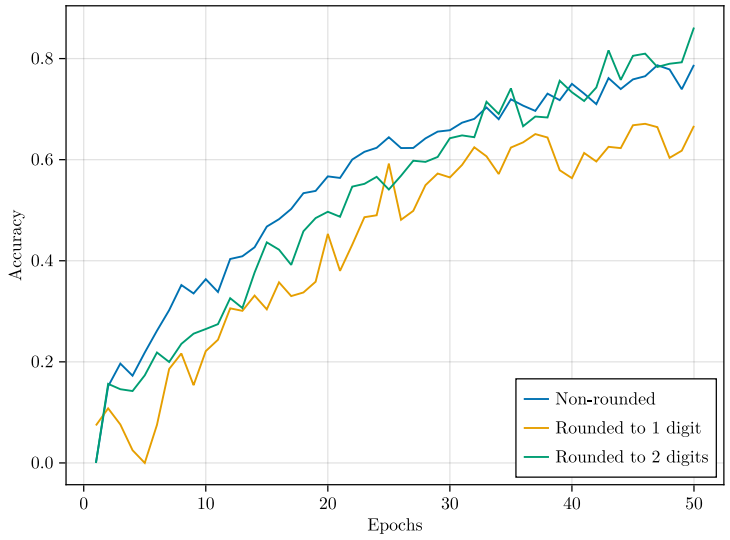
Table 5.1: The threshold value categories used on the expected improvement of possible solutions from the first epoch to the last epoch measured.

Parameter name	Values
Threshold categories	≥ 0.00, ≥ 0.01, ≥ 0.05

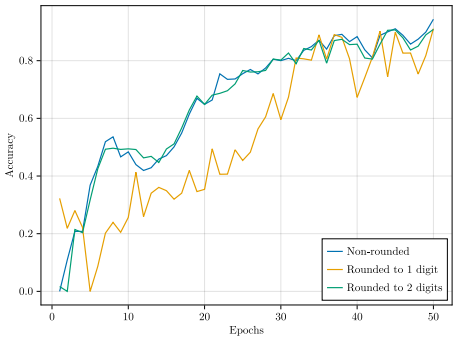
The two fitness thresholding approaches –conservative and optimised– will be calculated for all the different threshold categories as defined in Table 5.1. These categories have been defined as to highlight certain characteristics of the data. The goal of the ≥ 0.00 category is to highlight all the genomes who improve over time. The ≥ 0.01 categories' goal is to see the impact on all genomes that improve at least a little bit and aren't a random deviation from the baseline. Lastly, the ≥ 0.05 category was created to focus on the genomes that improve significantly over their starting point.

With these trained data points collected and the fitness thresholding approaches defined, the following metrics and values will be calculated to display the impact of these approaches on:

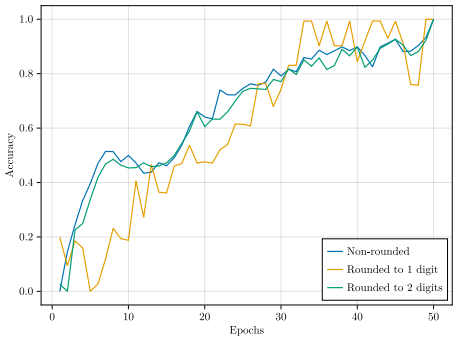
1. the number of possible solutions meeting the thresholds;
2. the number of promising solutions not meeting the thresholds;



(a) All genomes.



(b) Improvement ≥ 0.00 (53).



(c) Improvement ≥ 0.01 (43).

Figure 5.3: The accuracy of the selection function with different training periods for the phenotypes –measured in epochs– in the evaluation stage of the genomes.

3. the time saved training in epochs with these fitness threshold strategies compared to not implementing an early termination policy.

Table 5.2: Thresholds values for fitness scores each epoch for the conservative approach.

	1	2	3	4	5	6	7	8	9	10
≥ 0.00	5.601	5.589	5.593	5.568	5.598	5.556	5.557	5.554	5.552	5.545
≥ 0.01	5.601	5.589	5.593	5.568	5.598	5.554	5.553	5.554	5.548	5.535
≥ 0.05	5.602	5.589	5.593	5.568	5.598	5.553	5.552	5.545	5.536	5.425

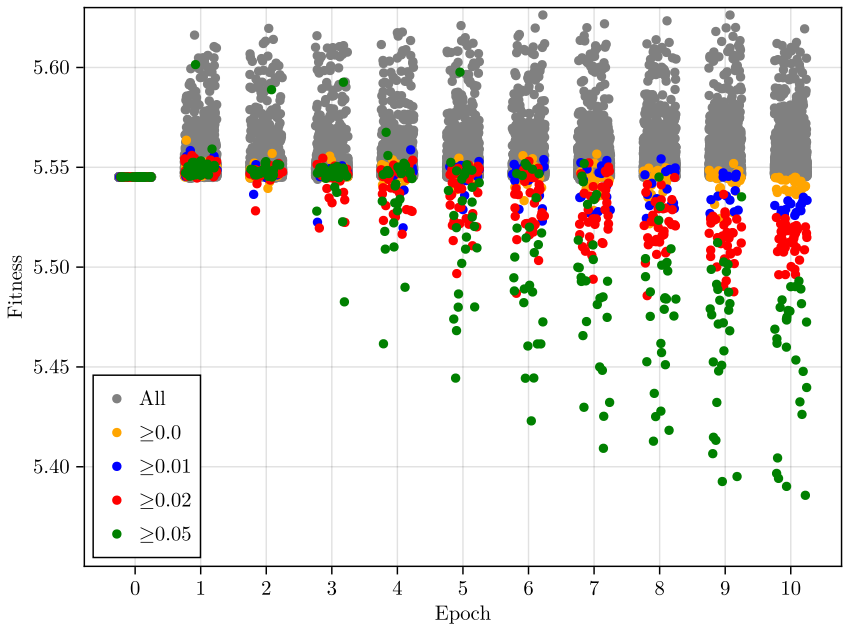


Figure 5.4: The validation fitness of 2009 random genomes over 10 epochs.

Table 5.3: Thresholds values for fitness scores each epoch for the optimised approach.

	1	2	3	4	5	6	7	8	9	10
≥ 0.00	5.601	5.589	5.593	5.568	5.598	5.556	5.557	5.554	5.552	5.545
≥ 0.01	5.601	5.589	5.593	5.568	5.598	5.554	5.553	5.554	5.548	5.535
≥ 0.05	5.555	5.552	5.593	5.568	5.598	5.553	5.552	5.545	5.536	5.425

Such a fitness threshold will terminate the training of a possible at an intermediate training epoch if the improvements of the fitness of a solution compared to the starting fitness does not meet the requirements set by the threshold.

Table 5.4: The number of solutions considered for further training per epoch depending on the requirements set on the improvement in the fitness evaluation from the starting point when making use of the conservative fitness threshold policy.

	1	2	3	4	5	6	7	8	9	10	Total
Baseline	2009	2009	2009	2009	2009	2009	2009	2009	2009	2009	22099
≥ 0.0	1999	1977	1974	1925	1925	1735	1684	1552	1344	99	18223
≥ 0.01	1999	1977	1974	1925	1925	1657	1537	1473	607	77	17160
≥ 0.05	1999	1977	1974	1925	1925	1518	1375	85	68	27	15075

For this we have selected certain threshold levels of required improvement of the fitness

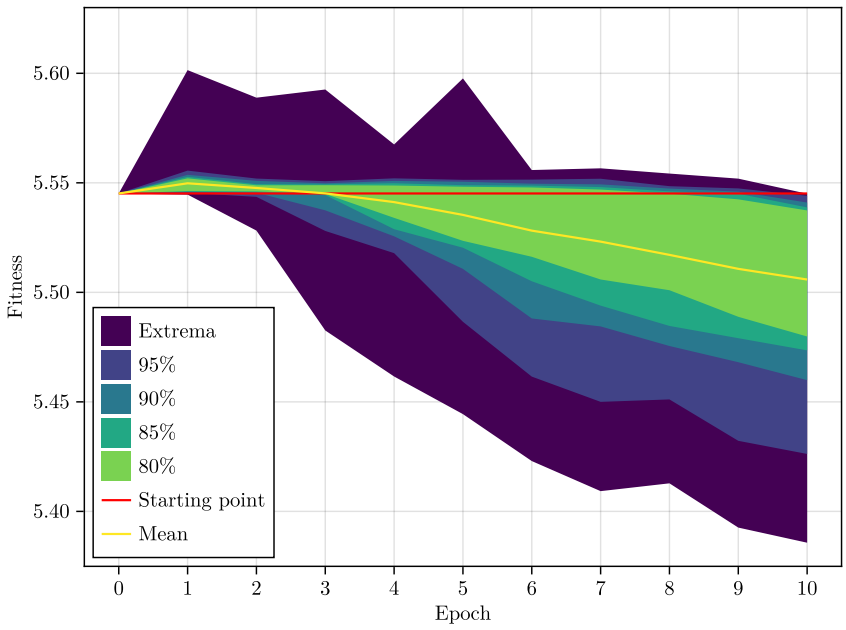
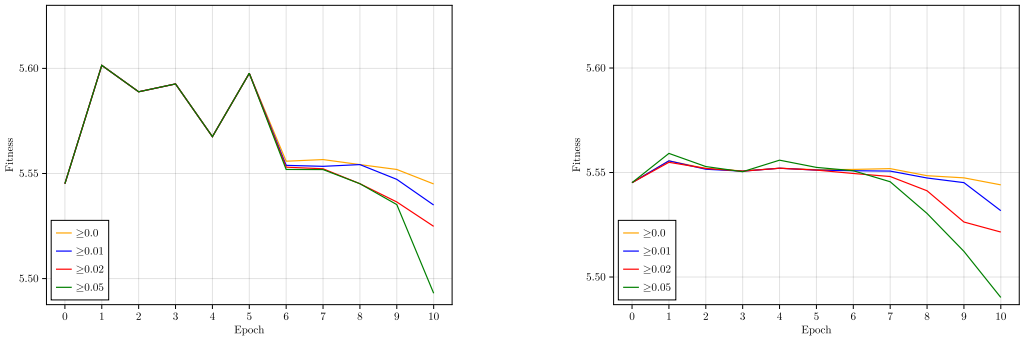


Figure 5.5: The validation fitness of 2009 random genomes over 10 epochs, showing the extremes and the population dispersion.



(a) Conservative (216).

(b) Optimized (65).

Figure 5.6: The fitness thresholds projected on graph for both the conservative (A) and optimised (B) threshold values.

of selected possible solutions. These thresholds are selected upon the findings presented in Figure 5.1 and Figure 5.2 that mark, that any interesting solutions for the generic algorithm –those solutions whose fitness score are lower– exhibit two characteristics. First over time with training the quality of the fitness score improves, and secondly that any of the more interesting solutions, that are those who see at least an improvement greater than 0.01, start improving their fitness score before the tenth training epoch.

Table 5.5: The number of solutions considered for further training per epoch depending on the requirements set on the improvement in the fitness evaluation from the starting point when making use of the optimised fitness threshold policy.

	1	2	3	4	5	6	7	8	9	10	Total
Baseline	2009	2009	2009	2009	2009	2009	2009	2009	2009	2009	22099
≥ 0.0	1749	1613	1612	1610	1610	1513	1476	1387	1229	91	15899
≥ 0.01	1749	1593	1592	1590	1590	1446	1363	1315	569	70	14886
≥ 0.05	1839	1708	1707	1705	1705	1434	1308	84	67	26	13023

The goal of setting a fitness threshold in one of the epochs before the final evaluation is to reduce the time spend on training solutions one might be considered as undesirable while not excluding novel solutions that would fit the considered interesting. Table 5.4 and Table 5.5 show the number of solutions meeting the fitness threshold improvement requirements for each of its training epochs respectively for the conservative and optimised approach. According to the numbers as found in Table 5.6 the gains one gets in time saved training –an approximate ∼10% additional saving–, comes with an approximate similar loss of around ∼10% of potential interesting candidates in most situations.

Table 5.6: Percentage of saved training epochs when applying an early stoppage policy for training when evaluating possible solutions when applying the conservative approach (I) and the optimised approach (II).

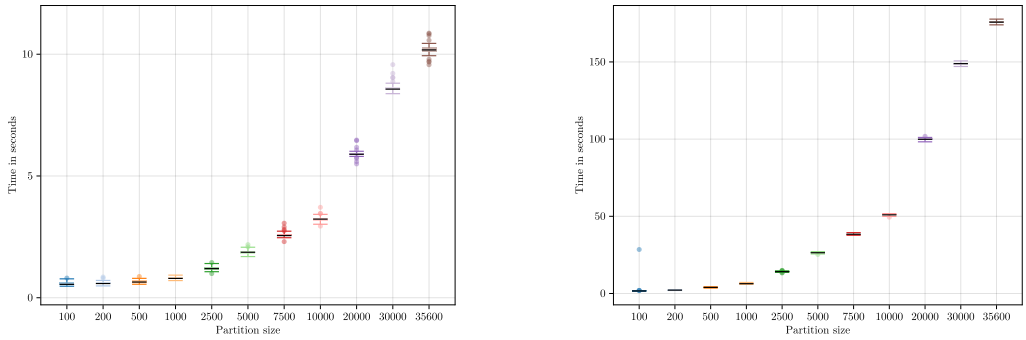
Threshold level	False negatives	Total epochs	Percentage
Baseline	0	22099	1.000
≤ 0.00 (I)	0	18223	0.825
≤ 0.01 (I)	0	17160	0.777
≤ 0.05 (I)	0	15075	0.683
≤ 0.00 (II)	8	15899	0.719
≤ 0.01 (II)	7	14886	0.674
≤ 0.05 (II)	1	13023	0.590

For ease of readability and ease of reasoning about the results, the time saved by implementing early termination with the help of fitness thresholds will be expressed in training epochs in Table 5.6 and from here on out. For the time needed by a machine to train a machine learning for a single epoch differs highly on the configuration and system used, while the number of epochs required to evaluate a possible solution should be constant.

5.3 Training dataset size

This experiment is split up in two segments, where in the first segment it is limited to two proven designs of neural network architectures. These limitations are set in place to have selections on which the impact on fitness score can be more easily observed. The second segment in contrast considers a number of randomly generated genomes whose performance is not known in advance.

In this first segment of this experiment two types of machine learning models will be evaluated. A CNN and an MLP style genome, both genomes are based upon proven



(a) Wouters *et al.*[42].

(b) Schijlen *et al.*[16].

Figure 5.7: The time needed per epoch to train a neural network model depending on the size of the partition of the training dataset used each epoch.

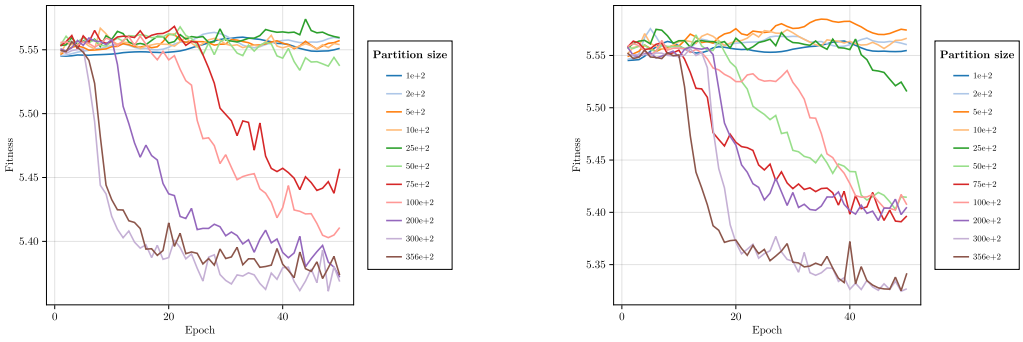
designs with good results. This is done in order to more clearly discern the impact of limiting the number of data points used in each training epoch on both the performance and training time in seconds. However, if instead for this part of the experiment random genomes had been used to study the impact on performance there would be no guarantee that these random genomes would evolve in a meaningful way over time during training. For the MLP style genome the design will be that of the MLP as proposed by Wouters *et al.* in [42]. The CNN style genome in turn is based upon the findings found in the original NASCTY paper [16].

For the third experiment first segment two neural network architectures were trained for 50 epochs and the fitness measured after each training session in their respective epochs.

This was repeated with different sized training data partitions as defined in Table 4.1, limiting the number of unique random data points used from the original complete training dataset with the goal of improving the time needed for evaluating a genome's fitness score.

The needed time per epoch for a model to train the model as seen in Figure 5.7 gives us two distinct results. Both results have been found on a machine by training the models on the GPU, as this is the primary way we expect that others would evaluate neural network architectures and run the NASCTY algorithm. The results for Wouters *et al.* [42] shows that as an MLP style neural network architecture, the needed time per epoch for training is shown to scale linearly with the size of training dataset partition. The neural network architecture proposed by Schijlen *et al.* [16] shows a similar behaviour for the time needed to train the network for a single epoch. Both therefore seem to scale and behave in an identical manner if not only for the difference in the amount changed per step up in size.

The second category of results collected in this experiment are the fitness score per epoch for the neural network models. This information is collected with the intention of highlighting the impact of reducing the partition size of training data on the fitness score achieved in the evaluation of the neural network models. They show the impact of changing the training set partition size used in evaluating the neural network architecture. For both Wouters *et al.* [42] and Schijlen *et al.* [16] seem to be sensitive to any of the changes made to the size of training dataset partition sizes, and also reflect these immediately within the fitness score. This causes linearly longer training times for similar



(a) Wouters *et al.*[42].

(b) Schijlen *et al.*[16].

Figure 5.8: Evaluation fitness per epoch with different partition sizes for the training dataset size.

results. While also having trouble reaching the same values as the complete dataset when not making use of the complete dataset each training epoch.

With these two well performing neural network examples, a twenty additional random generated genomes have been trained for 50 epochs with differing sized partitions. The impact of the size the partition used for training each possible solution on the time needed for evaluating such a possible solution, can be seen in Figure 5.9.

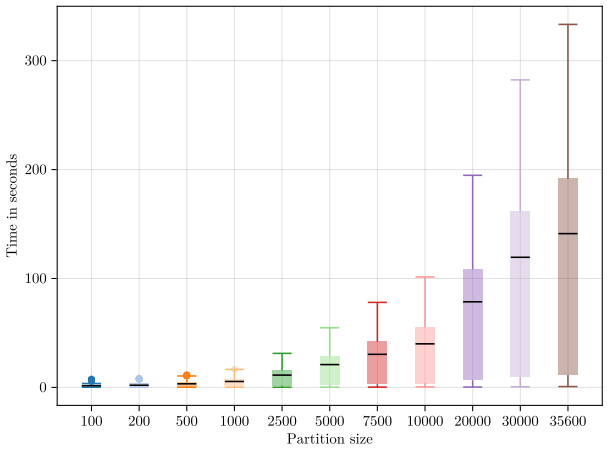
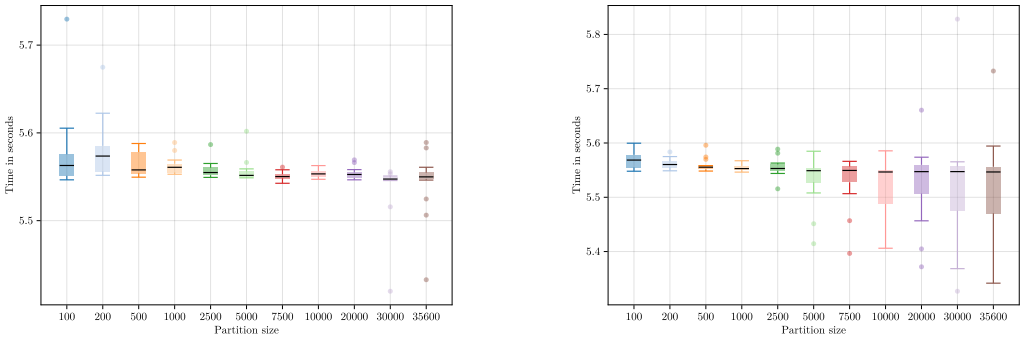


Figure 5.9: The time needed per epoch to train a neural network model depending on the size of the partition of the training dataset used each epoch for twenty random neural network designs.

These show a similar image to the results begotten earlier for the two proven designs with an increase in the time needed for training scaling equally to the number of data points in the training partition.

However, any decrease in time spending training with the reduced training dataset



(a) 10 epochs.

(b) 50 epochs.

Figure 5.10: The evaluation fitness dispersion of twenty random neural network designs at the end of 10 (A) and 50 (B) epochs when trained with different partition sizes.

partition comes at the cost of the potential and performance of the different genetic algorithm. Figure 5.10 shows us the dispersion of the different neural networks when trained with different partition sizes, and show that the more one limits the number of data points available in the training of the neural network the more limited its resulting performance will be.

5.4 Population size

For the third experiment, we collected metrics of ten different population size constraints for the NASCTY genetic algorithm. These ten different population size follow the values defined in Table 4.2, we ran the NASCTY algorithm ten times per population size parameter option. The metrics recorded allow us to contrast and inspect the results in the areas of convergence, diversity, and coverage.

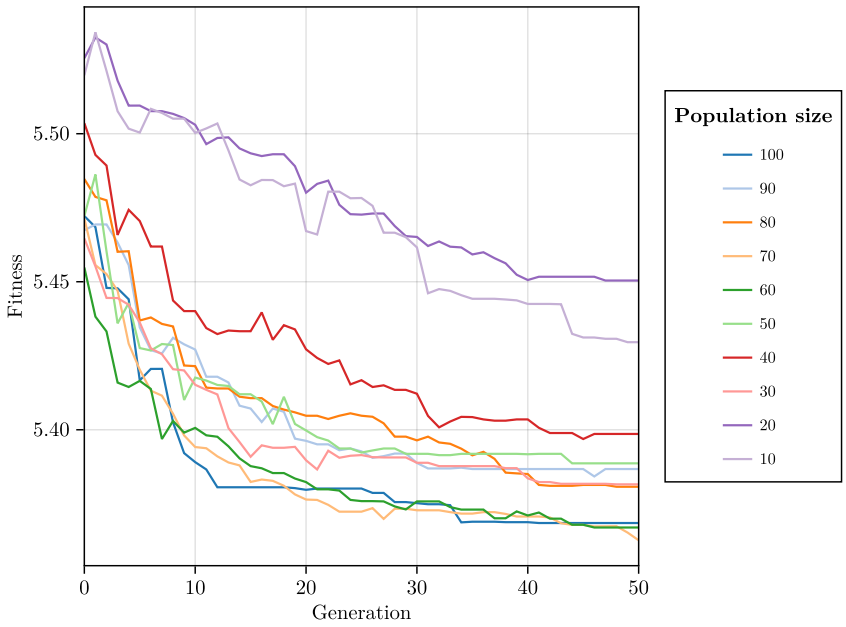


Figure 5.11: The mean validation fitness per generation of the best genome in the current population.

The convergence in these measurements are measured as the improvement in fitness score over each of the generations of the genetic algorithm. While the mean best solution in the population each generation as seen in Figure 5.11 gives a fine rough overview of the improvement in convergence. This is most visible in the speed of how larger populations descent and stagnate at an earlier generation compared to the other smaller populations. Two interesting outliers are the two smallest population options, with population sizes of 10 and 20 respectively, these seem to struggle with converging upon a single solution a lot more than those with greater population numbers.

A box plot of the dispersion found of the different fitness values at the end of the 50th epoch is presented in Figure 5.12 shows the different dispersion of twenty runs with different population sizes and their general effect and spread on the resulting quality of the final solutions produced.

An important metric for making sure that the complete population is converging upon a single optima is the usage of diversity metric. From such a metric one can infer multiple statements regarding the genetic algorithm. For a normal genetic algorithm the general form of these metrics are an indication of how the initial completely random population

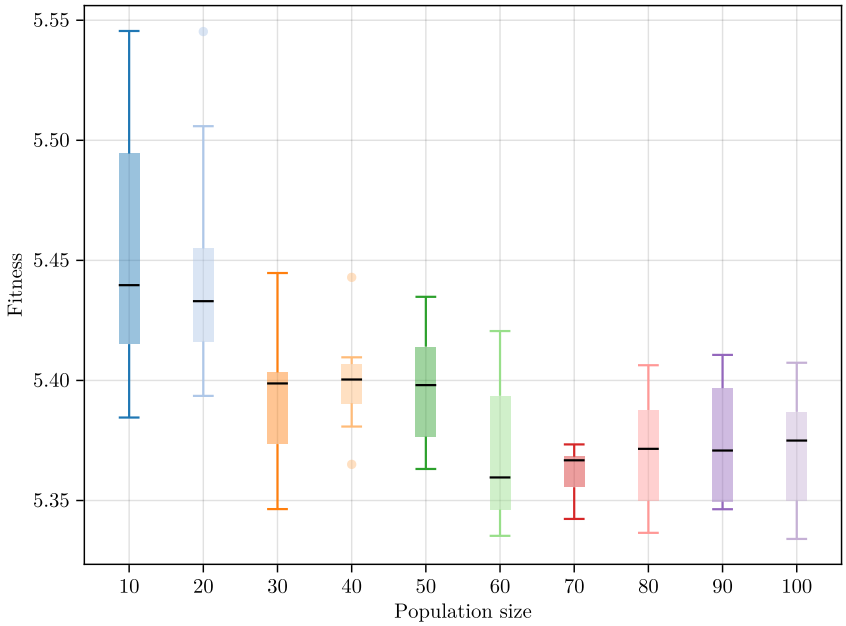


Figure 5.12: The validation fitness of the best genome in the current population at the 50th generation.

moves closer to each other over time. It is also an indication of the ability of the genetic algorithm to diversity its population and maintain enough of a diversity within a population as to be more able to escape local optimum. The findings seen in Figure 5.13 show the diversity of the genetic algorithms over time measured in generations. The diversity measured is the mean distance between each of the genomes in a population.

One interesting fact that stands out is the steep reduction of the diversity among all population sizes in the first twenty generations. This seems to somewhat stabilise after that point, this coincides with the witnessed stagnation of the convergence around the same time.

The coverage metrics of these runs collect and enumerate the number of unique genomes visited in the population. This too is done by converting all the genomes into points on an Euclidean space. With the genomes converted into coordinates, the next step then is removing all duplicate coordinates and tallying the amount. These measurements have been moved to Appendix D as they are more of a safeguard for checking that the genetic algorithm is actually exploring the search space in a non-predictable manner and not repeatedly visiting the same genome designs.

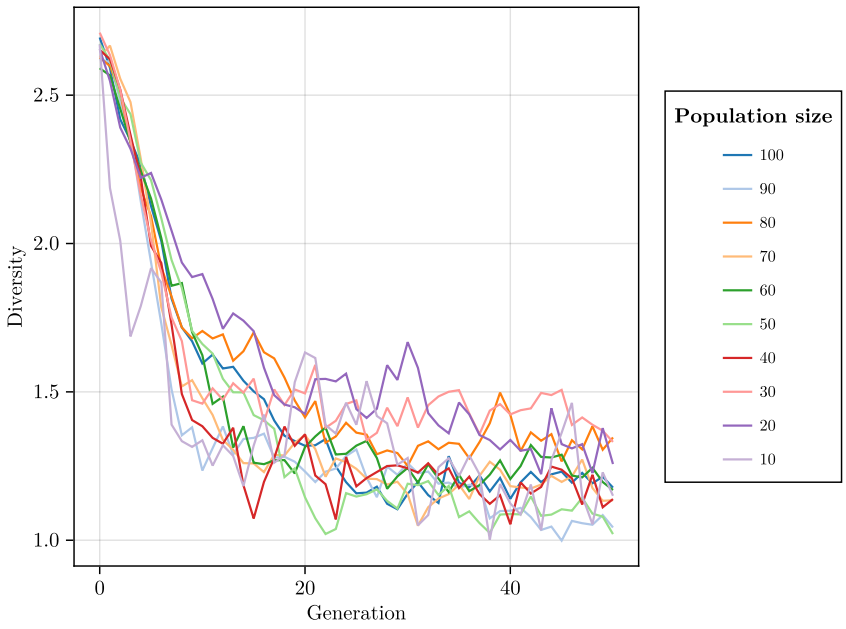


Figure 5.13: The population diversity measured as the mean distance between genomes per generation.

Chapter 6

Complexity

These findings below are the results of experiments with the goal of answering the research question RQ2.

Each of the below experiments explore one of the possible given directions for a custom fitness function whose job it is to limit unneeded complexity in the results of the genetic algorithm. For the two definitions given in Section 4.5.2 of the unneeded complexity of the genetic algorithm three different custom fitness functions have been designed. All three function are influenced by the parameters α and β denoting the strength of the modifiers used in the function. Custom fitness function I makes solely use of the α parameter and custom fitness function II is only concerned with parameter β, with custom fitness function III combining both indicators for unneeded complexity in the same function, and in turn makes use of both the α and β parameters.

For this the experiments then too follows the same order of custom fitness function and will discover the impacts and effectiveness of both custom fitness function I and II for both improving on the fitness scores and more importantly on the complexity metrics of the population.

Therefore, this experiment is split up into two steps. In the first step, each of the two custom penalty functions parameters will be evaluated as defined in Table 4.3 while the other parameter is set to 0. This is to test and see the effect of only one of the penalty functions on the custom fitness function. For this 35 runs of the genetic algorithm will be held for 10 generations.

After comparing both ranges of parameters and custom fitness functions on their own to the baseline where both α and β are equal to 0, the second step of the experiment will commence. In the second step of this experiment, all possible combinations of parameters for both α and β will be evaluated by running 30 times the genetic algorithm with a 10 generations time limit.

Having found our best performing custom fitness function parameters for both the α and β values, the most promising of parameter combinations will be extended to a 20 generation run of the algorithm. This extension from 10 to 20 generations is to highlight the growing divergence between the results.

6.1 Parameter count

Figure 6.1 displays the final fitness performance of the best performing solutions in the then current genetic population. These numbers have been acquired after running the genetic algorithm for 10 generations with a custom fitness function as defined in Equation 4.1 and substituting all values for α as described with those in Table 4.3 with β being equal to

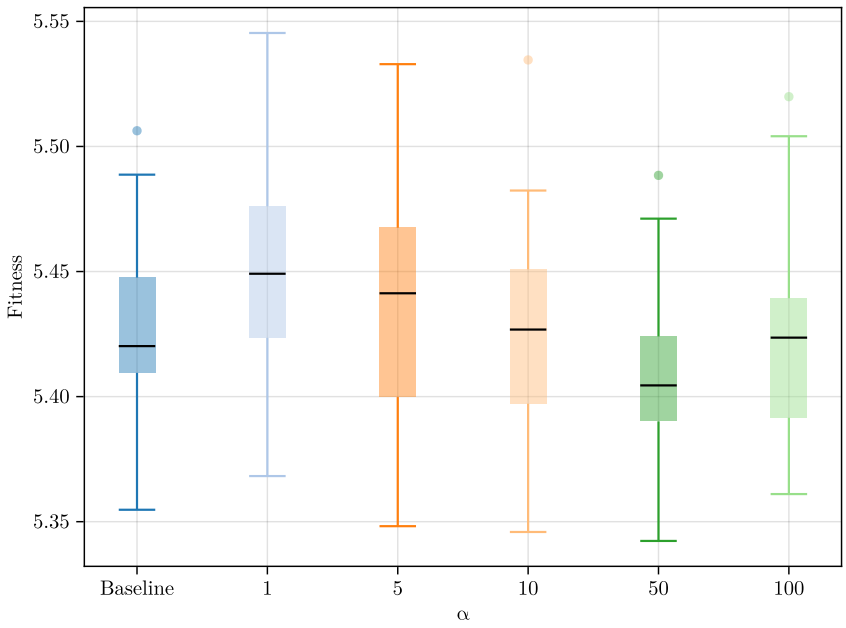


Figure 6.1: The fitness score of the genetic algorithm after 10 generations with the use of custom fitness function I.

0 in all tested cases. This experiment has been performed for 35 individual times for all different values for α.

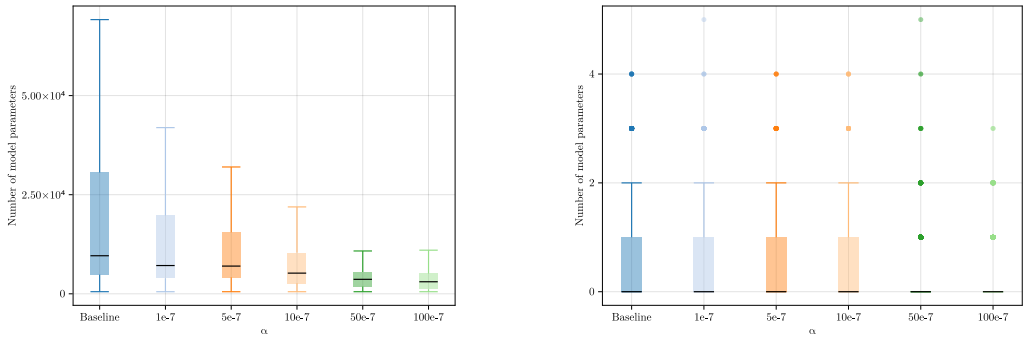
Aiming to disprove the hypothesis that all custom fitness parameter α measurements are the same as the baseline values for convergence, parameter count, and unintuitive design choices. A Kruskal-Wallis [60] test will be conducted on the measurements in order to gauge if any of the measurement distributions differ from the complete group. If found to be the case that there are differences between all the measurement sets in the group, a post-hoc pairwise comparisons using the Mann-Whitney [61] test will be conducted to find and prove if results are (dis)similar.

The Kruskal-Wallis calculated p-value for the convergence metrics is 0.0110 for the custom fitness function I, suggesting that the custom parameter α has for certain values enough of an impact to shift the distribution.

Table 6.1: The two-sided p-values for the fitness measurements of custom fitness function I calculated with the Mann-Whitney U test against the baseline.

α	p-value	H_0 rejected
1	0.3813	Failure
5	0.0116	Success
10	0.9679	Failure
50	0.3625	Failure
100	0.0302	Success

Making use of the Mann-Withney test allows us to test the individual α parameter values with the baseline against the hypothesis H_0 stating that both groups of measurements share the same distribution and therefore have no significant difference in medians. Table 6.1 contains the p-values calculated for the α parameter values against the baseline, showing that for both α parameter values 5e-7 and 100e-7 the impact is great enough to reject H_0. Comparing that with the values shown in Figure 6.2 it can be easily surmised that for α being equal to 5e-7 the fitness score is positively impacted and shows better performing numbers. In contrast, when α is equal to 100e-7 the fitness score is seen to be shifted negatively enough to reject the hypothesis.



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.2: The unneeded complexity metrics after 10 generations when making use of custom fitness function I of the population.

The effectiveness of the usage of using custom fitness function I in curbing some of the tendencies to not reduce the amount of unneeded complexity in the genome population can be best described by measuring and comparing the metrics of unneeded complexity. In Figure 6.2 both the number of parameters in the models for each genome in the population at the final generation was calculated and recorded, and the number of unintuitive design choices were recorded.

Table 6.2: The two-sided p-values for the parameter count (A) and unintuitive design choice (B) measurements of custom fitness function I calculated with the Mann-Whitney U test against the baseline.

α	p-value	H_0 rejected
1	0.0000	Success
5	0.0000	Success
10	0.0000	Success
50	0.0000	Success
100	0.0000	Success

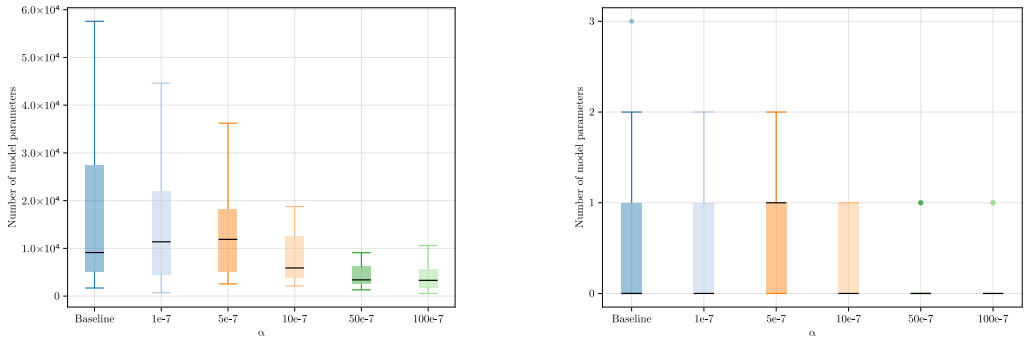
(a)

α	p-value	H_0 rejected
1	0.0055	Success
5	0.9204	Failure
10	0.0000	Success
50	0.0000	Success
100	0.0000	Success

(b)

Both the p-values calculated for the parameter count and unintuitive design choices measurements when calculated for the Kruskal-Wallis was 0. The individual measurements for the Mann-Withney tests can be found in Table 6.2. The values calculated in Figure 6.2

show that lower values for α (i.e. 1e-7, 5e-7, and 10e-7) seem to give better values for the parameter count metrics while incidentally also reducing the number of unintuitive design choices. The larger values for α (i.e. 50e-7 and 100e-7) do not seem to benefit either the parameter count, unintuitive design choice, and convergence metrics.



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.3: The unneeded complexity metrics after 10 generations when making use of custom fitness function I of the best performing genomes in the population.

The same calculations were then repeated for the single best performing solution in the population at the end of 10 generations. This limits our number of data points available and lets us focus on the most important genome and final result in a genetic algorithm. The results of which can be found in Figure 6.3 showing similar but more concentrated results. Reducing the number of outliers.

The figure shown in Figure 6.2 for the unintuitive design choice metric shows all the options considered for parameter value. However, the values presented in the figure related to the median are either very close or similarly close to zero. To distinguish which of the custom fitness function I measurement distributions can be considered the best in reducing the unintuitive design choices in the global population, the different populations where pitted against each other. First, the α' confidence is the modified Bonferroni correction confidence, done so to account for the multiple groups used in the calculation of the best performing group. The overall significance level α of 0.05 in use for rejecting the m null hypothesis's for each of the groups. This results in the following equation for calculating the corrected significance level α':

$$\alpha' = \frac{\alpha}{m} \tag{6.1}$$

Then each of the groups is pitted against the others comparing the one-sided lesser p-value against the corrected α' tallying the number of rejections of the lesser null hypothesis. This results in a table as seen in Table 6.3 where the time that a measurement group could be considered as having rejected the hypothesis on the lesser side with a singe-sided p-value less than the corrected significance level α'.

From the values displayed in Table 6.3 the higher values for α (i.e. 10e-7, 50e-7, and 100e-7) stand out as better performing as their peers. Surprisingly, the baseline value zero scored the worst of all options with the lower α values scoring but in one metric some improvement over the baseline.

Table 6.3: The tallied number of better performing instances per parameter α when compared to all other options including the baseline value of zero with (I) signifying the population metrics and (II) denoting the metrics for the best performing genome in the population.

	0	1	5	10	50	100
Parameter count (I)	0	0	0	3	4	5
Parameter count (II)	0	0	0	0	4	4
Unintuitive designs (I)	0	1	1	3	4	5
Unintuitive designs (II)	0	0	0	3	4	5

6.2 Unintuitive design choices

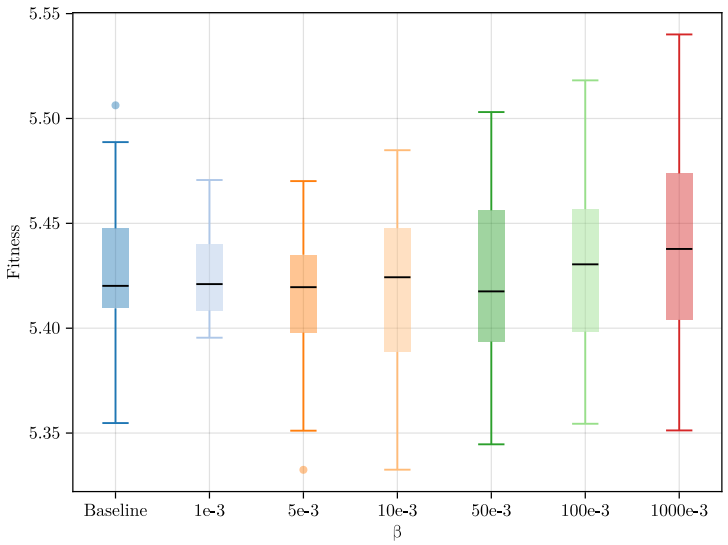


Figure 6.4: The fitness score of the genetic algorithm after 10 generations with the use of custom fitness function II.

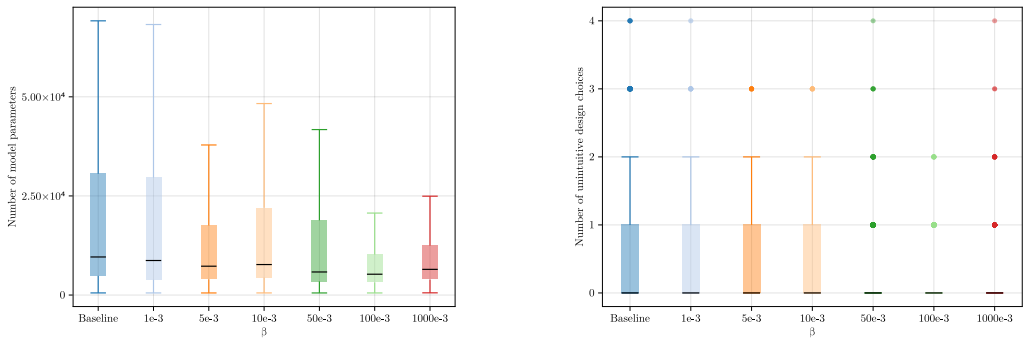
Similarly to the findings for custom fitness function I, for custom fitness function II the Figure 6.4 displays the final fitness performance of the best performing solutions in the then current genetic population. These numbers have been acquired after running the genetic algorithm for 10 generations with a custom fitness function as defined in Equation 4.1 and substituting all values for β as described with those in Table 4.3 with α being equal to 0 in all tested cases. This experiment has been performed for 35 individual times for all different values for β. The values for α have been chosen for extended runs in order to better observe the impact on the convergence metrics. The β parameter values have not been selected as the impact on convergence was found to be negligible and could be said to originate from the same distribution.

Unlike the measurements obtained for custom fitness function I, the impact as seen in Figure 6.4 on the fitness score does not seem to differ when confronted with custom

Table 6.4: The two-sided p-values for the fitness measurements of custom fitness function II calculated with the Mann-Whitney U test against the baseline.

α	p-value	H_0 rejected
1	0.9016	Failure
5	0.4062	Failure
10	0.4590	Failure
50	0.5606	Failure
100	0.8924	Failure
1000	0.2822	Failure

fitness function II. The p-value of 0.6222 acquired after running the fitness scores through the Kruskal-Wallis test then too is not enough to reject the null hypothesis H_0 stating that the groups are from the same distribution. Therefore, the individual Mann-Withney p-values as seen in Table 6.4 are also not able to reject the null hypothesis H_0.



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.5: The complexity metrics after 10 generations when making use of custom fitness function II of the population.

The parameter values for β impact on the complexity metrics can be seen in Figure 6.5. They show that the impact on the intended goal of reducing unintuitive design choices for the lower options (i.e. 1e-3, 5e-3, and 10e-3) is palpable with some outliers on the complete population. For the smallest β values (i.e. 1e-3 and 5e-3) they seem to have an impact on the parameter count of the model in the population after 10 generations.

The measurements gathered of the complete population have also been considered for a null hypothesis to ascertain a difference in all the complexity metrics. The Kruskal-Wallis tests for both complexity metrics returned 0.0000, and the individual low scoring p-values for the individual groups have been collected in Table 6.5.

Figure 6.5 and Figure 6.6 display the complexity metrics for both the complete population and the best genome in each of the data point population. These result show an overall lesser impact at all to the overall fitness progression of the genetic algorithm and a minimal impact on the unneeded complexity of the population.

The numbers presented in the figures of Figure 6.5 and Figure 6.6 show for some of the parameter choices β similar scores. As done with custom fitness function I, the groups were

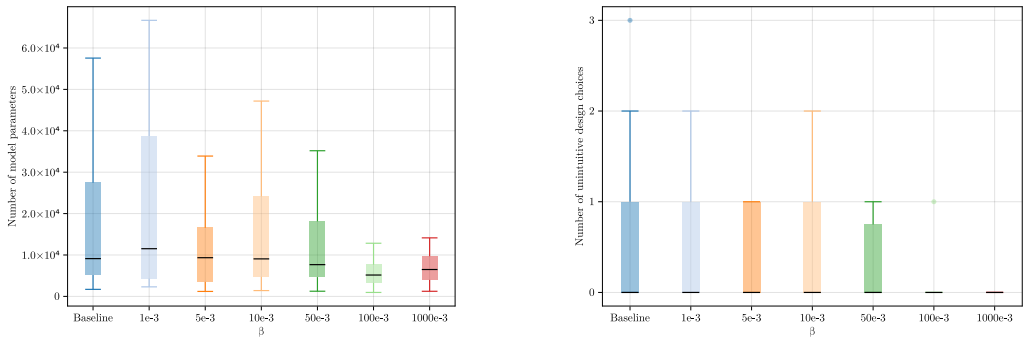
Table 6.5: The two-sided p-values for the parameter count (A) and unintuitive design choice (B) measurements of custom fitness function II calculated with the Mann-Whitney U test against the baseline.

α	p-value	H_0 rejected
1	0.0320	Success
5	0.0000	Success
10	0.0000	Success
50	0.0000	Success
100	0.0000	Success
1000	0.0000	Success

(a)

α	p-value	H_0 rejected
1	0.0000	Success
5	0.0000	Success
10	0.0000	Success
50	0.0000	Success
100	0.0000	Success
1000	0.0000	Success

(b)



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.6: The unneeded complexity metrics after 10 generations when making use of custom fitness function II of the best performing genomes in the population.

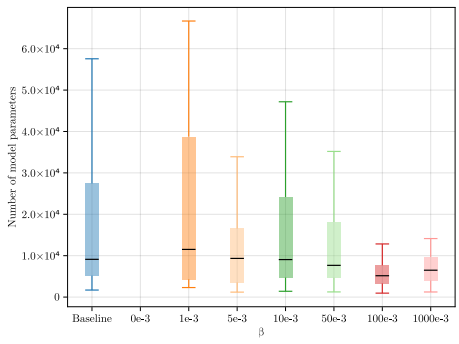
Table 6.6: The tallied number of better performing instances per parameter β when compared to all other options including the baseline value of zero with (I) signifying the population metrics and (II) denoting the metrics for the best performing genome in the population.

	0	1	5	10	50	100	1000
Parameter count (I)	0	0	4	1	4	6	4
Parameter count (II)	0	0	0	0	0	2	0
Unintuitive designs (I)	0	1	1	2	4	5	5
Unintuitive designs (II)	0	0	0	0	0	5	5

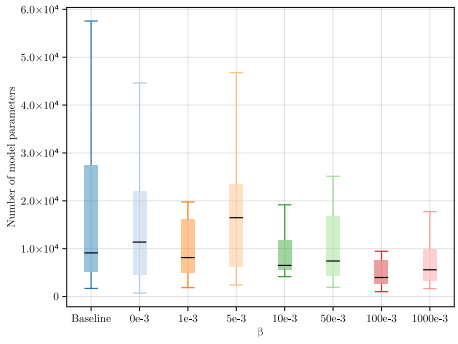
pitted against each other in order to highlight the better scoring groups on the metrics. These tallied rankings can be found in Table 6.6 and from their the impression can be gathered that once more the better scoring metrics come from the higher β values. The β value of 100 seems to score the most reliably and best over the different metrics and measurements.

6.3 Composite custom fitness function

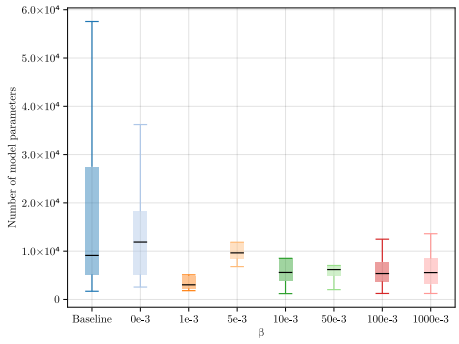
As with custom fitness function I and II for custom fitness function III results are composed of 35 runs for 10 generations of the genetic algorithm. The complete overview of complexity



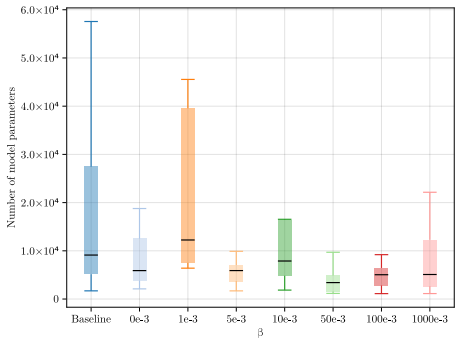
(a) α = 0e-7.



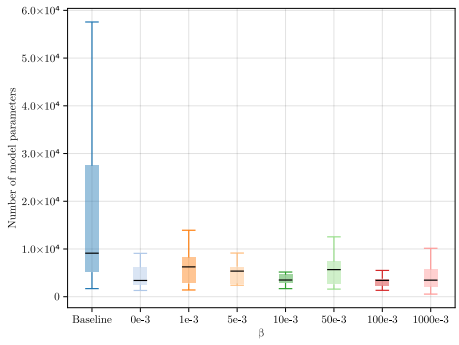
(b) α = 1e-7.



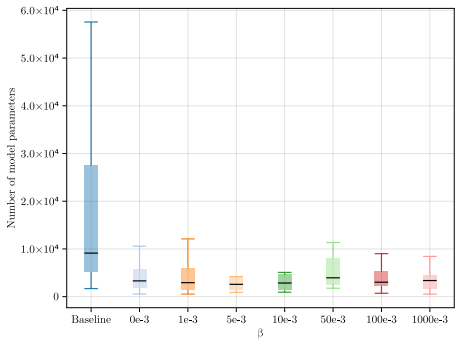
(c) α = 5e-7.



(d) α = 10e-7.



(e) α = 50e-7.



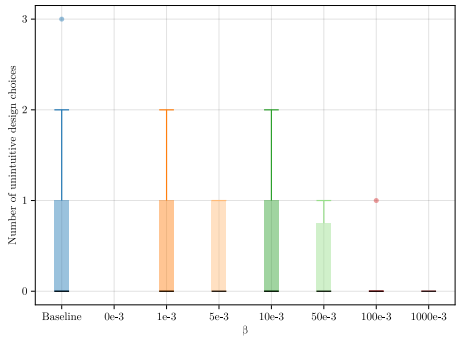
(f) α = 100e-7.

Figure 6.7: The number of model parameters in the end result of the genetic algorithm after 10 generations with the use of custom fitness function III.

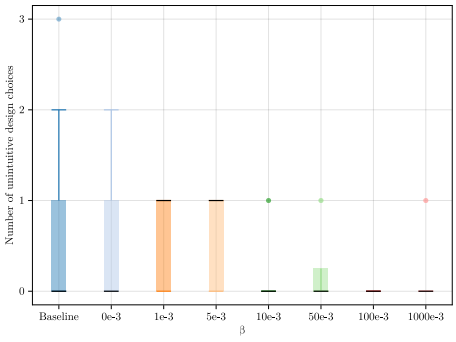
impact for all combinations of α and β can be seen in Figure 6.7 and Figure 6.8 where each of the combinations are laid out per plot such that each plot represents a constant for α and the differing values are from the β parameter options. The figures for the genetic algorithm related to both the convergence and complexity of the population metrics can be found in the Appendix E.

Figure 6.7 showcases the impact of parameter β when taken together with parameter α. The higher values of β (i.e. 50e-3 and 100e-3) consistently give stronger results than the lower values and extremes. However, while more predictable over the scala of α options the lower values for β (i.e. 5e-3 and 10e-3) seem to score better on the higher values of α when scoring it on the numbers of parameters of the end result.

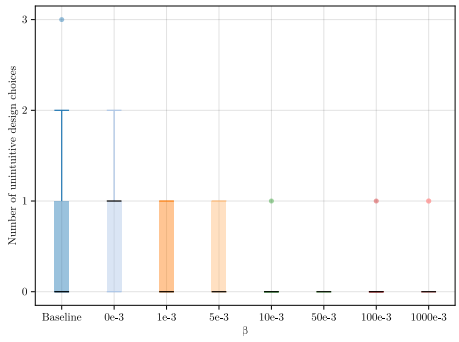
Figure 6.8 presents the combined impact of the α and β parameter values on the number of unintuitive design choices in the end result of the genetic algorithm when run for ten generations. The higher values of β (i.e. 50e-3, 100e-3, and 100e-3) seem to have significant impact on reducing the number of unintuitive design choices regardless of the α parameter chosen.



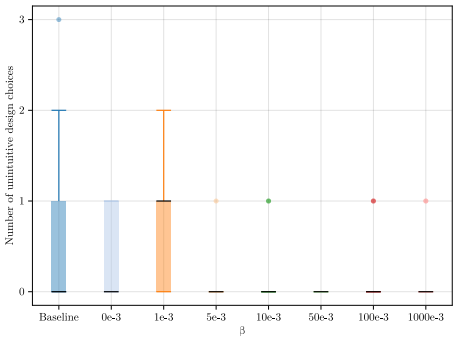
(a) α = 0e-7.



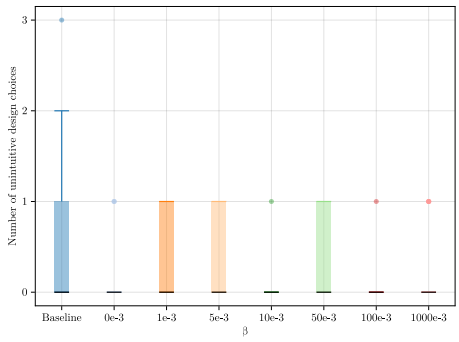
(b) α = 1e-7.



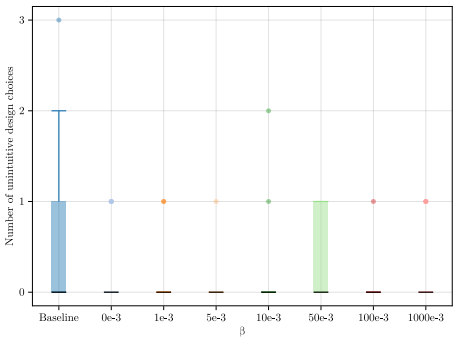
(c) α = 5e-7.



(d) α = 10e-7.



(e) α = 50e-7.



(f) α = 100e-7.

Figure 6.8: The number of unintuitive design choices in the end result of the genetic algorithm after 10 generations with the use of custom fitness function III.

6.4 Most promising parameters continued

From the results provided in the above sections regarding the different custom fitness functions, a number of most interesting custom fitness function combinations have been selected for extended review. Those selected have been chosen on criteria of most promising in increasing the quality of best genomes in the population in both the unneeded complexity and fitness score metrics.

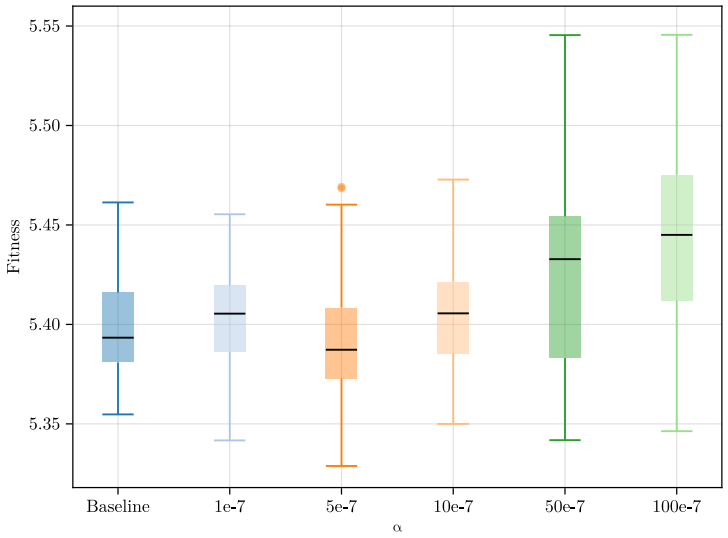
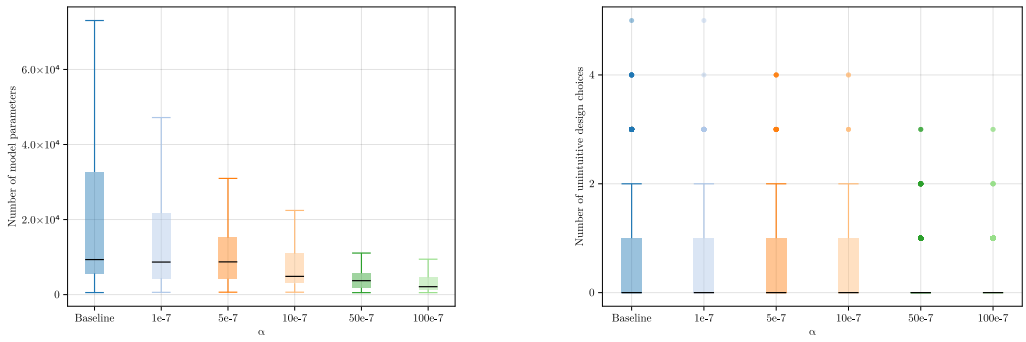


Figure 6.9: The fitness score of the genetic algorithm after 20 generations with the use of custom fitness function I.

Figure 6.9 shows a more in depth exploration of certain parameter combinations taken from 35 runs of the genetic algorithm for 20 generations. From these guidelines stated above, the custom fitness functions parameters of custom fitness function I were chosen as the most interesting to explore further. The parameter α of custom fitness function I deals with the impact of the size of the neural network in parameter count. As both seen in Figure E.1 and Figure 6.4 the second custom fitness function (II) parameter β is stable when viewed as the fitness score over time. Whereas, custom fitness function I does have some variation in the fitness scores based on what value has been chosen for parameter α.

The fitness score of the best performing genome in the population experiences a negative shift up when dealing with high α values of 50e-7 and 100e-7.

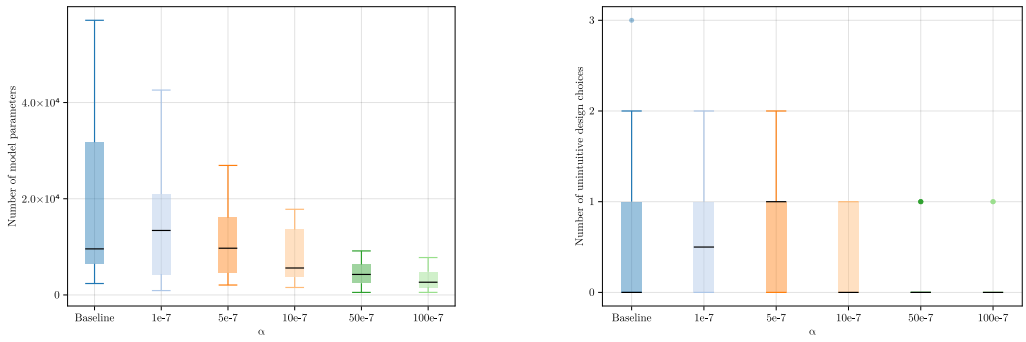
Figure 6.10 in contrast does show that the higher values for α do impact the parameter count and unintuitive design choices. From this one can deduct that while custom fitness function I does not aim to directly reduce the unintuitive design choices, higher values for α do definitely reduce the probability that genomes with unintuitive design choices are propagated. This also suggest that because the size of neural networks in layers (and therefore parameters) is often related to the number of possible unintuitive design choices.



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.10: The unneeded complexity metrics after 20 generations when making use of custom fitness function I of the population.



(a) Parameter count.

(b) Unintuitive design choices.

Figure 6.11: The unneeded complexity metrics after 20 generations when making use of custom fitness function I of the best performing genomes in the population.

Chapter 7

Premature convergence

These results in this section were collected as part of experiments conducted with the goal of answering research question RQ3. The three anti-early stagnation strategies defined in Section 4.5.3 have been implemented for this experiment and research question. With the three scenarios implemented, 30 runs of the genetic algorithm for all three scenarios were conducted for the length of 75 generations and a population size of 50.

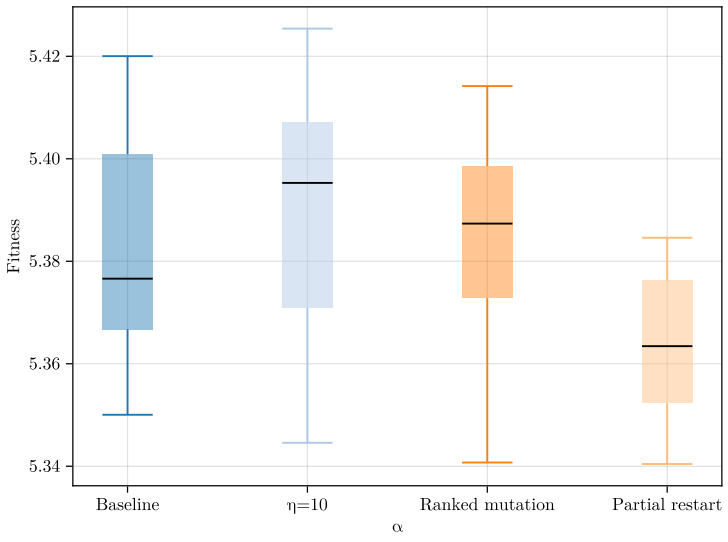


Figure 7.1: The fitness score of the genetic algorithm after 75 generations with differing anti-stagnation strategies employed.

The fitness scores as recorded –and seen in Figure 7.1– of the genetic algorithm after 75th generations give the impression that all of them belong to the same distribution. That would mean that the anti-premature convergence strategies have been unsuccessful in improving fitness scores. In order to test if this null hypothesis is indeed correct a test has been done in order to reject this hypothesis. The Kruskal-Wallis test was conducted and the resulting p-value of 0.3109 for the different strategies and the baseline fail to reject the null hypothesis with a confidence level of 95%.

This is in contrast with the expectations set by Figure 7.2, these do give the impression

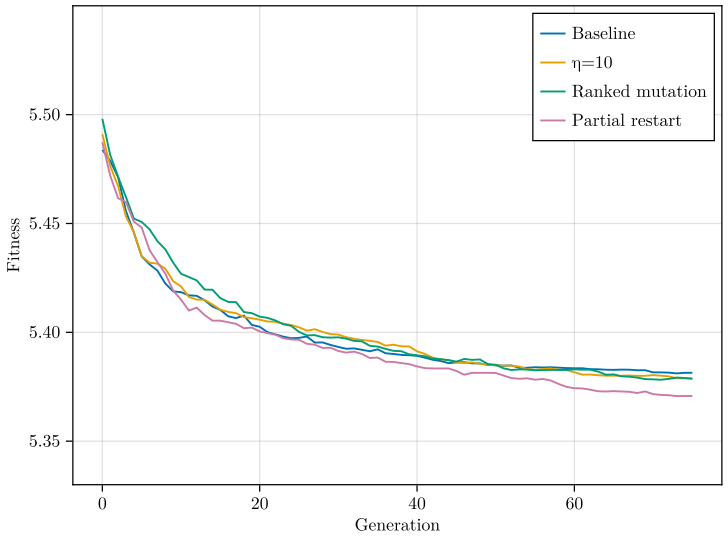


Figure 7.2: The mean fitness score of the genetic algorithm over 75 generations with differing anti-stagnation strategies employed.

of having an sizable impact on the best performing fitness score.

Unlike the impact on convergence metrics the different anti-premature convergence combatting strategies do seem to excel in maintaining the diversity levels. The upkeep of diversity levels using the anti-premature convergence strategies can be seen in Figure 7.3.

However, this claim requires the statistical assurance that indeed these samples do not share the same distribution. The diversity results too were subjected to the Kruskal-Wallis test to prove that they do not belong to the same distribution. The diversity scores at the 75th generation –as seen in Figure 7.3– represent the mean distance between genomes, and serve as the basis for our analysis. These values show how well the genetic algorithm has been able to maintain diversity. If the Kruskal-Wallis test is not able to disprove null hypothesis H_0, and that they share a distribution.

The calculated p-value for the diversity of measurements is 0.0625. While unable to prove that all samples do not share the same distribution, the p-value itself here is rather low. This might hint that a single strategy might differ enough from the baseline configuration.

The final part of this chapter concerns itself with the scoring of the individual anti-premature convergence strategies against the baseline configuration for both the convergence and diversity metrics. For this each of the individual results belonging to an anti-premature convergence strategy was compared to the baseline configurations for both the convergence and diversity. The Mann-Whitney U test performs each of the comparisons. Table 7.1 displays these results from the test. For the convergence metrics, none of the strategies would result in string enough metrics. Metrics able to reject the null hypothesis H_0 claiming that they share the same distribution. Although, the p-value for the partial

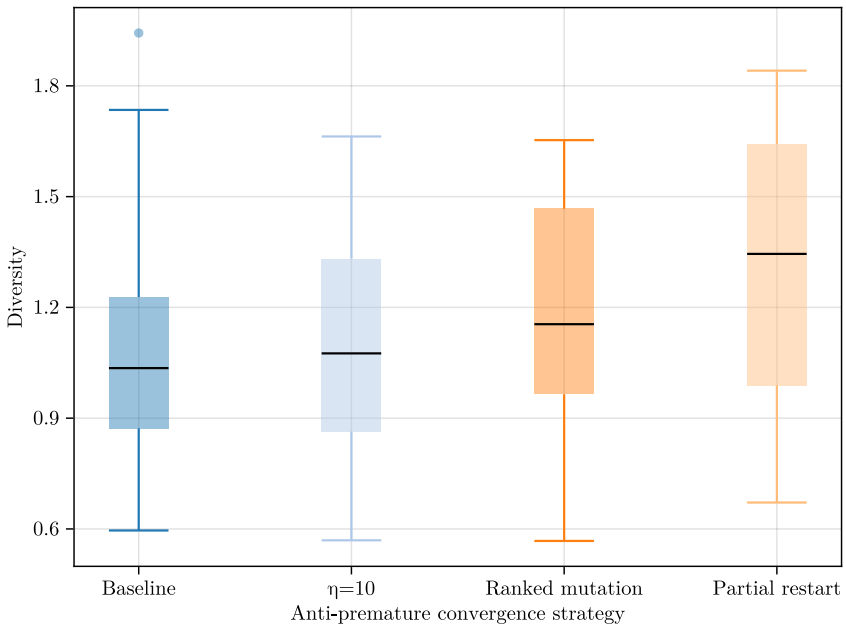


Figure 7.3: The population diversity measured as the mean distance between genomes per generation after 75 epochs with differing anti-stagnation strategies employed.

restart strategy was substantially lower than the other strategies.

In contrast, when analysing the diversity metrics in the same way, the partial restart strategy can claim improvements over the baseline. The other strategies fail to reject the null hypothesis with values higher than 0.0500. The partial restart strategy is able to reject the null hypothesis with a p-value of 0.0106. Thereby, proving the partial restart strategy as a viable method of maintaining diversity within the population.

Table 7.1: The two-sided p-values for the fitness score (A) and mean distance (B) measurements of the anti-premature convergence strategies, calculated with the Mann-Whitney U test against the baseline.

Strategy	p-value	H_0 rejected
$\eta = 10$	0.7620	Failure
Ranked mutation	0.9607	Failure
Partial restart	0.1264	Failure

(a)

Strategy	p-value	H_0 rejected
$\eta = 10$	0.7089	Failure
Ranked mutation	0.2395	Failure
Partial restart	0.0106	Success

(b)

Chapter 8

Combined result

This last chapter of the results, will combine the best elements of the alterations made in the previous selection into one new genetic algorithm. These alterations based upon the previously collected results to the genetic algorithm will be tested against the original NASCTY algorithm. For this 10 runs of 75 generations with a population size of 100 will be conducted on the synchronized ASCAD dataset before moving onto the desynchronised dataset at desynchronisation level 50 and 100. The choice to test the modified algorithm on the fixed key dataset on desynchronisation level 0, 50, and 100 together with the synchronised variable key ASCAD dataset, comes down it being common datasets. From these runs the best end results will be filtered from the results and trained for a longer period of time. The best possible solution for each situation is then evaluated on the attack section of the ASCAD dataset.

8.1 Selected modifications

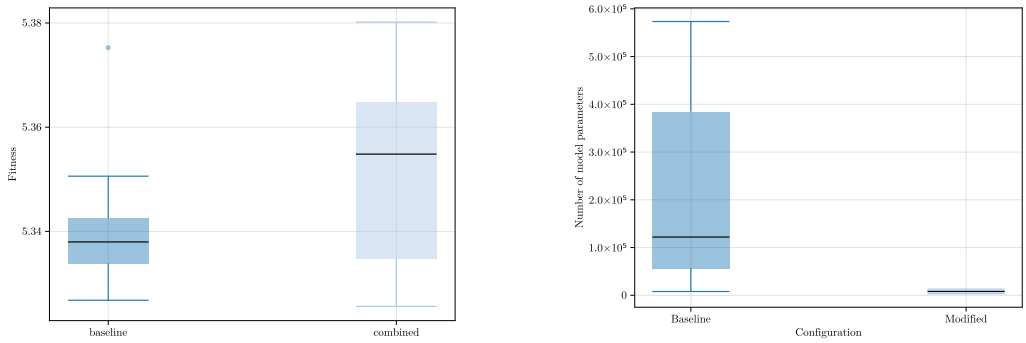
The selected alterations for the modified NASCTY genetic algorithm come as a result from the findings of Chapter 5, Chapter 6, Chapter 7 along with the discoveries done discussing the results in Chapter 9.

From Chapter 5 the early termination policy and the accompanying threshold values for the conservative approach are taken.

From Chapter 6 the custom fitness function as defined in Equation 4.1 is taken and implemented. The chosen values for parameter α 5e-7 and β 100e-3 based upon the complexity minimising - convergence cost trade-off. Chapter 9 delves into the reasoning behind the parameter value choices in further detail.

From Chapter 7 the partial replacement is taken as the anti-premature convergence strategy. None of the proposed strategies for combatting premature convergence performed well enough –or worse enough– for the results to be considered significantly better than the baseline. However, the partial replacement does seem to maintain a higher level of diversity than the baseline. In theory a higher level of diversity might make it easier for a genetic algorithm to escape a local optimum. For the full array of augments and reasons, Chapter 9 contains the more points of discussion.

8.2 Synchronised fixed key ASCAD dataset



(a) Fitness.

(b) Parameter count.

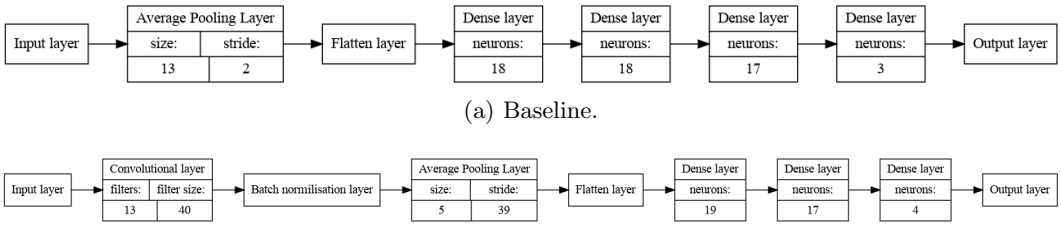
Figure 8.1: The algorithm end result's fitness score (A) and number of parameters (B) when dealing with the fixed key non-desynchronised ASCAD dataset.

The first benchmarking dataset considered is the fixed key non-desynchronised ASCAD dataset. It is the easiest of the datasets to attack from benchmarking datasets considered.

This benchmark has served as the source dataset for ten runs. These ten runs, were run both using the stock configuration of the NASCTY algorithm and contrasted against the proposed modified algorithm.

Every run the population over time was recorded. From these the most important gene for our purposes is the best performing gene in the last generation.

Figure 8.1 shows us the fitness score and number of model parameters. It shows a slight sacrifice in fitness score stability while gaining a great reduction of the model parameters.



(a) Baseline.

(b) Modified.

Figure 8.2: The best genome designs produced by (A) baseline configuration and (B) modified algorithm for the non-desynchronised ASCAD dataset.

From the ten benchmarkings, the best performing (i.e. the lowest fitness score value) end result is selected for further training. Figure 8.3 shows the two runs from the end result was chosen. These genomes get an additional 40 training epochs, for a total of 50. The same figure shows a difference in run time characteristics. The original algorithm displays a stable progression plateauing just after the twentieth generation. In contrast, the modified configuration is more likely to have temporary regressions in the its run. It is not in comparison with the original algorithm as likely to get stuck in local optima.

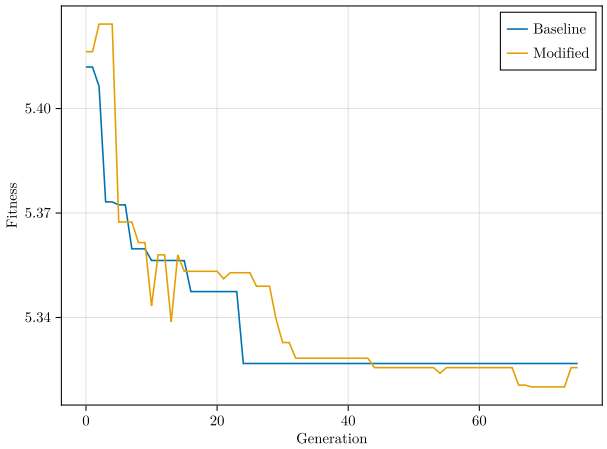


Figure 8.3: The fitness progressions of the best runs for the stock and modified configuration for the non-desynchronised ASCAD dataset.

Figure 8.2 two best performing genomes and their design. The baseline configuration algorithm came up with a multilayer perceptron design with 7953 model parameters. In the testing done in this paper this model was able to obtain a key rank of 0 after 150 traces.

The genome proposed by the modified algorithm is a smaller convolutional algorithm coming in with a 6716 model parameters. It requires only 88 traces to obtain a mean key rank of 0.

Figure 8.4 shows similar findings with the convolutional neural network achieving the key rank 0 quickly and the baseline proposed solution failing to do so.

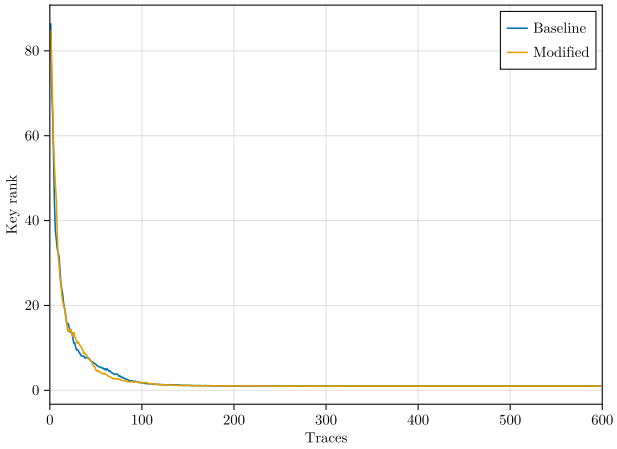
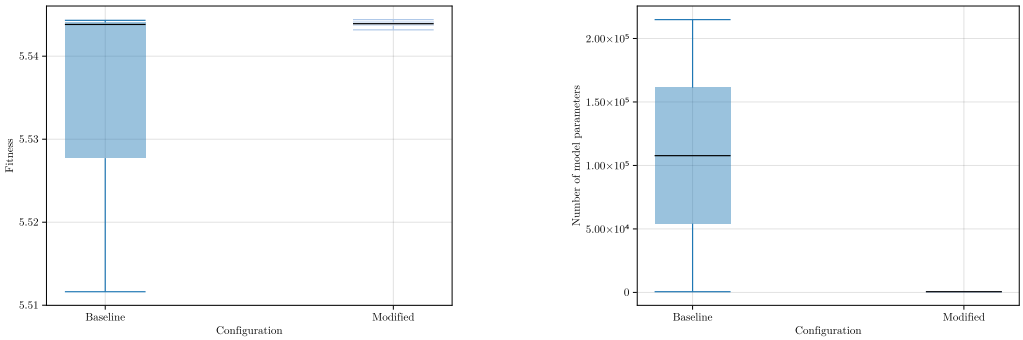


Figure 8.4: The mean performance of the best performing neural network result over 100 folds for the non-desynchronised ASCAD dataset.

8.3 Desynchronised level 50 fixed key ASCAD dataset



(a) Fitness.

(b) Parameter count.

Figure 8.5: The algorithm end result's fitness score (A) and number of parameters (B) when dealing with the fixed key level 50 desynchronised ASCAD dataset.

The benchmark runs for level 50 desynchronised fixed key ASCAD dataset measurements number five total. These runs show the modified algorithm struggling.

As with the previous measurements for the non-desynchronised dataset, the modified algorithm is successful in the reduction of model parameters in the end result. However, as seen in Figure 8.5 the modified algorithm struggles with finding promising solutions. The original algorithm did not have any problems in finding solutions for this problem. It is as also rather uncaring as to the size of the end result. This is tendency is displayed in both the results for non-desynchronised dataset and the easier synchronised fixed key dataset.

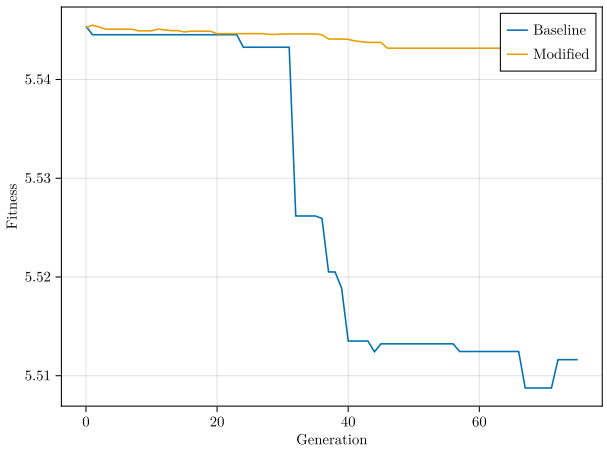
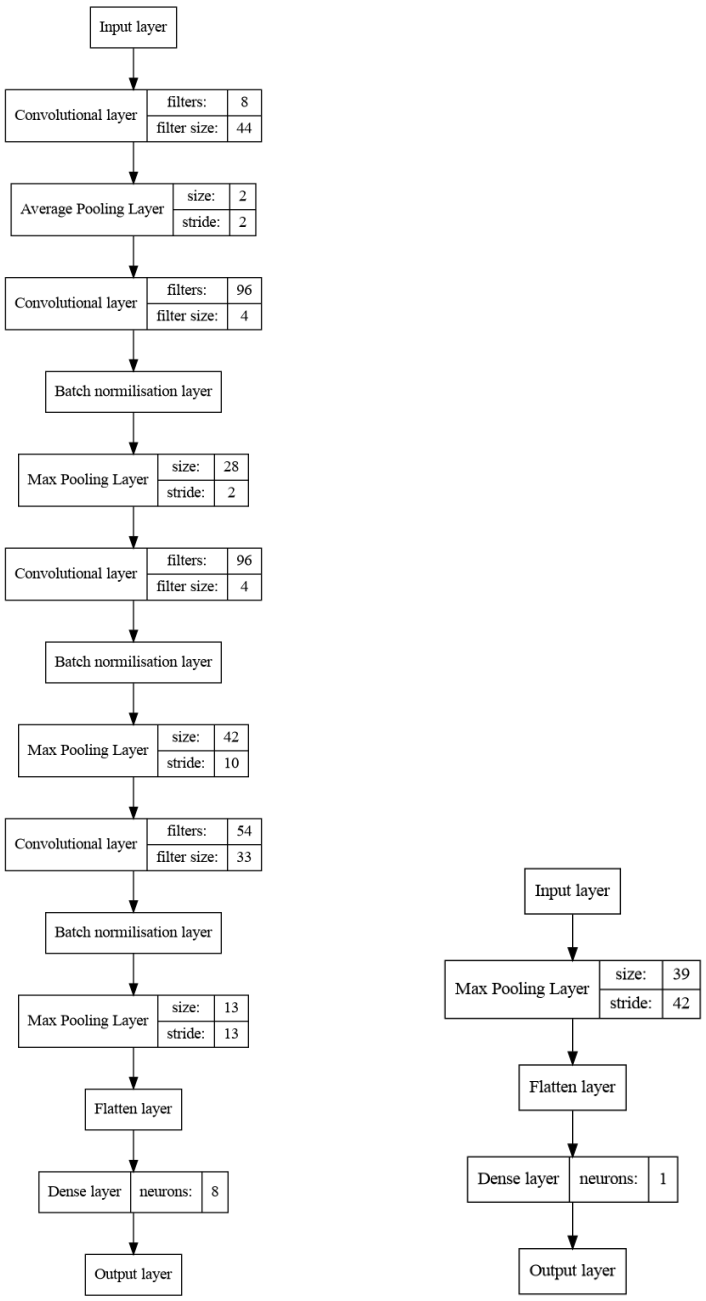


Figure 8.6: The fitness progressions of the best runs for the stock and modified configuration for the level 50 desynchronised ASCAD dataset.

Figure 8.6 reflects similar as the previous results. The modified algorithm over its best

runs is barely able to improve its fitness scores.



(a) Baseline. (b) Modified.

Figure 8.7: The best genome designs produced by (A) baseline configuration and (B) modified algorithm for the fixed key level 50 desynchronised ASCAD dataset.

The genomes of that best run –seen in Figure 8.7– give a hint as to the nature of the modified algorithms struggles. The design chosen by the baseline configuration consists of four convolutional blocks before leading into but a single dense layer of eight neurons. It is able to achieve a mean key rank of 0 in 300 traces using 214 850 model parameters. The end result genome of the modified algorithm is the smallest possible multilayer perceptron design possible within the constraints of the NASCTY genome design. It fails to achieve a mean key rank of 0.

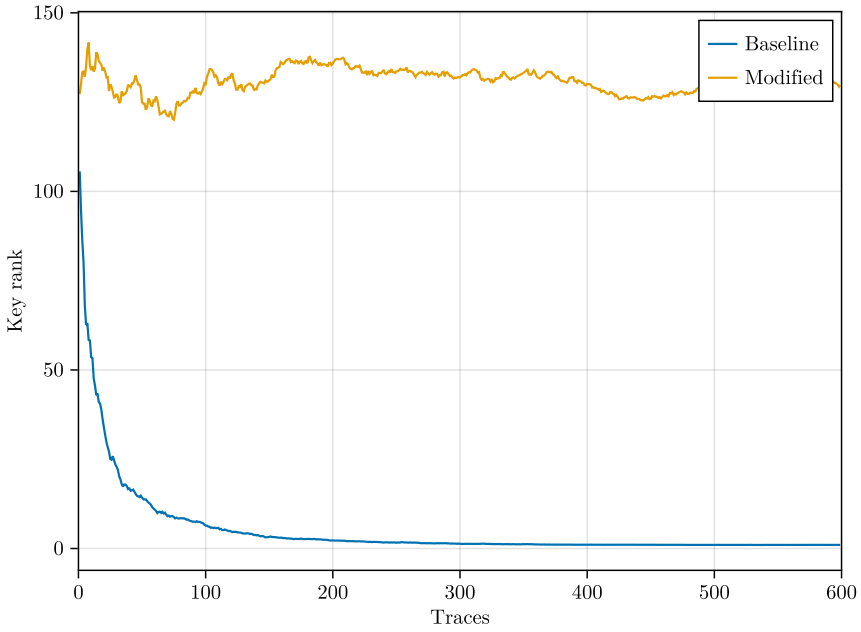
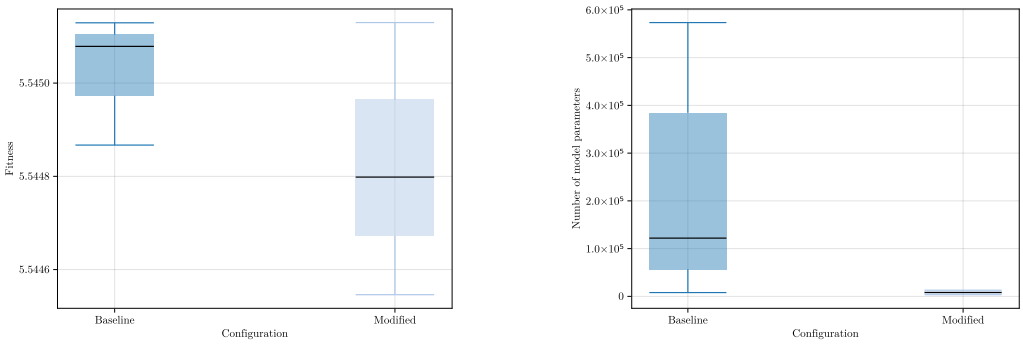


Figure 8.8: The mean performance of the best performing neural network result over 100 folds for the fixed key level 50 desynchronised ASCAD dataset.

8.4 Desynchronised level 100 fixed key ASCAD dataset



(a) Fitness.

(b) Parameter count.

Figure 8.9: The algorithm end result's fitness score (A) and number of parameters (B) when dealing with the fixed key level 100 desynchronised ASCAD dataset.

As with the fixed key level 50 desynchronised ASCAD dataset benchmark, the benchmark was run for five times. Figure 8.9 shows both configurations of the algorithm struggle with producing promising end results.

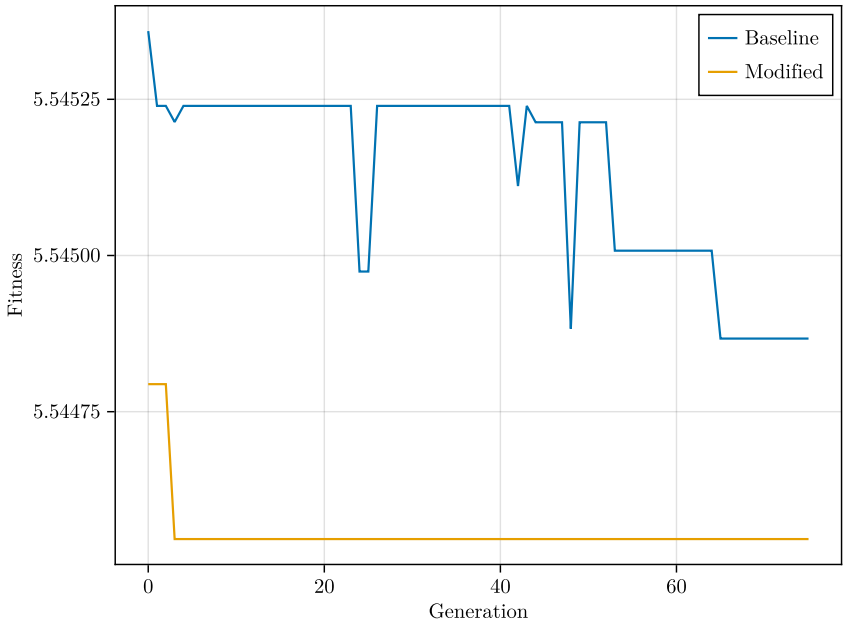
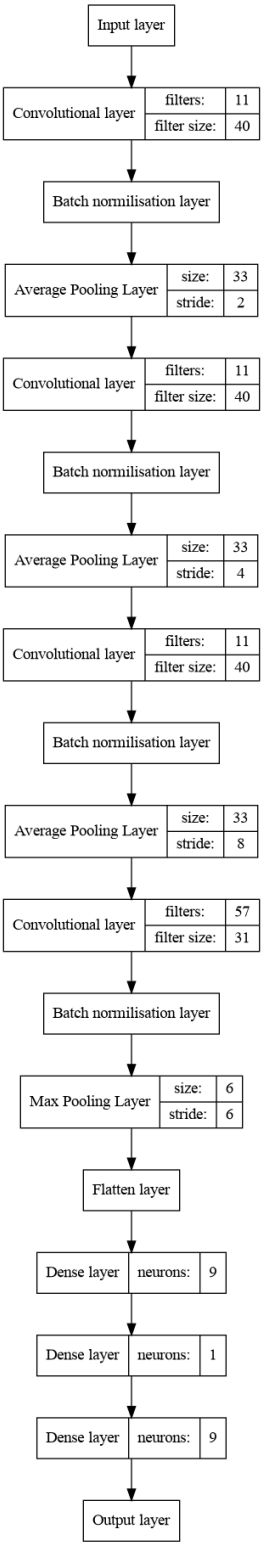


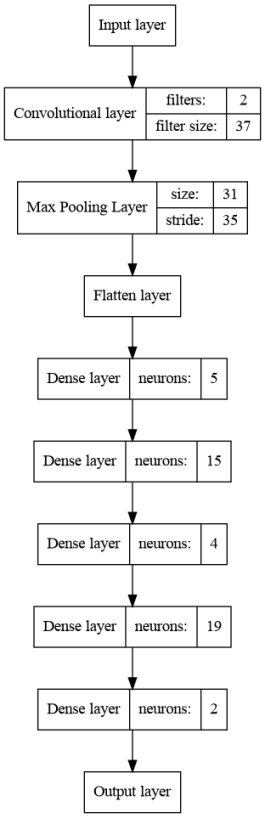
Figure 8.10: The fitness progressions of the best runs for the stock and modified configuration for the fixed key level 100 desynchronised ASCAD dataset.

The best runs –seen in Figure 8.10– of both configurations make almost no progress over time. Figure 8.11 shows two designs, the greatest commonality between the designs is the rather larger number of dense layer blocks. The modified algorithm's design has 1338 model parameters and the original algorithm's design has 32937. The larger model of the baseline configuration is able to obtain a mean key rank of 0 in 4962 traces. In contrast, the smaller modified algorithm's genome is unable to achieve a mean key rank of 0.

Figure 8.12 show that the performance expressed as key rank per trace count is poor when evaluated on the attack dataset. Both models hovering around the start position.



(a) Baseline.



(b) Modified.

Figure 8.11: The best genome designs produced by (A) baseline configuration and (B) modified algorithm for the fixed key level 100 desynchronised ASCAD dataset.

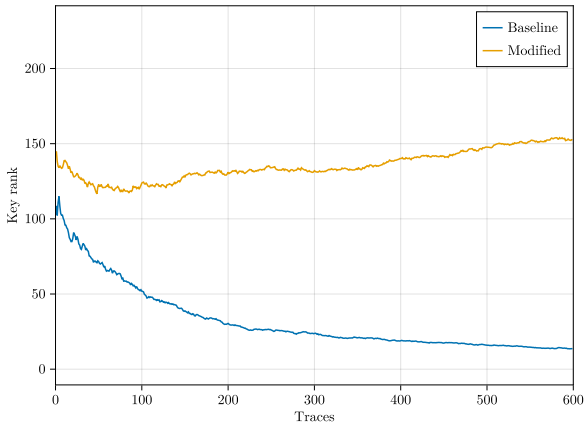
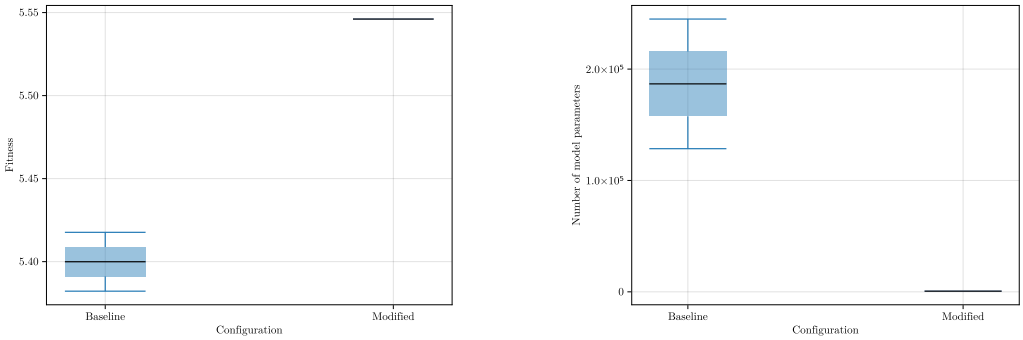


Figure 8.12: The mean performance of the best performing neural network result over 100 folds for the fixed key level 100 desynchronised ASCAD dataset.

8.5 Synchronised variable key ASCAD dataset



(a) Fitness.

(b) Parameter count.

Figure 8.13: The algorithm end result's fitness score (A) and number of parameters (B) when dealing with the variable key non-desynchronised ASCAD dataset.

The final dataset up for benchmarking is the variable key non-desynchronised ASCAD dataset. The benchmark was run for three full runs for both configurations. As with the other datasets and benchmarks, the modified is able to reliably create significantly smaller end results. Figure 8.13 shows the reduction in both end result size and performance.

Figure 8.14 also repeats similar findings. The baseline configuration is able to find well performing designs according to the validation set. The modified algorithm is unable to find any improvements over its runtime.

The design –as laid out in Figure 8.15– for both the original algorithm and for the

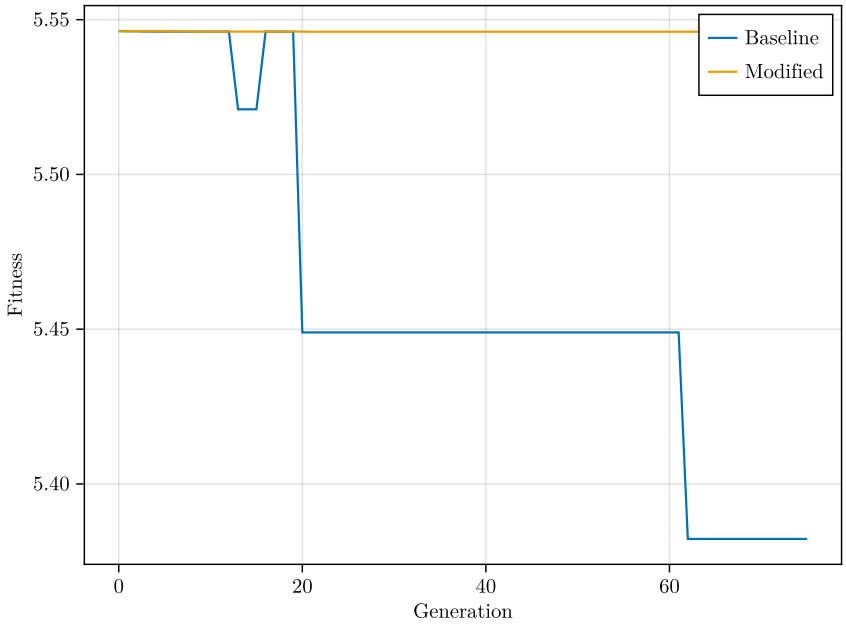
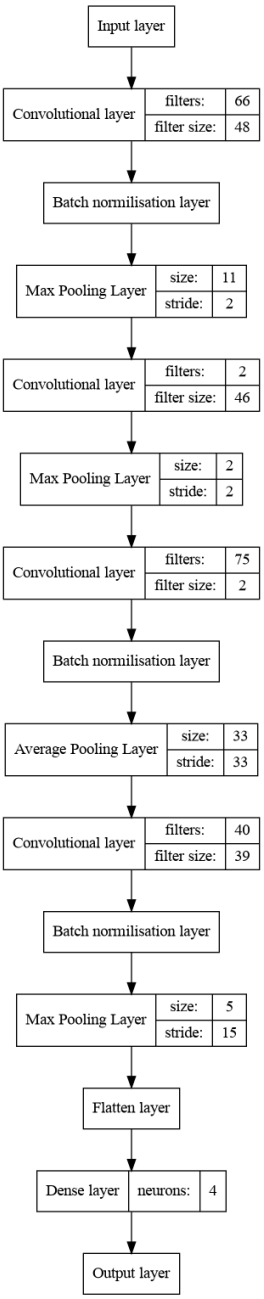


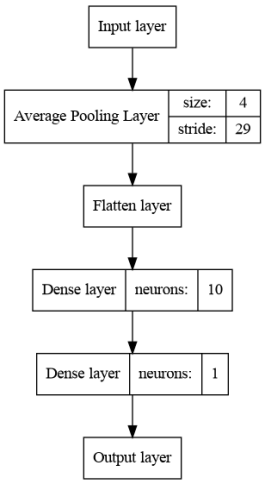
Figure 8.14: The fitness progressions of the best runs for the stock and modified configuration for the variable key non-desynchronised ASCAD dataset.

modified algorithm betray something of the internal turmoil and origin of the modified algorithms problems. The convolutional genome design seems to be formed directly as a response to the quest of finding the best performing algorithm. It is a 128 524 parameter design and unable to obtain a mean key rank of 0. In contrast, the multilayer perceptron design of the modified algorithm seems to be a direct response from the difficulty of finding a solution coupled with a strong complexity penalty. It is a 783 parameter design also unable to attain a mean key rank of 0.

From the numbers presented in Figure 8.16 the modified algorithm's design seems to outperform the original algorithm's design. This while the original convolutional network design scored much better in with the training and validation datasets.



(a) Baseline.



(b) Modified.

Figure 8.15: The best genome designs produced by (A) baseline configuration and (B) modified algorithm for the variable key ASCAD dataset.

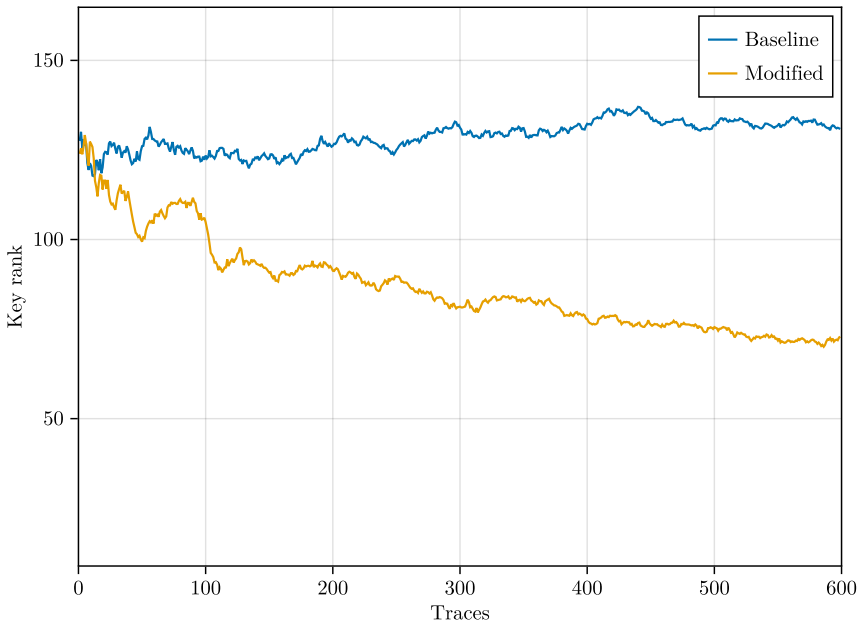


Figure 8.16: The mean performance of the best performing neural network result over 100 folds for the variable key non-desynchronised ASCAD dataset.

Chapter 9

Discussion

This chapter discusses and interprets the findings of the experiments results from Chapter 5, Chapter 6, Chapter 7, and Chapter 8 which where described in Chapter 4. The experiments in turn hoped to shine a light upon the research questions posed in Chapter 3. These discussions are to the basis from which the conclusions of this research are to be made in Chapter 10.

9.1 Hyperparameters

The results found in Chapter 5 are mainly concerned with improving upon the genetic algorithm hyperparameters with the goal of speeding up the time needed to conduct future experiments and runs in general. These findings came from four different experiments, and in turn were concerned with: evaluation time, implementing early stoppage policy for evaluating genomes, reducing the training data partition size, and reducing the size of the population of the genetic algorithm.

The first experiment shows that for accurately predicting the evaluation fitness score of a genome after 50 epochs, ten epochs are needed. The space between the tenth and twentieth epoch in prediction accuracy can be spoken of as plateauing. Continuing training for more than ten epochs would lead to a high increase of time spent evaluating for almost no gain.

Any reduction from the ten training epochs to less lead to a sharp decline in the prediction accuracy.

The second experiment was mainly concerned with reducing the wasted training epochs. For this it was conducting test to implement an early termination policy along with finding fitness threshold values triggering an early termination. For this two approaches were defined: a conservative approach which tries to limit the number of wasted training epochs without missing any promising solutions; and an optimised approach geared toward more strongly reducing training epochs by sacrificing the outliers.

The prospective reduction in training time measured in epochs aimed at the genomes which have at least an improvement $\Delta \geq 0.01$ is respectively $\approx 23\%$ and $\approx 33\%$ for the conservative and optimised approach. The results also show that for 10% improvement in time savings one loses $\approx 9\%$ of promising solutions.

The third experiment is related to reducing the training time per epoch. It tries to do this by limiting the training dataset partition size the neural network has access to.

The smallest of the partition sizes of 100 and 200 large produce unpredictable results. The larger partition size seems to scale in a linear fashion to the divergence from the starting training fitness score as they get larger. Ultimately, a partition size of 35600 matching the full size of the dataset is the most preferred choice. As any lower of a value scales the possible improvement down, this reduced divergence allows for less of an accurate prediction of performance. So, while smaller partition sizes promise significant reductions in training time in seconds per epoch it comes at the cost of losing fitness accuracy, predictability of results, and reliability making this not a worthwhile endavour to consider.

9.2 Complexity

The results found in Chapter 6 paint a complex picture. Ultimately, the right combination of α and β parameter values must be selected for the custom fitness function that reduce the redundant complexity while not impacting the convergence and search abilities of the genetic algorithm.

For the first parameter α higher values (i.e. 50e-7 and 100e-7) have shown to have a negative impact on the fitness score. The impact on parameter count scales with the strength given to α, with the strongest values also impacting the unintuitive choices metric. One can deduce from these metrics that the impact of the higher α values is so strong that it has a strong bias for selecting small networks that have few layers and therefore in turn fewer possible unintuitive choices. It hints at the possibility that the solutions considered for propagation into the next generation are significantly reduced. The lower values considered values for α in turn do not seem to reduce the search space in a fashion akin to the higher α values.

Therefore, the values considered for α as the improving the genetic algorithm in a positive manner while reducing the parameter count are on the lower end of the considered spectrum (i.e. 1e-7, 5e-7, and 10e-7).

All the values considered for parameter β, in contrast with the α parameter values considered seem not to posses enough strength to sway the fitness score of the genetic algorithm. The aimed for impact on unintuitive design choices can be seen reliably by all the β values excluding 1e-3 when viewed for the complete population. However, when zoomed in on only the best performing genome in the population only the β values of 100e-3 and 1000e-3 seem to have any significant impact. From the overview of the better solution the β value of 100e-3 seems to score the strongest on the metrics.

The trade-off of minimising the complexity on one hand and not degrading the convergence performance on the other hand steers us to certain preferred α and β parameter combinations for the custom fitness function.

All values for α have a visible impact on the parameter count recorded. Additionally, the higher values of α have noticeable negative impact on the convergence of the genetic algorithm. Therefore, the values 50e-7 and 100e-7 are discounted from further consideration as suitable parameter choices. For the β value, the choice is set on the dependable 100e-3 which scores in a predictable manner across any of the α combinations for the unintuitive design choice metric. This leaves the parameter count metric as the one to score the α choices on. One can turn to the custom fitness function III results collected to get a global overview of all the metrics of associated with these unknowns.

Taking in account these findings, the recommendation for the α parameter falls to 5-e7 and for the β parameter to 100e-3.

9.3 Premature convergence

The main point of discussion for the anti-premature convergence is the lack of success for improving on the genetic algorithm's ability to discover better solutions compared to the baseline. Even though the figures (as seen in Figure 7.2 and Figure 7.1) seem to suggest that the partial restart strategy is a successful avenue for improving the fitness score as a result of more easily being able to escape local optima. The statistical test performed on the results failed to reject the null-hypothesis. However, while the strategies failed to achieve provable improved convergence fitness score metrics, the diversity of the population maintained a higher level of diversity. Statistical tests proved that one strategy does not share a distribution with the baseline configuration. Proving it to be a successful strategy for maintaining the diversity in a population. The partial replacement strategy performed the best of the strategies in this metric followed by the ranked mutation strategy and lastly the strengthening of the mutation strength η from 20 to 10. The ranked mutation is not considered a good enough solution to the premature-convergence tendencies of the genetic algorithm as its requires an additional evaluation phase of the offspring in order to calculate the mutation rate of the genomes. It is discounted as a solution as the additional time spend ranking the offspring far outweighs any of the improvements seen to the diversity levels.

The partial restart strategy is the solution with the strongest impact on both metrics. It is therefore, the recommended strategy for maintaining diversity within the population.

9.4 Combined results

The benchmarking results clearly showcase the success and the improvements of the new algorithm design while simultaneously presenting the downsides of the new approach.

The first benchmark gives us strong results in both complexity and performance. The custom fitness function contributes to the improved end results.

It is the same custom fitness function that when dealing with more difficult datasets struggles. The smaller improvements of possible promising solutions are offset by the complexity penalties received. The modified algorithm then stops giving good and acceptable results. When it cannot optimise for the best solution in conjunction of reducing complexity it will settle for just reducing complexity. Figure 8.7 perfectly represents this conundrum, where the end result is simply the smallest possible design instead an acceptable performing design.

In case the other modifications made to the new algorithm might help outperform the original algorithm. Such opportunities might arise when dealing with more complicated datasets. The results found in such cases are not able to disclose the effectiveness of the anti-premature convergence on its own on those benchmarks.

Chapter 10

Conclusion

This is the final chapter of this thesis. In it the context of the thesis is discussed along with the limitations of this research. Followed by possible avenues for future work to investigate. The last part concludes this thesis with the main takeaways of this thesis' research.

10.1 Context

This thesis is written as an extension of the NASCTY algorithm. In it the aim was to explore the usage of automated search algorithms for designing neural network architectures matching or even outdoing the previously researched manually constructed neural networks. Specifically, this thesis is tries to overcome the limitations of the NASCTY algorithm respect to redundant complexity of its end results and the premature convergence tendencies of the algorithm.

To answer and solve these limitations a custom fitness function has been designed to combat the redundant complexity of the end results and a strategy has been implemented and tested to combat the premature convergence tendencies.

10.2 Limitations

This section discusses the limitations this thesis was faced with and in what forms it manifested. The greatest limitations came forth from the lack of computational time allotted to conducting the experiments. Even though the time allocated for this thesis is on the larger side, it does not take away the consumptive nature of this research.

One limitation of this thesis is in regard to the numerous α and β parameter combinations considered. If given more time, the certainty of the impact on convergence might be stronger if it had been possible to experiment for more generations.

Similarly, if more time had been available for testing an additional anti-premature convergence strategy, more strategies might have been implemented and tested.

Lastly, the modified genetic algorithm together with the unmodified NASCTY algorithm might have been able to run on the different ASCAD datasets for more than ten runs. That would allow for more and better end results, together with the establishing of a more clear and expected view of performance.

The limitations of the modified NASCTY algorithm are twofold. Starting with the smallest one, the current method of reducing the time spent evaluating genomes' pheno-

types is dependent on the used dataset. The fitness thresholding policy requires of the user that they calculate these thresholds before running the algorithm.

The second and more important limitation is that the complexity minimising approach proposed has not been as generalisable as aimed for. The algorithm when run with different datasets needs the optimal parameters for the custom fitness function. These must be acquired before running the algorithm. Currently, this is a tedious and time consuming activity.

10.3 Future work

As a compliment to the limitations section, this section will discuss the possible avenues if further future research works.

The newly proposed implementation suffers a large limitation. Its importance stems from its reliance on previously calculated values for the custom fitness function. The discovered parameter values for the fitness function seemed not as easily transferred to other datasets. One possible direction that could be taken to reduce the complexity of end results in a more dependable and generalisable way would by making use of a multi-objective optimization.

In addition, further research in the area of using genetic algorithms for automating neural network designs might focus on implementing other premature convergence combatting strategies.

10.4 Takeaways

This thesis has shown that a complexity-minimising approach is feasible direction for improving upon the exiting NASCTY algorithm.

Additionally, we have been able to implement a method for minimising the complexity of our results that still respects the efficacy of the algorithm. However, the custom fitness function has turned out to be not the ideal solution. It suffers from not being as generalisable to other problem sets as much as hoped. This is a possible point of improvement in further research.

The research put into alleviating the problems of premature convergence have delivered us a successful strategy. Partial replacement of the population when confronted with a plateauing run has shown to be good for maintaining population diversity.

Finally, benchmarking the modified algorithm against the baseline configuration has shown that this approach is an improvement in performance and complexity when instantiated with the proper custom fitness function parameters. In case of the fixed key non-desynchronised ASCAD, the reduction of needed traces for obtaining mean key rank 0 was roughly 71% while using 36% less neural network model parameters. Parameter combinations that have been proven on one dataset do not translate to another. These require careful selection for each new dataset. If not properly selected the algorithm performance will suffer greatly and select not for quality of solutions but for the complexity parameters of the possible solutions.

Bibliography

[1] G. S. Tranquillus, *De vita Caesarum*. 121 (cit. on p. 1).

[2] K. Goyal and S. Kinger, "Modified caesar cipher for better security enhancement," *International Journal of Computer Applications*, vol. 73, no. 3, pp. 0975–8887, Jul. 2013. DOI: 10.5120/12722-9558 (cit. on p. 1).

[3] N. P. Smart, *Cryptography Made Simple* (Information Security and Cryptography), 1st ed. Springer International Publishing, 2016, ISBN: 9783319219356. DOI: 10.1007/978-3-319-21936-3. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21936-3 (cit. on p. 1).

[4] M. Rejewski, "An application of the theory of permutations in breaking the enigma cipher," *Applicationes mathematicae*, vol. 16, no. 4, pp. 543–559, 1980. DOI: 10.4064/am-16-4-543-559 (cit. on p. 1).

[5] B. Randell, "The colossus," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G. Rota, Eds., Academic Press, 1980, pp. 47–92, ISBN: 9780124916500. DOI: 10.1016/c2009-0-22029-0 (cit. on p. 1).

[6] O. Goldreich, *Foundations of Cryptography: Volume 1*. USA: Cambridge University Press, 2006, ISBN: 0521035368 (cit. on p. 1).

[7] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd. Chapman & Hall/CRC, 2014, ISBN: 1466570261 (cit. on p. 1).

[8] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Advances in Cryptology – EUROCRYPT 2005*, R. Cramer, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 19–35, ISBN: 978-3-540-32055-5 (cit. on p. 1).

[9] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, Springer Berlin Heidelberg, 1996, pp. 104–113, ISBN: 9783540686972 (cit. on p. 2).

[10] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, Springer Berlin Heidelberg, 1999, pp. 388–397, ISBN: 9783540484059. DOI: 10.1007/3-540-48405-1_25 (cit. on p. 2).

[11] M. Backes *et al.*, "Acoustic Side-Channel attacks on printers," in *19th USENIX Security Symposium (USENIX Security 10)*, ser. USENIX Security'10, Washington, DC: USENIX Association, Aug. 2010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/acoustic-side-channel-attacks-printers (cit. on p. 2).

[12] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. 2014, pp. 444–461, ISBN: 9783662443712 (cit. on p. 2).

[13] B. Nassi *et al.*, "Video-based cryptanalysis: Extracting cryptographic keys from video footage of a device's power led captured by standard video cameras," in *2024 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 166–166. DOI: 10.1109/SP54263.2024.00163. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00163 (cit. on p. 2).

[14] W. Van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?" *Computers & Security*, vol. 4, no. 4, pp. 269–286, Dec. 1985. DOI: 10.1016/0167-4048(85)90046-x (cit. on p. 2).

[15] H. Maghrebi, T. Portigliatti, and E. Prouff, "Breaking cryptographic implementations using deep learning techniques," in *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6*. 2016, pp. 3–26, ISBN: 9783319494456. DOI: 10.1007/978-3-319-49445-6_1 (cit. on pp. 2, 15).

[16] F. Schijlen, L. Wu, and L. Mariot, "Nascty: Neuroevolution to attack side-channel leakages yielding convolutional neural networks," *Mathematics*, vol. 11, no. 12, p. 2616, 2023, ISSN: 2227-7390. DOI: 10.3390/math11122616 (cit. on pp. 2, 16, 17, 19, 20, 27, 28, 33, 38, 46, 47, 89, 90).

[17] W. Diffie and M. E. Hellman, "Multiuser cryptographic techniques," in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, ser. AFIPS '76, New York, New York: Association for Computing Machinery, 1976, pp. 109–112, ISBN: 9781450379175. DOI: 10.1145/1499799.1499815 (cit. on p. 5).

[18] N. I. of Standards *et al.*, "Specification for the advanced encryption standard (aes)," Nov. 2001. DOI: 10.6028/NIST.FIPS.197 (cit. on p. 6).

[19] J. Daemen and V. Rijmen, "The block cipher rijndael," in *International Conference on Smart Card Research and Advanced Applications*, Springer, 1998, pp. 277–284 (cit. on p. 6).

[20] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999 (cit. on p. 6).

[21] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – the Advanced Encryption Standard*. Springer Berlin Heidelberg, 2002, ISBN: 978-3-540-42580-9. DOI: 10.1007/978-3-662-04722-4 (cit. on p. 6).

[22] P. Emmanuel, S. Remi, B. Ryad, C. Eleonora, and D. Cecile, "Study of deep learning techniques for side-channel analysis and introduction to ascad database," *IACR Cryptol. ePrint Arch.*, pp. 1–45, 2018. [Online]. Available: http://eprint.iacr.org/2018/053 (cit. on pp. 7, 15, 16, 24, 89, 90).

[23] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman," *Cryptography*, vol. 4, no. 2, 2020, ISSN: 2410-387X. DOI: 10.3390/cryptography4020015. [Online]. Available: https://www.mdpi.com/2410-387X/4/2/15 (cit. on p. 7).

[24] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31, ISBN: 978-0-387-38162-6. DOI: 10.1007/978-0-387-38162-6 (cit. on p. 8).

[25] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ascad database," *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020. DOI: 10.1007/s13389-019-00220-8 (cit. on pp. 9, 16, 89, 90).

[26] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*, Springer, 2009, pp. 443–461, ISBN: 9783642010019. DOI: 10.1007/978-3-642-01001-9_26 (cit. on p. 9).

[27] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, pp. 386–408, 1958, ISSN: 0033-295X. DOI: 10.1037/h0042519 (cit. on p. 10).

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org (cit. on p. 10).

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on p. 10).

[30] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning*. Springer, 2006, vol. 4, ISBN: 9780387310732 (cit. on p. 10).

[31] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. [Online]. Available: probml.ai (cit. on p. 10).

[32] S. Picek, I. P. Samiotis, J. Kim, A. Heuser, S. Bhasin, and A. Legay, "On the performance of convolutional neural networks for side-channel analysis," in *Security, Privacy, and Applied Cryptography Engineering: 8th International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings 8*, Springer International Publishing, 2018, pp. 157–176, ISBN: 978-3-030-05072-6. DOI: 10.1007/978-3-030-05072-6_10 (cit. on p. 11).

[33] M. Shaikh, Q. A. Arain, and S. Saddar, "Paradigm shift of machine learning to deep learning in side channel attacks-a survey," in *2021 6th International Multi-Topic ICT Conference (IMTIC)*, IEEE, 2021, pp. 1–6. DOI: 10.1109/IMTIC53841.2021.9719689 (cit. on p. 11).

[34] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980. DOI: 10.1007/bf00344251 (cit. on p. 12).

[35] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, Jan. 1962. DOI: 10.1113/jphysiol.1962.sp006837 (cit. on p. 12).

[36] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," *Advances in neural information processing systems*, vol. 2, pp. 396–404, 1989 (cit. on p. 12).

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Dec. 1998. DOI: 10.1109/5.726791 (cit. on p. 12).

[38] E. Cagli, C. Dumas, and E. Prouff, "Convolutional neural networks with data augmentation against jitter-based countermeasures: Profiling attacks without preprocessing," in *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, Springer, 2017, pp. 45–68, ISBN: 978-3-319-66787-4 (cit. on p. 15).

[39] H. Maghrebi, "Deep learning based side channel attacks in practice," *Cryptology ePrint Archive*, 2019. [Online]. Available: https://eprint.iacr.org/2019/578 (cit. on p. 15).

[40] L. Masure, C. Dumas, and E. Prouff, "A comprehensive study of deep learning for side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 348–375, 2020. DOI: 10.13154/tches.v2020.i1.348-375 (cit. on p. 15).

[41] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, "Methodology for efficient cnn architectures in profiling attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 1–36, Nov. 2019. DOI: 10.13154/tches.v2020.i1.1-36 (cit. on pp. 15, 16, 89, 90).

[42] L. Wouters, V. Arribas, B. Gierlichs, and B. Preneel, "Revisiting a methodology for efficient cnn architectures in profiling attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 147–168, 2020. DOI: 10.13154/TCHES.V2020.I3.147-168 (cit. on pp. 15, 16, 28, 38, 46, 47, 89, 90).

[43] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, "Sok: Deep learning-based physical side-channel analysis," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–35, 2023. DOI: 10.1145/3569577 (cit. on p. 15).

[44] L. Wu, "The circle of dl-sca: Improving deep learning-based side-channel analysis," Ph.D. dissertation, Delft University of Technology, Netherlands, 2023. DOI: 10.4233/UUID:66F0C152-65A0-45BC-B542-BA9799D6A0C1 (cit. on p. 15).

[45] L. Wu, G. Perin, and S. Picek, "I choose you: Automated hyperparameter tuning for deep learning-based side-channel analysis," *IEEE Transactions on Emerging Topics in Computing*, vol. 12, no. 2, pp. 546–557, 2024. DOI: 10.1109/TETC.2022.3218372 (cit. on pp. 16, 89, 90).

[46] J. Rijsdijk, L. Wu, G. Perin, and S. Picek, "Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, pp. 677–707, 2021. DOI: 10.46586/TCHES.V2021.I3.677-707 (cit. on pp. 16, 89, 90).

[47] R. Y. Acharya, F. Ganji, and D. Forte, "Infoneat: Information theory-based neuroevolution of augmenting topologies for side-channel analysis," 2021. DOI: 10.48550/arXiv.2105.00117. arXiv: 2105.00117 [cs.CR] (cit. on pp. 16, 17, 89, 90).

[48] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016. arXiv: 1611.02167 (cit. on p. 16).

[49] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002. DOI: 10.1162/106365602320169811 (cit. on p. 17).

[50] Z. Michalewicz, *Genetic algorithms + data structures = evolution programs*. Springer Science & Business Media, 1996, ISBN: 978-3-540-60676-5. DOI: 10.1007/978-3-662-03315-9 (cit. on p. 20).

[51] L. Booker, "Improving search in genetic algorithms," *Genetic algorithms and simulated annealing*, 1987 (cit. on p. 20).

[52] L. Masure and R. Strullu, "Side-channel analysis against anssi's protected aes implementation on arm: End-to-end attacks with multi-task learning," *Journal of Cryptographic Engineering*, vol. 13, pp. 1–19, Mar. 2023. DOI: 10.1007/s13389-023-00311-7 (cit. on p. 24).

[53] S. Picek, A. Heuser, A. Jovic, S. Bhasin, and F. Regazzoni, "Tipping the balance: Imbalanced classes in deep-learning side-channel analysis," *IEEE Design Test*, vol. 41, no. 2, pp. 32–38, 2024. DOI: 10.1109/MDAT.2023.3288808 (cit. on p. 24).

[54] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. DOI: 10.1137/141000671 (cit. on p. 25).

[55] M. Innes *et al.*, "Fashionable modelling with flux," *CoRR*, vol. abs/1811.01457, 2018. arXiv: 1811.01457. [Online]. Available: https://arxiv.org/abs/1811.01457 (cit. on p. 25).

[56] M. Innes, "Flux: Elegant machine learning with julia," *Journal of Open Source Software*, 2018. DOI: 10.21105/joss.00602 (cit. on p. 25).

[57] A. Basak, "A rank based adaptive mutation in genetic algorithm," *International Journal of Computer Applications*, vol. 175, no. 10, pp. 49–55, Aug. 2020, ISSN: 0975-8887. DOI: 10.5120/ijca2020920572 (cit. on p. 32).

[58] S.-H. Jung, "Rank-based control of mutation probability for genetic algorithms," *International Journal of Fuzzy Logic and Intelligent Systems*, vol. 10, no. 2, pp. 146–151, 2010. DOI: 10.5391/IJFIS.2010.10.2.146 (cit. on p. 32).

[59] M. Sewell, J. Samarabandu, R. Rodrigo, and K. McIsaac, "The rank-scaled mutation rate for genetic algorithms," *International Journal of Information Technology*, vol. 3, no. 1, pp. 32–36, 2006 (cit. on p. 32).

[60] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952. DOI: 10.1080/01621459.1952.10483441 (cit. on pp. 36, 53).

[61] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. DOI: 10.1214/aoms/1177730491 (cit. on pp. 36, 53).

Appendix A

NAS performance comparison

Table A.2 and Table A.3 show the recorded performance and complexity of the resulting neural architecture design found. The traces to obtain mean key rank 0 (T_{GE0}) represent the performance and the number of parameters gives an indication of the complexity. If a particular design failed to obtain a mean key rank of 0, it is represented with a ∞. Compared to Table 2.1 they may miss some records not found in the papers they originate from. The NASCTY [16] algorithm in its paper misses the performance for traces with a desynchronisation level of 100. Instead, the findings found in establishing a baseline level of performance in Chapter 8 have been used. The found baseline results are marked with a †, and the modified results have received the moniker NASCTY2. Additionally, these tables and the updated table for non-desynchronised traces Table A.1 have gained the results (as seen in Chapter 8) of the modified NASCTY algorithm developed in this thesis.

Table A.1: Comparison of performance and parameter size of different hyperparameter tuning methods on the synchronized fixed key ASCAD [25] dataset.

Model	Type	Traces to obtain mean key rank 0	Parameters
ASCAD[22]	MLP	410	393 936
ASCAD[22]	CNN	480	66 652 444
AutoSCA[45]	MLP	129	478 656
AutoSCA[45]	CNN	158	54 752
Zaid *et al.*[41]	CNN	191	16 960
Wouters *et al.*[42]	CNN	$\approx$ 200 [46]	6 436
MetaQNN[46]	CNN	202	79 439
MetaQNN[46]	CNN	242	1 282
InfoNEAT[47]	InfoNEAT	130	15 107
NASCTY[16]	CNN	314	10 470
NASCTY†	MLP	150	7 953
NASCTY2	CNN	88	6 716

Table A.2: Comparison of performance and parameter size of different hyperparameter tuning methods on the desynchronized fixed key ASCAD [25] dataset with the desynchronized level of 50.

Model	Type	Traces to obtain mean key rank 0	Parameters
ASCAD[22]	MLP	-	-
ASCAD[22]	CNN	-	-
AutoSCA[45]	MLP	-	-
AutoSCA[45]	CNN	-	-
Zaid *et al.*[41]	CNN	244	87 279
Wouters *et al.*[42]	CNN	$\approx$ 250 [46]	41 052
MetaQNN[46]	CNN	313	2 100
MetaQNN[46]	CNN	443	41 052
InfoNEAT[47]	InfoNEAT	-	-
NASCTY[16]	CNN	531	68 427
NASCTY†	CNN	300	214 850
NASCTY2	MLP	∞	529

Table A.3: Comparison of performance and parameter size of different hyperparameter tuning methods on the desynchronized fixed key ASCAD [25] dataset with the desynchronized level of 100.

Model	Type	Traces to obtain mean key rank 0	Parameters
ASCAD[22]	MLP	-	-
ASCAD[22]	CNN	-	-
AutoSCA[45]	MLP	-	-
AutoSCA[45]	CNN	-	-
Zaid *et al.*[41]	CNN	270	142 044
Wouters *et al.*[42]	CNN	-	42 652
MetaQNN[46]	CNN	-	-
MetaQNN[46]	CNN	-	-
InfoNEAT[47]	InfoNEAT	-	-
NASCTY†	CNN	4 962	32 937
NASCTY2	CNN	∞	1 338

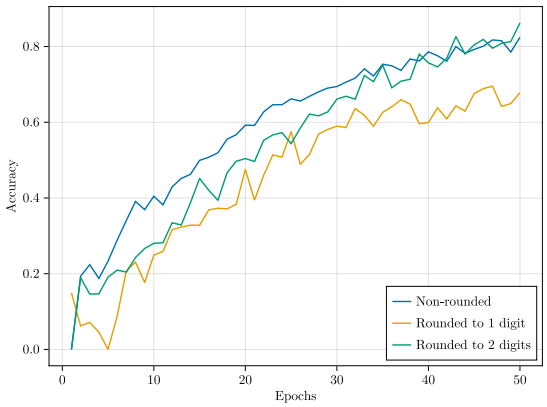
Table A.4: Comparison of performance and parameter size of different hyperparameter tuning methods on the synchronized variable key ASCAD [25] dataset.

Model	Type	Traces to obtain mean key rank 0	Parameters
AutoSCA[45]	CNN	1 568 [46]	2 076 744 [46]
AutoSCA[45]	CNN	496 [46]	1 314 009 [46]
MetaQNN[46]	CNN	490	70 492
InfoNEAT[47]	InfoNEAT	120	317 408
NASCTY†	CNN	∞	128 529
NASCTY2	MLP	∞	783

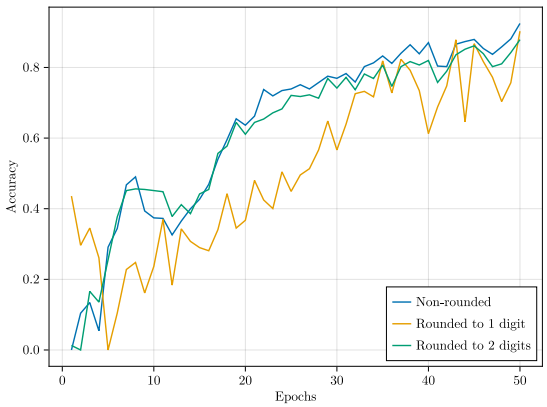
Appendix B

Accuracy

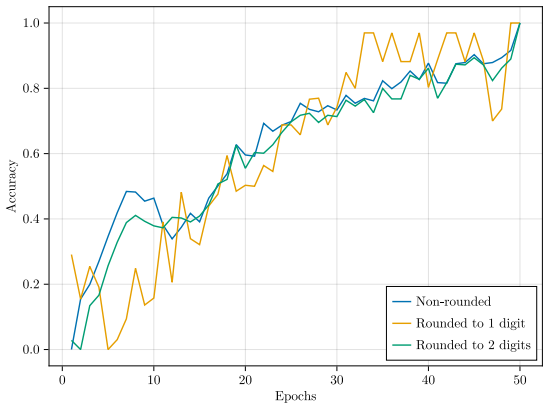
Chapter 5 contains Figure 5.3 describing the selection accuracy in a three contestant tournament. As a complementary overview of accuracy Figure B.1 shows the selection accuracy at each epoch for only two contestants in a tournament. Allowing for a more straight-forward overview of the accuracy.



(a) All genomes.



(b) Improvement ≥ 0.00 (53).



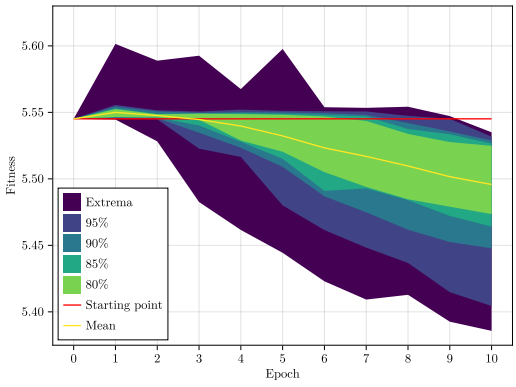
(c) Improvement ≥ 0.01 (65).

Figure B.1: The accuracy of selection function consisting of only two contestants compared with different training periods for the phenotypes –measured in epochs– in the evaluation stage of the genomes.

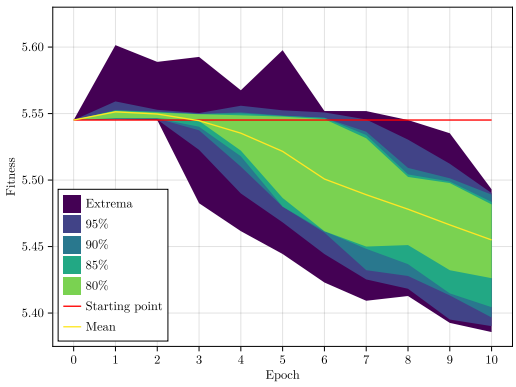
Appendix C

Early termination

Figure C.1 is a complementary figure to Figure 5.5 showcasing the rough distribution of fitness values over time meeting a certain threshold.



(a) ≥ 0.01.



(b) ≥ 0.05.

Figure C.1: The validation fitness of 2009 random genomes over 10 epochs, showing the extremes and the population dispersion for all genomes ≥ 0.01 (A) and ≥ 0.05 (B).

Appendix D

Population size

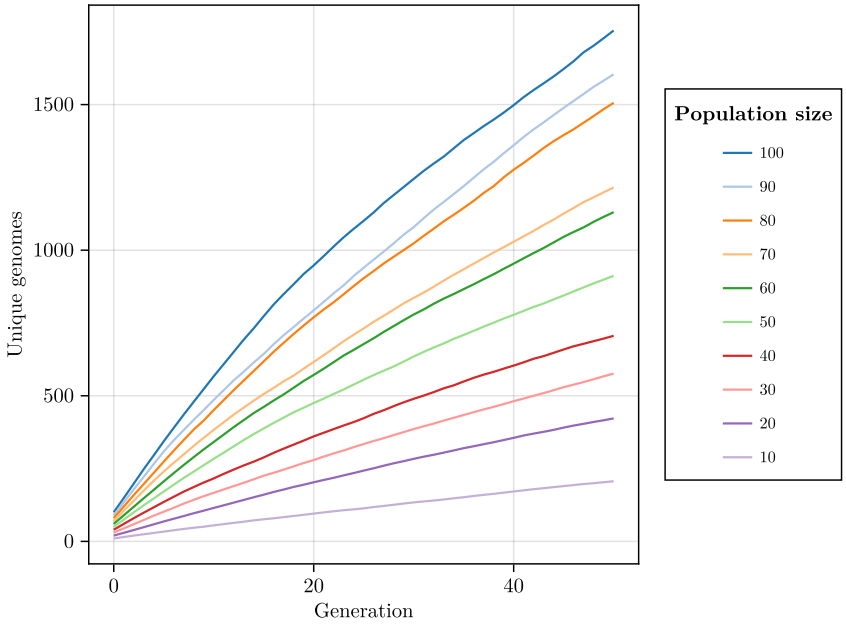


Figure D.1: The total number of unique genomes evaluated per generation.

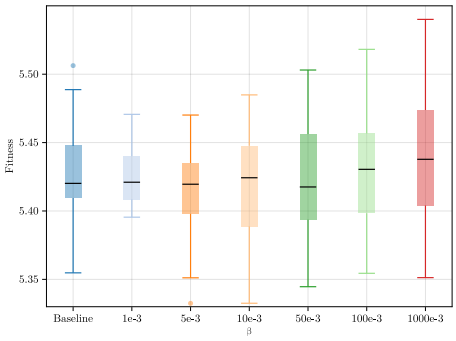
Appendix E

Composite custom fitness function

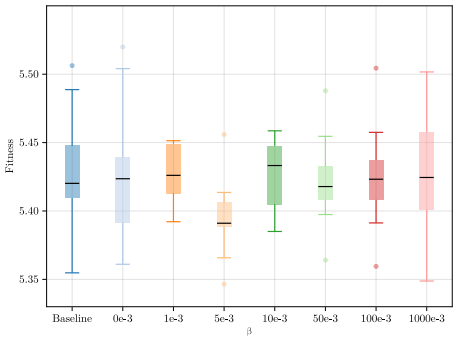
This appendix contains the figures created from the metrics of custom fitness function III. Because of the large number of figures and metrics collected the less important figures have been moved here.

The main goal of custom fitness function III is to select for the better α and β parameter combinations for combatting the complexity of the genetic algorithms output. The findings of custom fitness function I and II show that only the α values have significant impact on the end results fitness score. Therefore, Figure E.1 with its convergence metrics show similar movements across the sub-figures.

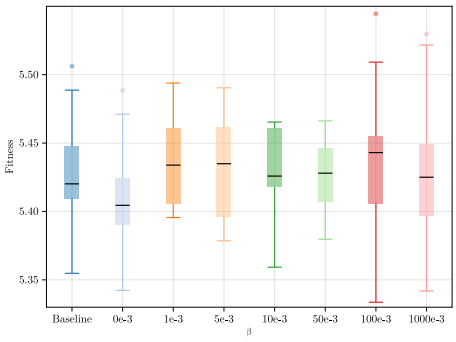
Figure E.3 and Figure E.2 are gathered from the complete population after ten generations show similar results to the metrics collected for the best results.



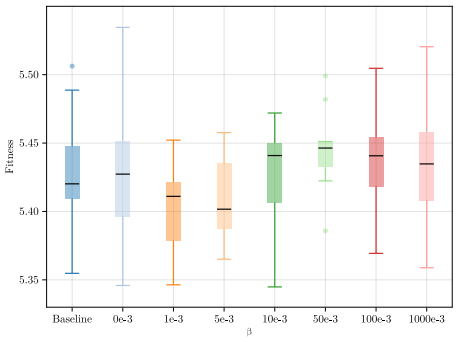
(a) α = 0e-7.



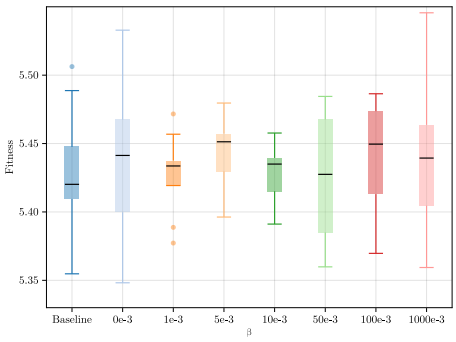
(b) α = 1e-7.



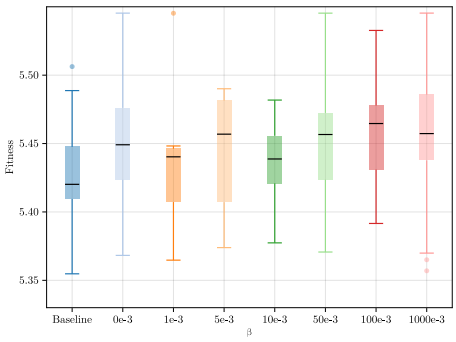
(c) α = 5e-7.



(d) α = 10e-7.

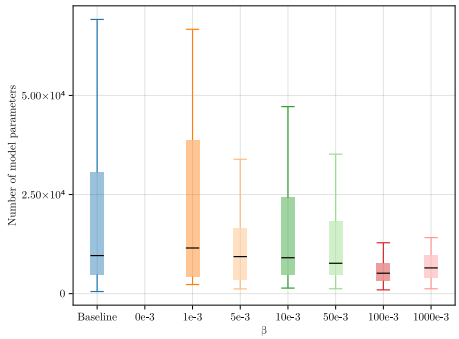


(e) α = 50e-7.

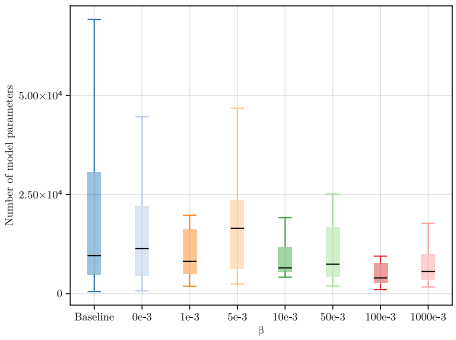


(f) α = 100e-7.

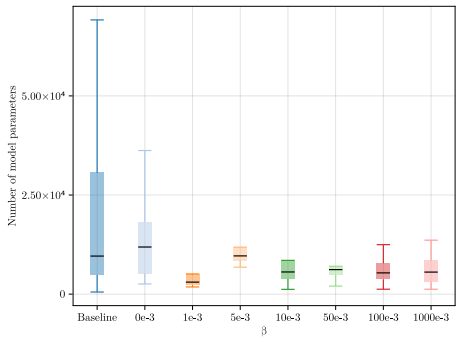
Figure E.1: The fitness score of the genetic algorithm after 10 generations with the use of custom fitness function III.



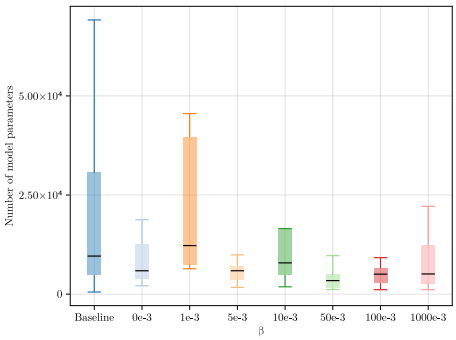
(a) α = 0e-7.



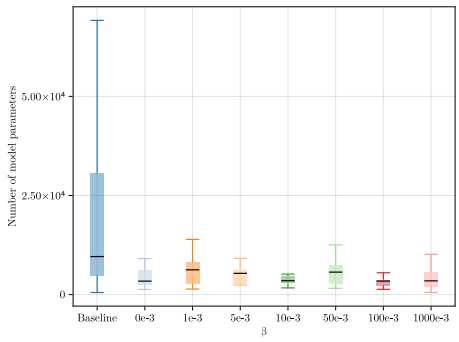
(b) α = 1e-7.



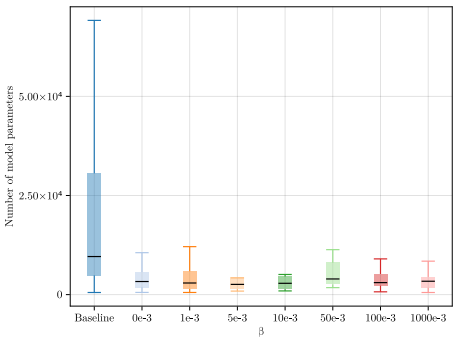
(c) α = 5e-7.



(d) α = 10e-7.

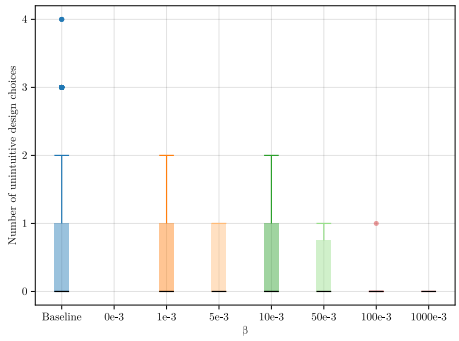


(e) α = 50e-7.

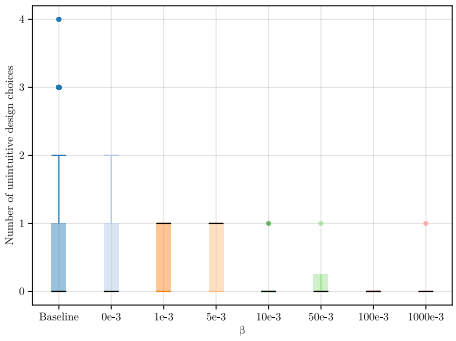


(f) α = 100e-7.

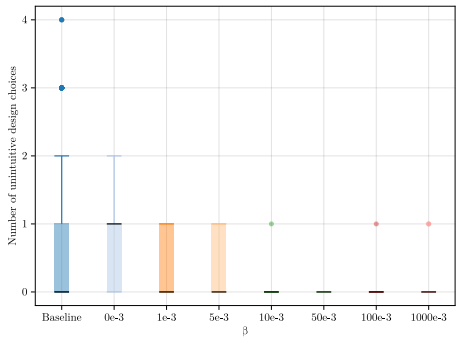
Figure E.2: The mean number of model parameters in the population of the genetic algorithm after 10 generations with the use of custom fitness function III.



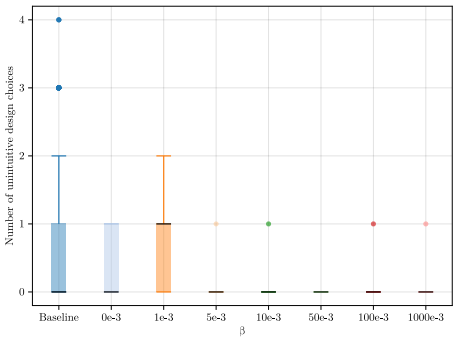
(a) α = 0e-7.



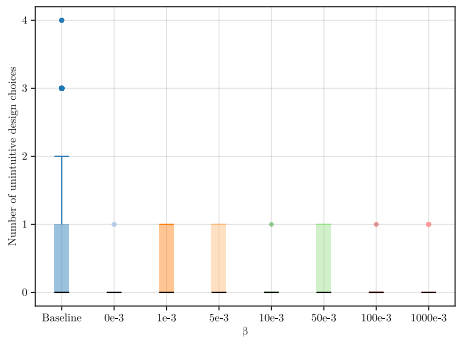
(b) α = 1e-7.



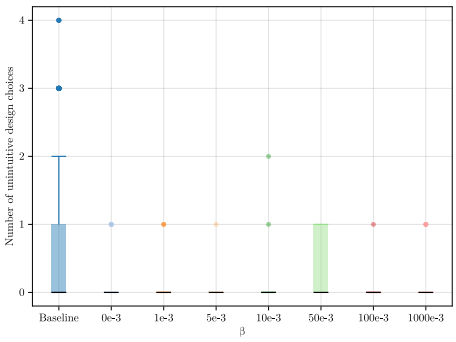
(c) α = 5e-7.



(d) α = 10e-7.



(e) α = 50e-7.



(f) α = 100e-7.

Figure E.3: The mean number of unintuitive design choices in the population of the genetic algorithm after 10 generations with the use of custom fitness function III.