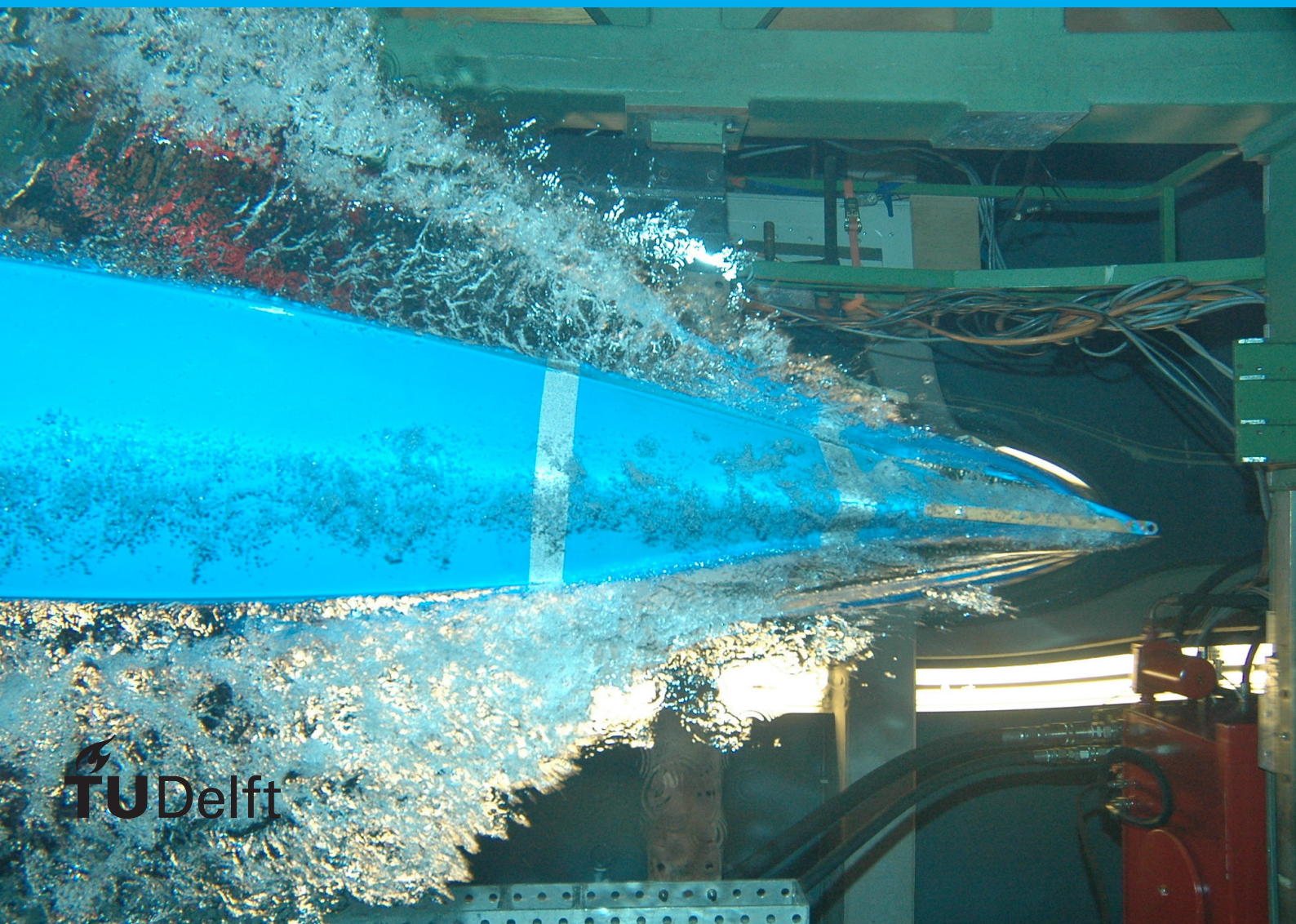
Methodologies for deep learning SCA

An analysis on the design and construction of convolutional neural networks for side-channel datasets

Philip Blankendal



Methodologies for deep learning SCA

An analysis on the design and construction of convolutional neural networks for side-channel datasets

by

Philip Blankendal

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 26, 2022 at 09:00 AM.

Student number:	1547682	
Supervisor:	Dr. ir. S. Picek,	TU Delft
Thesis committee:	Prof. dr. ir. Inald Lagendijk,	TU Delft
	Dr. ir. S. Picek,	TU Delft
	Dr. E. Isulfi,	TU Delft

This thesis is confidential and cannot be made public until August 26, 2022.

TU Delft

Abstract

Side-channel attacks leverage the unintentional leakage of information that indirectly relates to cryptographic secrets such as encryption keys. Previous settings would involve an attacker conducting some manual-statistical analysis to exploit this data and retrieve sensitive information from the target. With the adoption of deep learning techniques, side-channel attacks have become more powerful and require less manual analysis; hence, approaches involving deep learning have become the de facto standard for side-channel analysis. Especially the convolutional neural network has been highly effective in bypassing side-channel-specific countermeasures. The research surrounding the application of deep learning in the side-channel domain has so far primarily focused on either introducing architectures that perform well on specific datasets, data-preprocessing techniques, or the assessment of model output. Only a few attempts have been made toward the methodology aspect involving the generation of convolutional neural network architectures. The negligence of this part lends itself to the challenge of interpreting the model's decision-making process and the high number of tunable parameters and design choices. In this work, we assess the architectural components and the hyper-parameters encountered when constructing a CNN and attempt to determine the steps of a conceivable methodology for building well-performing CNNs in the side-channel domain regardless of the dataset used. We assess the most influential hyperparameter values and architectural design choices for CNNs and decide on suitable ranges and architectural constraints. We also evaluate how attribution methods can help us achieve better results when tuning hyperparameters for SCA. Combining these acuities, we summarize the general guidelines obtained and present them as a methodology for constructing CNNs for the SCA domain. We evaluate our proposed methodology by applying it to unseen datasets and show that we can obtain state-of-the-art results compared to other previously proposed SCA methodologies.

Preface

This thesis is the final product of about nine months of effort. Even though I took a short break in between, I truly enjoyed working on this project. Even with the challenges of overcoming multiple obstacles within unprecedented times, it allowed me to become a true expert in deep learning mechanics and appreciate some of the luxuries some of us might take for granted. First, I would like to thank my family and friends for their support throughout my studies, my thesis project, and during tough times; secondly, I would also like to thank Stjepan for his support and guidance throughout the project, as many have said before it truly is a privilege to have such a Supervisor.

Philip Blankendal
Delft, August 2022

Contents

Chapter 1

Introduction

Many companies today depend entirely on computer systems and have almost all sensitive data in a digitized form. Moreover, this data must be accessible across multiple interconnected embedded systems across the globe. In other areas, such as hospitals, embedded systems can be found in various medical equipment, ranging from wheelchairs to hospital beds.

Each of these electrical devices either stores or has access to sensitive data such as patient information, financial data (credit card info, bank statements), company information, intellectual property, key codes, and many other forms of potentially harmful material. If such devices were to be compromised, the consequences could be catastrophic. Data breaches could risk a patient's health, cause company bankruptcies, and disrupt entire supply chains.

Indeed the digitalization era has had many benefits for both automation and communication; however, there has also been a significant drawback as it has become easier for an attacker to retrieve sensitive data without having physical access to generative devices. Because of this, there has also been a tremendous increase in the number of cyber-security attacks aimed at the retrieval of sensitive information.

Due to these attacks, security researchers have been working around the clock on developing new security measures that can be embodied into electrical devices and embedded systems to prevent this data from falling into the wrong hands. So far, this task has proven to be non-trivial as present cyber-attacks seem highly adaptive and able to circumvent many of these newly implemented measures within a relatively short time after deployment. The cybersecurity attacks of today can be segmented into two main categories, namely, passive and active attacks.

Active attacks are cyber security attacks where an attacker actively attempts to alter the inner-workings of a system or actively attempts to exploit a weakness that is present in the system. In contrast, passive attacks do not disrupt the system but rather observe it in its natural state. Passive attacks are brutal to prevent because they do not seek to alter the target system in any way, shape, or form, rendering many of the present cyber security measures useless against these specific types of threats.

Among passive attacks, the side-channel attack [26] is currently amongst the most potent because it is relatively inexpensive and, in many cases, has proven exceptionally challenging to detect. Side-channel attacks exploit information sources that are not directly related to the attacker's target; instead, these sources unintentionally leak exploitable information that can be utilized to obtain the target. The information sources used in these attacks are called side channels. Examples of side-channels that can be used in SCA attacks are power consumption [27], EM waves [40] as well as any other data source that may contain some form of leakage (e.g., cache readings [32] [25] or padding information [1]).

Typically, the attacker's target would be the secret key used by the encryption scheme on a target device, which, once obtained, would allow the attacker to decrypt any sensitive messages or data that passes through the device that could be of interest. There are two types of side-channel analysis: profiled side-channel analysis and non-profiled side-channel analysis. In profiled side-channel analysis, the attacker has complete access to a device similar to the target instrument. This copy device allows the attacker to gather as much side-channel information (e.g., traces) as needed and plan his attack accordingly: by using this copy device first before actually attempting an attack on the target device. In

the non-profiled scenario, the attacker can only gather a limited amount of such information because he does not have access to such a copy device.

The profiled side-channel analysis consists of two phases: the profiling stage, where information is gathered from a clone device and analyzed in order to derive a model (or plan) that can be used to attack the target device, and the attack phase, where the model constructed during the profiling stage is used to retrieve the target information from the target device. In the past, side-channel attacks often consisted of conducting a manual statistical analysis of the information contained within these sources. However, with the advent of deep learning, side-channel attacks have become even more powerful, making the need for manual statistical analysis obsolete. From an information-theoretical point of view, the template attack [7] is among the most potent; however, this approach is susceptible to side-channel-specific countermeasures. However, contrary to the template attack, research has shown that deep learning architectures can easily break any encryption quickly, even when side-channel specific countermeasures are present [3, 59].

So far, the research surrounding the use of deep learning architectures for side-channel analysis has primarily focused on presenting architectures that perform well on specific datasets [59] [3]. However, very little research is available on the steps required to create such a well-performing deep learning model that can be applied to the SCA domain regardless of the data set used. In [59], numerous state-of-the-art CNN architectures were presented that were effective for different data sets, whereas in [41], the first actual methodology was introduced that produced high-performing models for SCA. In this thesis, we will primarily focus on the methodology aspect of deep learning in SCA. We analyze the building blocks, and standard practices [4] from other domains to produce steps that can be used to construct high-performing models in the SCA domain. Furthermore, we also analyze various attribution methods that have been useful in other domains and evaluate their current application in the SCA domain. We aim to answer the following research questions:

Research question 1

- Can we identify structural constraints and hyper-parameter ranges that produce high-performing CNNs and aid in the design and construction of CNNs in the field of side-channel analysis?

Research question 2

- Can attribution methods from other domains be used to aid in the design of deep learning models for SCA?

Research question 3

- Can we derive a new methodology for the construction of CNNs that gives good results and is easy to follow?

The structure of this thesis is as follows. In chapter 2, we provide some background necessary for our research. In chapter 3, we summarize previous work concerning the usage of deep learning in SCA. In chapters 4, 5, and 6, we focus on our research question, and in chapter 7, we summarize our contributions and provide some possible areas that can be expanded upon in future work.

Chapter 2

Background

2.1 Machine learning

Machine learning refers to the process of automatically deriving a distribution function from a given set of data points, with the purpose of inferring values of interest from new, unseen data points using this newly derived function. The algorithms that automatically derive such a distribution function are referred to as machine-learning algorithms. The process of creating such a distribution function is called training (often referred to as learning). Machine learning algorithms are usually divided into two main categories: 1)supervised-learning algorithms and 2)unsupervised-learning algorithms. *Supervised learning* is the field in which all the input values used during training have a set of appropriate corresponding output values (often referred to as labels or target values), whereas the field of unsupervised learning only has the input values without any corresponding targets. Next to supervised and unsupervised learning, there is also an intersection of the two called semi-supervised learning; however, In this thesis, we only consider supervised learning.

As stated before, supervised algorithms require each input value to have an associated label; these labels are often also called target variables and are usually denoted as "y". These labels represent the correct value that a learned function should infer when it is given the input belonging to that specific label. During training, these labels (or target values) are compared to the corresponding inferred values to create an effective model where the difference between these two values is minimal. Next to the training phase, there is also a testing phase for supervised-learning algorithms. During the testing phase, these labels are only used to determine the performance of the learned model by computing performance-specific metrics such as accuracy.

2.2 Deep learning

Deep learning is a sub-field of machine learning in which architectural structures (called layers) are used to construct the distribution function. In the context of deep learning, a layer is an abstract representation of a function (or part of a function). A layer has both an input and an output value. The term "architecture" is used to denote the amount and types of layers used and how they are combined (a particular combination of specific layers connected in a specific matter) to create the final overall distribution function. The resulting function is often referred to as a "model" as it tries to approximate the underlying distribution function by using a set of sample points (often called training samples) drawn from this underlying function.

The layers used in deep learning algorithms are typically placed in a sequence where one could have different types of layers connected to create a more significant function. Such a function is often called a "Sequential" Model, consisting of smaller sub-functions (layers) placed in sequence. In a sequential model, the output of one sub-function (layer) serves as input for the next until the final layer is reached, at which point the output of the final layer is served as the model's output. An example of a deep learning model is given below in figure 2.1.

$$f(x) = f^3(f^2(f^1(x)))$$

Figure 2.1: A distribution function f (represented as a deep learning architecture), consisting of the 3 sub-functions f^1, f^2 and f^3 (called layers), where x is the input of the model f.

The first layer of a deep learning architecture is called the input layer, whereas the last layer of the model is called the output layer. The layers in between the input and output layers are usually called ”hidden” layers. An example of a deep learning architecture is given in figure 2.1 where $f1$ is the input layer, $f2$ is a hidden layer and $f3$ is the output layer. It has been common to use graphical diagrams to depict the different types of layers used in deep learning architectures (and their connectivity). An example of a graphical representation of deep learning architecture with one input layer, one hidden layer, and one output layer is shown in figure 2.2.

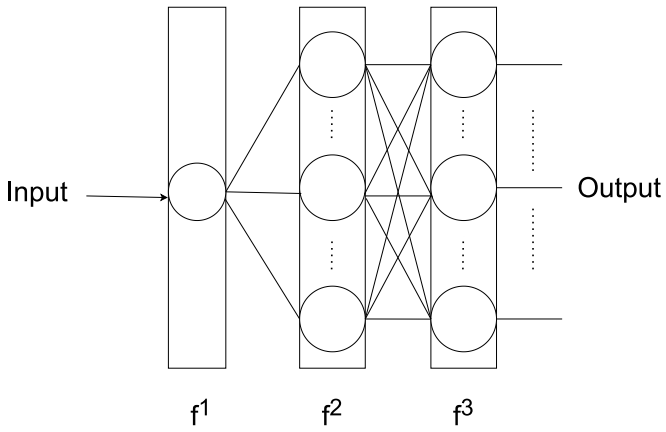


Figure 2.2: An example of a graphical representation of a neural network with three layers.

A deep learning architecture can have any amount of layers, provided it has at least one input layer. This layer is considered both the input and the output layer (with no hidden layers) for architectures containing a single layer.

The term ”deep” refers to the number of layers used in a network: the more layers, the deeper the network. Deep learning networks are often called neural networks, where the term ”neural” refers to one of the most elementary functions that can be used to create a layer called the neuron function. Neurons are often the most rudimentary building blocks for constructing a layer and are often referred to as the ”perceptrons”. The term ”network” is commonly used due to the interconnectivity between neurons and the layers. If all the layers in a deep architecture only have forward connections, it is referred to as a feed-forward network. In this thesis, we only consider feed-forward networks.

2.2.1 Neuron

As stated before, the building block used to create a layer in a deep learning network is called a neuron (or perceptron). A neuron is nothing more than a function that has several inputs and one output.

The function used for any neuron in this thesis is of the form:

$$g(W^T * x + b) \tag{2.1}$$

Where W is a weight vector, x is the input vector, b is called the bias, and $g()$ is a fixed predefined function often referred to as the ”activation” function. The weight vector W is a vector of scalar values, and its size is equal to the size of the input vector x. The weight vector W is usually initialized to contain random values, where the final values are then gradually learned during the training phase of the deep learning algorithm. The bias b is a single scalar value updated in a similar matter. Because these last two parameters are not set until the end of the training phase, they are called ”trainable” parameters. The input vector x is given to the neuron as input. The ”activation” function g can have many forms and is chosen by the creator of the deep learning architecture. Even though an activation function can be

any predefined function, $g(x)$ is almost always, except for the last layer in most classification models, set to be a function that contains some form of non-linearity.

Among a large amount of commonly used non-linear activation functions, one of the most commonly used non-linear activation functions in deep learning architectures (and often the default choice for hidden layers) is called the rectified linear unit (ReLU) shown below in equation 2.2:

$$g(x) = \begin{cases} x, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

Linear activation functions can also be used in deep learning architectures; however, they are primarily used in the final layer of deep learning architectures. An example of a linear activation function is the Softmax function, commonly used in the final layer of a classification model to perform some form of regression, and output class probabilities, which has become common practice if the goal is to classify the input x.

Because the perceptron is one of the main building blocks in neural networks, it is common to present the neuron function as a graphical node when visually presenting neural architectures, as depicted in figure 2.3:

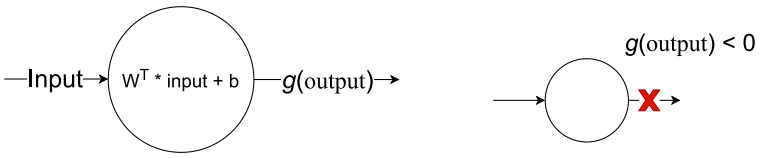


Figure 2.3: Graphical representation of the neuron function of a deep learning network where edges represent input/output connections. When a neuron does not ”fire” (e.g., a neuron has an output of 0), the outgoing edges can often be neglected for that specific input.

There are many more types of activation functions; however, in this thesis, next to the ReLU function, we only consider the following activation functions:

Exponential linear unit (ELU), which was originally introduced in [9] with the purpose of improving ReLU as it allows negative values and hence gives a gradient when the input is negative at the cost of computational expense.

$$g(x) = \begin{cases} a(e^x - 1), & \text{if } x \leq 0. \\ x, & \text{otherwise.} \end{cases} \tag{2.3}$$

Scaled exponential linear unit (SELU), initially introduced in [24] as an improvement upon the ELU activation function, with the potential to significantly boost the training process of the deep learning architecture as it is considered self-normalizing.

$$g(x) = \begin{cases} \lambda\alpha(e^x - 1), & \text{if } x < 0. \\ \lambda x, & \text{otherwise.} \end{cases} \tag{2.4}$$

With parameters $\lambda = 1.0507$ and $\alpha = 1.67326$.

Hyperbolic tangent (Tanh), which is extremely common as its derivative, has a simple form and is hence computationally efficient when using gradient descent.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

2.2.2 The importance of non-linearity

The non-linearity of an activation function makes a neuron (and deep learning architectures in general) so powerful compared to other areas of machine learning. Whereas most machine learning techniques are limited to linear functions, deep learning algorithms do not exhibit this limitation, allowing them to approximate relatively complex relations between input and output values, as opposed to some other machine learning methods.

Non-linear activation functions significantly increase the amount and complexity of distribution functions that a deep neural architecture can model. Because neurons can perform non-linear operations on their input, using multiple such neurons in layers allows deep learning algorithms to learn not only linear functions but also allow deep learning algorithms to learn non-linear functions, such as the XOR function.

2.2.3 Regularizing Layer types

A wide variety of layer functions exists, and it has become standard practice to incorporate different types of layers in deep learning architectures for different purposes. Here we discuss two relevant layer types that reoccur trough-out this thesis and are commonly used in deep learning architectures for regularization purposes. Because we are using one-dimensional data in this thesis, we only consider layers for one-dimensional data.

Pooling layer

A pooling layer is a special kind of layer that reduces the size of its input by using a so-called ”filter”. A filter can be thought of as a sliding window that slides over the input until the end of the input is reached.

When a pooling layer receives an input vector, the filter is gradually moved across the input, where, at each filter position, a so-called ”pooling operation” is performed until the end of the input vector is reached. The filter of a pooling layer is set by using two predefined parameters: filter size and stride. The filter size represents the number of values the pooling operation ”spans” and takes as input before returning any output. The stride of the filter represents how many elements the filter needs to skip or move across the input after it has performed a pooling operation and before it uses the next set of input values to calculate the consequent output value.

The value that a pooling operation returns at each filter position is dependent on the type of pooling layer used. This thesis will consider two types of pooling: Max pooling and Average pooling. When average pooling is used, the operation will return the average of the elements the filter covers for each position. When max pooling is used, the maximum value is returned. An example of max pooling is given figure 2.4.

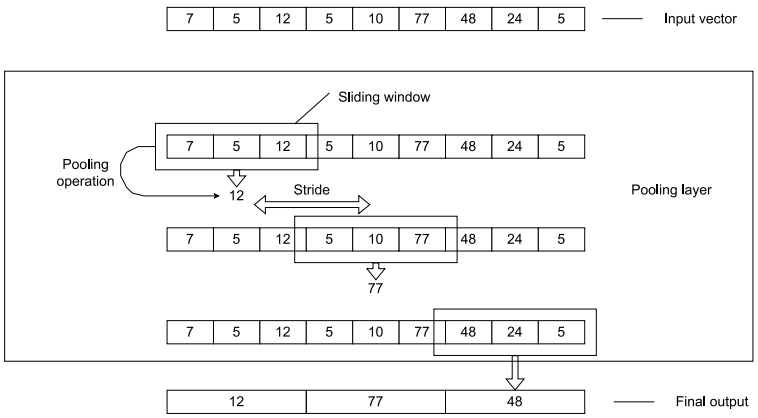


Figure 2.4: example of the operations performed within a max-pooling layer with a filter (sliding window) size of 3 and a stride of 3, with an input vector x of length 9. Here the size of the input vector x is reduced from 9 to 3 by only selecting the maximum value for every three elements (filter size) whilst moving 3 elements (filter stride) to the right after every pooling operation.

Because either only peak information or summarized information is retained in the output, this reduction usually not only results in a reduction in training time but also has the potential to reduce noise (as the output also tends to become invariant to small translations in the input), which, in turn, can increase the overall model performance. The choice of pooling layer and pooling hyper-parameters is often empirically chosen: if the filter size or stride is too large, it is possible that some information loss can occur, which could result in poor performance of the model.

Batch normalization layer

A layer that does batch normalization [19] normalizes the input x to zero mean and unit variance based on the metrics calculated for the specific $mini - batch$ (batch of inputs) that specific input x belongs to.

2.2.4 Multilayer perceptron

A Multilayer perceptron [15] (MLP) is a deep learning architecture that only uses dense layers and contains at least one hidden layer. It is consequently also one of the more simple deep learning architectures because it only allows feed-forward connections from its layers and also because its architecture only contains dense layers. The number of neurons in such a layer type is not limited and varies highly for different architectures. The number of neurons is often empirically chosen by the creator of the architecture. An example of an MLP that was used to solve the XOR function is shown below in figure 2.5. Another example of an MLP is given in figure **??**. So far, the MLPs used in the side-channel domain [3] have typically used multiples of 100 for the number of neurons in their dense layers.

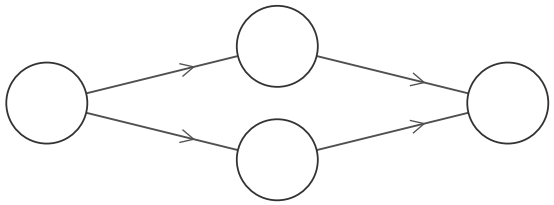


Figure 2.5: A Multilayered perceptron with one input layer, one hidden layer (dense layer) consisting of two neurons, and one output layer. Note that the output layer is also a dense layer (with one single neuron).

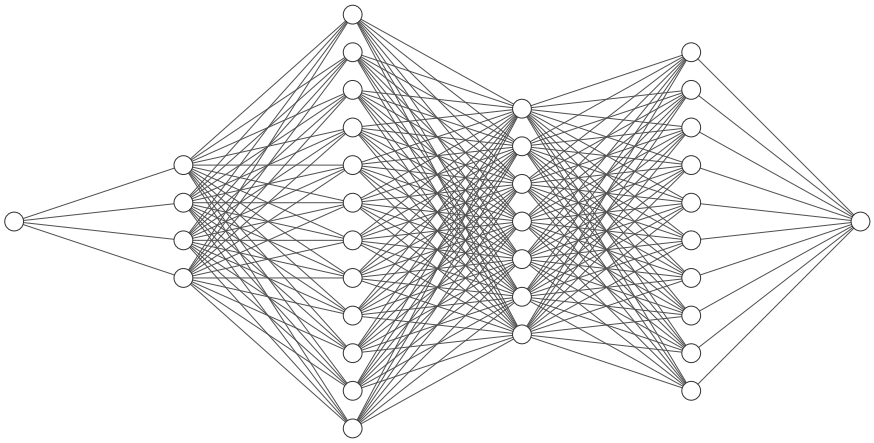


Figure 2.6: A larger mlp used in [].

2.2.5 Convolutional neural networks

Convolutional neural networks are deep learning architectures that use a so-called ”convolutional” layer, which allows for automated feature extraction. The critical component in a convolutional layer is the so-called ”filter,” often referred to as a ”kernel”. The kernel can be thought of as a vector of weights that, similarly to pooling, slides over the input and performs certain mathematical operations (in this case, a mathematical operation called convolution, of which the equation is given in equation 2.6) at specific locations on the input in order to construct the output. The mathematical operator used to perform convolution is often depicted by the $\circledast$ symbol, and the equation for convolution when using one-dimensional input is given in equation 2.6.

$$s(t) = (x \circledast w)(t) = \sum_{a=1}^{n} x(t-a)w(a) \tag{2.6}$$

A convolutional layer has three important hyper-parameters: filter size, amount of filters, and stride. The filter-size parameter is used to define the size of the filter, whereas both filter-size and filter-stride are used to determine the locations on the input were-with the filter convolves with the input in order to construct the output. Figure 2.7 shows an example of this process.

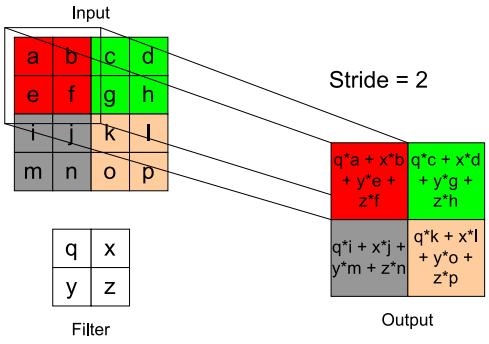


Figure 2.7: An example of convolution with an input size of 4X4 and a filter of size 2X2 with stride 2.

Normal convolution can reduce input dimensions when larger filters (or even smaller filters with a stride larger than 1) are used in a convolutional layer. To prevent this from happening, often, zero-valued vectors are added to the input x borders to increase the input size before passing x to the convolutional layer. Adding zeros to the input before passing it to the convolutional layer is called padding. Padding can also be used to ensure that the dimensions of the output of the convolutional layer will remain the same relative to the input: this is called ”same” padding. An example of ”same” padding is given in figure 2.8.

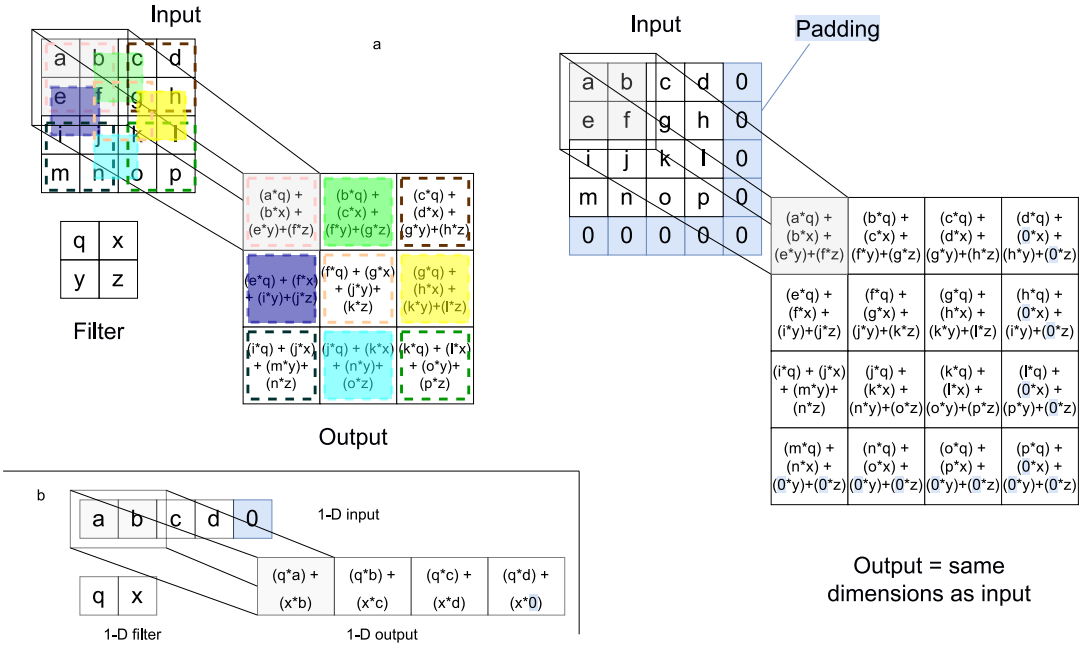


Figure 2.8: a) 2-dimensional example of how padding can be used to ensure that the output dimensions stay equal to that of the input when convolution is applied (with stride 1), b) one-dimensional convolution example for ”same” padding (with stride 1).

The amount-of-filters parameter sets the number of filters to be used in a convolutional layer. For every convolutional filter, a separate output is given. Hence when a convolutional layer with 16 filters is given an input of size $64x64$, the resulting output for this specific layer will be of size $64x64x16$ when using same-padding. If another convolutional layer with 2 filters is placed after this layer, the output of this next layer will be of size $16x16x32$, as each filter produces an output for every output of the

previous filter. The output for a single convolutional filter is called a feature map, and in the example given above, the first layer may produce 16 feature maps (one for each filter) and the second layer 32 feature maps. Figure 2.9 shows an example for this particular scenario.

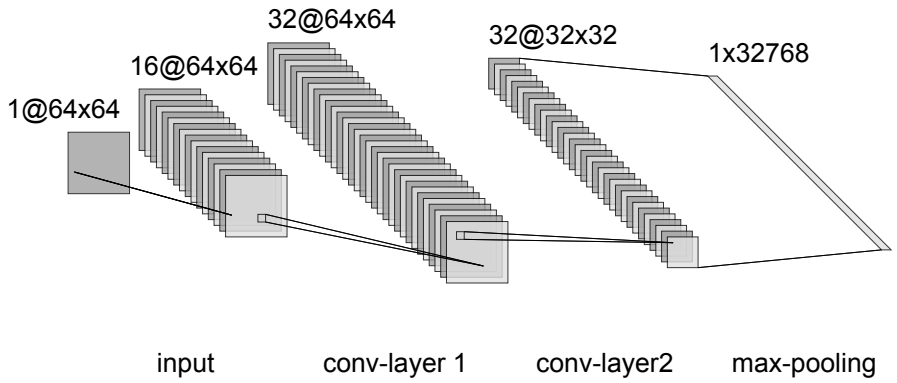


Figure 2.9: An example of convolution network.

Because a convolutional layer with multiple filters usually increases the input size for the next layer (as opposed to the original input size), it is common for CNN architectures to incorporate a pooling layer after a convolutional layer to adjust input dimensions or input range of a layer. When a decrease in input size is desired, one can also choose to use a stride larger than 1 or convolution without padding. However, it is more common that convolutional layers are used along with pooling layers as a pooling layer does not have any trainable parameters and hence can decrease the training time significantly as opposed to many other methods.

It has been standard practice for most architectures to incorporate several fully connected layers after the convolutional layer, as this is often where some form of classification would occur in the case of a classification network. The connection to the fully connected (or classification) part of a particular convolutional network is then often preceded by a so-called ”flatten layer,” which flattens the input by changing the input dimensions $32x32x32$ input matrix into a $1x32768$ input vector before it is passed to the classification part of the CNN.

2.2.6 Evaluation metrics

Accuracy and precision are standard metrics used to evaluate the effectiveness of a deep learning model for most machine learning tasks. However, it was shown in [39] that these metrics are often not indicative enough to assess a model's performance in the field of side-channel analysis. The reason for these shortcomings is that these metrics only tend to take single input traces into account, whereas in side-channel analysis, we are not interested in the model's performance on individual traces but in the encryption key that these traces employ. In order to address these shortcomings of standard ML metrics and correctly assess the performance of a model on a side-channel dataset, two side-channel specific metrics are introduced, namely, the success rate and the guessing entropy[48]. In this thesis, we only consider the guessing entropy.

The Guessing entropy of a deep learning model on a side channel dataset is defined as the average ranking of the correct key among all possibilities according to the inferred values on a fixed set of traces. Hence, when a model's guessing entropy on a specific dataset reaches 1, we know that we have retrieved the correct key (as its ranking is 1). We define the amount of traces needed to reach a guessing entropy of 1 as $N_{T_{GE}}$.

2.3 AES

At the time of writing, the Advanced Encryption Standard (Aes) [13] is one of the most common encryption schemes that are found on hardware chips. The size of its encryption key varies and is often either 128, 192, or 256 bits long. Depending on the encryption-key size, the algorithm for this encryption

scheme uses either 10, 12, or 14 rounds to encrypt plaintext values. At a high level, the AES algorithm can be described as follows:

1. KeyExpansion - Using the AES key schedule, all the round keys are derived from the secret key.
2. AddRoundKey - using the bitwise XOR, the first round key is combined with the plaintext.
3. For 9, 11, or 13 rounds (depending on the secret-key size)
   (a) SubBytes - Using a predefined lookup table called S-box, each byte of the intermediate value is substituted with its lookup value (this is a non-linear operation on the intermediate value).
   (b) ShiftRows - The last three rows of the new intermediate value is shifted according to a predefined cycle.
   (c) MixColumns - The columns of the new intermediate value are mixed in a predefined (and linear) matter.
   (d) AddRoundKey - the new intermediate value is combined with the round-key using the XOR operation.
4. Final round
   (a) SubBytes
   (b) ShiftRows
   (c) MixColumns

AES is currently one of the most secure encryption schemes as, at the time of writing, there is still no known attack on this cipher that has shown to be feasible if we take the computational complexity of the attack [51].

2.4 Counter-measures

There are a significant number of counter-measures specifically aimed at SCA to ensure the security of physical devices. There are two main counter-measures that we will consider in this thesis, which are masking and de-synchronization.

Masking

The masking counter-measures add a second layer of security to the AES encryption schemes by adding a random mask to the sensitive values. The random mask is often divided into multiple parts (called shares), where the name first-order masking refers to when only one share is used and higher order masking when multiple shares are used. When boolean masking is used, the mask is added to the sensitive value through a boolean operation such as XOR, whereas affine masking uses both arithmetic and additive operations. In this thesis, we only focus on boolean masking.

De-synchronization

The de-synchronization counter-measure is categorized as a hiding counter-measure because it actively attempts to hide the leakage of the device and the processing of sensitive values. The De-synchronization counter-measure often works by waiting a random amount of clock cycles before computing sensitive values, where the maximum waiting time is often fixed and pre-determined. De-synchronization has proven to be an effective counter-measure for SCA as datasets that employ this counter-measure have proven to be more difficult to break than those that only use (boolean)masking.

2.5 Side-channel databases

With the growth of deep learning in the field of side-channel analysis, there has also been a significant increase in the amount of publicly available side-channel datasets; however, many of the currently available public databases are trivial. For trivial SCA databases, even a simple MLP is sufficient to retrieve the secret key from such databases [58]), which implies that a CNN is not strictly necessary in such scenarios. For non-trivial databases there are only few methodologies [59] [41] that have obtained relatively competitive results. To conduct our research, we choose a selection of databases that are relatively more difficult to break based on their best-known $N_{T_{GE}}$ and SCA-specific counter-measures.

2.5.1 ASCAD

v1

The ASCAD [3] dataset consists of traces obtained by measuring the electromagnetic radiation of an ATMega8515 micro-controller encrypted with the AES encryption scheme protected with both (boolean-) masking and de-synchronization counter-measures against side-channel attacks. The ASCAD database contains data from the first round of AES only. The dataset contains $50,000$ profiling traces and 10.000 where the key is fixed. The dataset contains three different variants containing different counter-measures. The first dataset, referred to as $N = 0$, only contains the Masking counter-measures. The other two selected versions we consider, referred to as $N = 50$ and $N = 100$, contain both masking and a de-synchronization counter-measure of either 50 or 100.

The target value for this dataset is

$$sbox(p[3] \oplus k[3]) \oplus r_{out} \tag{2.7}$$

In the above equation, r_{out} is the unknown random masking value that was used by the encryption scheme during the encryption process.

2.5.2 AES HD

The AES_{HD} [23] data-set was obtained by measuring the electromagnetic emissions of a Xilinx Virtex-5 FPGA encrypted with AES encryption scheme using an 128 bit encryption key. The data-set consists of $1000,000$ traces of length $1,250$. This dataset sets itself apart from other datasets because the measured traces seem to contain a large amount of noise and has proven difficult to break when using a low amount of traces.

The target variable for this dataset is:

$$sbox^{-1}({C_j}^{(i)} \oplus k^*) \oplus {C_j'}^{(i)} \tag{2.8}$$

Where $C^{(i)}$ is a cypher-text byte associated with the i-th trace. The relationship between j and j' is obtained through the inverse shiftRows step of the encryption scheme algorithm. In this dataset j=12 is used (and hence j'=8) because this byte was easy to attack.

2.5.3 DPA V4

The DPAv4 dataset contains 4500 profiling and 500 attack traces with a total of 4000 features per trace. The traces from this database were obtained by measuring the power consumption of an ATMega-163 smart-card (employing AES-128 encryption scheme).

The target variable for this dataset is:

$$sbox({P_0}^i \oplus k^*) \oplus M \tag{2.9}$$

This dataset is relatively trivial compared to other datasets because of the known masked value; hence, it is easier to break.

2.5.4 AES Random delay

The traces for the AES RD [11] dataset were obtained by measuring the power consumption of an 8-bit Atmel avr microcontroller that makes use of an AES encryption scheme. This dataset contains 25000 profiling and 25000 attack traces, where each trace contains 3500 features. This dataset has the same target variable as the DPA V4 dataset ($sbox[{P_0}^i \oplus k^*] \oplus M$) with the addition of a random delay counter-measure. The addition of the random delay counter-measure makes this dataset significantly more difficult to break.

2.6 SNR

The signal-to-noise ratio (SNR) [34] is the ratio between the signal and the background noise of measurement and is given by the following equation:

$$SNR = \frac{Var(signal)}{Var(noise)} \tag{2.10}$$

The signal-to-noise ratio is often used in electrical engineering and other signal processing practices to analyze signal strength. In the field of side-channel analysis, this measurement is helpful if we measure the SNR when computations are performed that involve sensitive values because the power consumption or electromagnetic emission of a device is often dependent on the data used for these computations. Hence, if the SNR is analyzed on fragments of operations involving secret-key (or any exploitable) data, we can locate the areas where the SNR is high and likely to leak exploitable information to the attacker.

Chapter 3

Related work

There has been a significant amount of literature on deep learning strategies within the side channel analysis community. Current methodologies for constructing deep learning architectures are based primarily on trial-and-error and intuition. However, there has also been a significant amount of research on automated techniques that use a predefined sequence of steps, such as reinforcement learning and genetic algorithms, which are common methodology approaches for constructing deep learning architectures. Many approaches are still black-box solutions for improving existing models, generally focusing on adjusting the input or training setup instead of hyper-parameter adjustments.

3.1 Deep learning

The research published on Convolutional neural networks in the general field of deep learning has primarily been results-driven. In [28], it was shown that a convolutional network could significantly improve the state-of-the-art classification accuracy on several image datasets. This revelation resulted in the wide adaption of Convolutional neural networks in the field of image recognition as many of the state-of-the-art approaches since then made use of CNNs [43][44][47]. Many design choices concerning the models used have been based on architectures that have worked well in the past. These architectures resulted from intuition and empirical observations from trial-and-error instead of an architectural methodology. The hyperparameter tuning of these architectures has resulted from trial-and-error and the incorporation of automated processes such as grid search and random search. However, there have been some attempts to improve computational efficiency. Two of the most noteworthy when it comes to a methodology for the construction or topology of CNNs are Genetic Algorithms and Reinforcement learning.

Genetic approaches [14] aim to use a heuristic approach to solve specific types of combinatorial problems. It is often used by first generating an (often random) set of solutions and iteratively selecting viable solutions using a predefined function that analyzes the characteristics to calculate their viability. The iterative process of this algorithm runs until either predefined runtime or performance level is reached. This technique was proposed as a methodology for the construction of CNNs in several works, of which the most notable ones are NEAT [50], hyperNeat [49], and ES-HyperNEAT [42]. However, it was shown in [54] that these approaches are inefficient and not competitive with state-of-the-art architectures in terms of their accuracy in the image recognition field. Reinforcement learning [55] is a method that aims to choose actions based on rewards. This technique was used in [2] to generate high-performing CNN architectures for image classification tasks. In their work, they introduced MetaQNN, which used Q-learning and ϵ -greedy exploration strategy to choose CNN layers. So far, to the best of our knowledge, at the time of writing, these two techniques have been the only successful CNN-construction methodologies in the field of deep learning in general that showed significantly better performance than grid-search or random-search and are not based on manual trial-and-error techniques or manual human-intuition.

3.2 Deep learning in SCA

In [6] the first profiling attack strategy based on a Convolutional neural network was presented. Here it was shown that data-augmentation [46] can be used to prevent over-fitting and eliminate the effect of hiding countermeasures (e.g., misalignment) such as clock jitter. However, in this work, the SCA-datasets were not public. Furthermore, only an abstract view of their network architecture was presented where the explicit choices for hyper-parameters values such as filter size were not discussed in detail. In [3] one of the first challenging public side-channel databases was introduced to research, evaluate and compare deep-learning models and side-channel attack strategies on several databases containing different countermeasures. Here various deep learning architectures were assessed in terms of their hyper-parameter tuning and ability to retrieve the encryption key for encryption schemes protected with both masking and de-synchronization countermeasures. Later [59] became one of the first to introduce several deep learning architectures that provided a significant performance boost to previous models on non-trivial versions of the databases presented in [3]. Their work required less than 95 percent of attack traces compared to the previous state-of-the-art. Also, they had decreased model complexity by 99 percent, significantly improving models presented in [39] and [3] in both training time and guessing entropy. Furthermore, [59] suggested a selection of tools they used to create and assess their convolutional network architectures. However, Zaid et al. misused the word ”methodology” in the title of their work since they failed to produce a sequence of steps that could be used and applied to an unseen dataset to create effective deep-learning architectures in the side-channel-analysis domain. Later [56] showed that parts of the architectures presented in [59] could be substituted by specific data-preprocessing techniques, thereby significantly decreasing the number of trainable parameters, resulting in a significant decrease in training time. In [58] automated hyper-parameter tuning, based on Bayesian Optimization, was used to show that MLPs were sufficient for solving various side-channel datasets, therefore, again, significantly decreasing the complexity of the models required for specific side-channel datasets. However, the datasets used in their work were trivial as they only used SCA datasets without any strong countermeasures. Furthermore, in [38], pruning was presented as a measure to help reduce over-fitting while maintaining high performance with state-of-the-art models in the SCA domain. In [41] Rijsdijk et al. introduced reinforcement learning as a method to generate neural networks for breaking side-channel datasets and obtained relatively good results for both ASCAD and DPAv4 datasets. However, Jorai et al.'s approach does not give insights into the proper construction of CNNs. Another critique is that agents that were used did not seem to learn the underlying relationship between hyperparameters and the side-channel leakage, as they only produced good results on the trivial version of the ASCAD dataset and not on non-trivial datasets such as $N = 100$. Furthermore, this approach is far from optimal as it is a sequential method where relatively few models are found even with high resources and increased running times (seven days). The research presented in [33] was the first to produce an architecture that could do end-to-end analysis by obtaining the complete encryption key using raw traces. They argued that the implicit assumption that preprocessing the side-channel traces to obtain reduced traces using POI was unrealistic, as extracting significant POI from the dataset was not trivial. However, their architecture was relatively large regarding its parameter count, as they used LSTMs, MLPs, CNNs, and auto-encoders. The training time of their models was also quite large, making it unfeasible to run this strategy on machines with fewer resources. In [20] an assessment of loss functions for models in SCA was done, leading to the suggestion of a new focal loss function which showed increased performance compared to other standard loss functions in the SCA domain such as the categorical cross-entropy.

In [39], it was shown that evaluation metrics such as accuracy could be misleading when evaluating deep-learning models for side channel datasets. Additionally [39] also used various data preprocessing techniques for Hamming weight leakage models in order to decrease the amount of traces needed to retrieve the secret key from data-sets containing imbalanced unmasked traces [10][12][5]. In [23] artificial noise was used to increase the robustness and performance of models used on side-channel datasets.

In [36] gradient visualization was used as a tool to localize points of interest and show that this is as least as good as state-of-the-art characterization methods such as the signal-to-noise-ratio; however, this is not very useful in the construction of convolutional neural networks as their method is a post-training method and requires a fully trained convolutional neural network that is able to learn the underlying SCA pattern. In [16], follow-up research on attribution methods was done. In this work, tools

such as saliency maps, occlusion maps, and LRPs were used to detect POI to help embedded systems add more resilience to side-channel attacks by identifying which operations caused the highest leakage. Again in this research, a fully trained CNN or DNN is required and does not attempt to bring any insight into a model's hyper-parameter tuning or optimization. Hence, we can not use these research papers for constructing convolutional neural networks or assessing the ranking of hyper-parameter relevance.

3.3 Research questions

As stated before, there is very little research on the steps required to create such a well-performing deep learning model that can be applied to the SCA domain regardless of which dataset is used. In this work, we analyze the building blocks of CNNs and standard practices from other domains to evaluate the relevant hyper-parameters to produce steps that can be used to construct high-performing models in the SCA domain. We formulate the expansion of our research questions as follows:

Research question 1:

Can we identify structural constraints and hyper-parameter ranges that produce high-performing CNNs and aid in the design and construction of CNNs in the field of side-channel analysis?

- How do these ranges compare to the values used in state-of-the-art?
- Are any clear patterns identified concerning relationships between the architectural components and model performance?

Research question 2:

Can attribution methods from other domains be used to aid in the design of deep learning models for SCA?

- Can attribution methods help us find correct hyperparameter values?
- Can attribution methods reliably help us identify faults in structural components?

Research question 3:

Can we derive a new methodology for the construction of CNNs that gives good results and is easy to follow?

- How do the generated CNNs compare to the state-of-the-art in terms of performance?
- How do generated CNNs compare in terms of trainable parameters?

Chapter 4

Ranges and constraints

In this section, we examine the constraints and hyper-parameter ranges for deep neural network designs in the field of side-channel analysis. We seek to answer our first research question by constructing a hypothesis for each relevant hyper-parameter and verifying its validity through experimentation. Furthermore, we also construct several hypotheses on the architectural components and constraints that can aid in designing CNNs for side-channel analysis. We also compare the different models constructed by using these ranges and identify the crucial elements of these networks. We start by giving a short motivation on the usage of hyper-parameters and architectural constraints in the SCA domain next; we give our approach in section 4.2, our experimental set-up in section 4.3, the examination of our results in section 4.4, and in section 4.5, we conclude this chapter with a discussion.

4.1 Motivation

Hyper-parameter tuning has been quite challenging in deep-learning approaches for different domains. It has been common to resort to random hyper-parameters [58][4] and hyper-parameters found trough exhaustive methods such as grid search [17] [29]. Reasons for using such methods vary from lack of interpretation to insufficient domain knowledge. However, even though these methods may not guarantee top results, they have been proven quite effective in almost every domain in which they have been applied. As for the domain of side-channel analysis, there has been very little research done in terms of methodologies that can be applied to unseen side-channel datasets. Furthermore, the presently available research on methodologies is still quite lacking in interpretability and applicability. Next to this, the term ”methodology” has often been miss-used [59] [56][60] by omitting the most crucial element by which it may be identified: a sequence of steps that can be applied to an unseen dataset in order to generate effective models. In the research done so far, the crucial elements of an adequately working neural network so far have been proven different [41][59][33] for networks with a similar performance which led to different approaches for the development of methodologies where several past explanations [59] [56] have been inconsistent at best. The lack of proper methodologies for developing neural networks in the side-channel analysis domain has to do with the inconsistent findings and their in-applicability for new datasets. These findings suggest that the results obtained by current methodologies may be more in line with those of random hyper-parameters, which, in turn, could incline one to ask the question: can we achieve better results by simply identifying the correct hyper-parameter ranges and architectural constraints for CNNs?

4.2 Approach

A large number of hyper-parameters exist, where each can significantly affect the performance of a CNN in the field of side-channel analysis. Our strategy is to analyze each one and construct a hypothesis regarding the ranges of these hyper-parameters utilizing previous work, mathematical insights, and experimentation. We combine our hypothesis for these hyper-parameters in the results section and test their validity.

For weight initialization we argue the use of the $he\_unifrom$ initializer in all except the last layer as this was the setting that was used in [59] and showed relatively good results in both [59] and [41], as well as in [30][56]. In [30], where a comparison of weight initializers was made, it was pointed out that $glorot\_uniform$ assumed a linear activation function; hence it seems adequate to use this weight initializer in the last layer since this makes use of the $softmax$ activation-function (as this is encryption-key classification probabilities are calculated), which is a linear classifier.

We start with slightly different ranges but are nevertheless guided by the ones suggested in [58] for our hyper-parameter search.

Namely, we initially adjust the number of convolutional layers from $[1, 4]$ to $[1, 2]$; we do this because we would like to see how our constraints compete with the existing state-of-the-art[59] methodology, where a maximum of 3 convolutional layers are used. Consequently, we adjust the ranges based on the models found that perform well on different datasets.

We argue that because convolutional neural networks that have been used so far in SCA tend to be small in size, and hence we are not limited to small filters and can use larger filters, which is why we set the initial ranges for the convolutional-filter size in any layer to $[2, 20]$. In other areas where deep learning is used, such as image recognition, often the convolutional filter size is set to be small (e.g., size $1 * 3$) to prevent performance issues. However, in the field of side-channel analysis, this is perfectly acceptable as models tend to be much smaller compared to other domains and hence are less likely to run into performance issues.

We also introduce mini-batch as a random hyper-parameter, as it was shown in [22] that this hyper-parameter could significantly affect the minimization process concerning the gradient descent algorithm. Another reason is that the effect has not yet been fully inspected for the SCA domain and because both [41] and [59] used different batch sizes for different settings.

Initially, we set the pooling-filter size to $[2, 20]$ and adjusted these ranges based on the observations made throughout this chapter. Note that this is a perfectly valid range as the pooling operation tends to decrease the input size with a factor proportional to the filter size; hence, larger pooling sizes can only speed up the process, as opposed to the larger convolutional-filter size.

Because, in terms of trainable parameters, the contribution of the classification part is far less opposed to the convolutional part of the network, we can also increase the number of neurons for the dense layers from $[100, 400]$ to $[100, 1000]$. Combined with the fact that at the time of writing, most architectures used in the SCA domain have been relatively small compared to other domains; hence, there is no particular reason to limit the number of neurons until proven otherwise. We use the same number of neurons for every dense layer of the classification part, as this has been a common practice for deep learning and because this adds keeps the search space small.

Instead of using a fixed set of learning rates, we use range $[0.0001, 0.001]$ with step-size 0.0001. An overview of the ranges used in initial experiments along with the differences with [58] is shown in table 4.1.

4.2.1 Architectural design

There are several different injection strategies and injection architectures in which one could inject the randomly generated hyper-parameters chosen from the ranges from table 4.1; specifically, there are some essential design choices to be made when considering the convolutional part of the network.

One of the design choices is the matter in which we choose to incorporate pooling [57] into the convolutional part. The amount of different design choices that can be made concerning the pooling layers alone is depicted below in figure 4.1. Due to the vast amount of options available concerning the deep learning architectures and time limitations, we limit our architecture to exploring max-pooling and average pooling as these are often the most streamlined in other domains such as image recognition. We also decide to keep the pooling stride equal to the filter size because, intuitively speaking, if we use a pooling stride that is the same as our filter size, we do not miss any information by skipping data, which can occur when the stride is larger than the filter-size. If we were to allow the selection of a stride smaller than the filter size, we argue that this may lead to unnecessary redundancy in the output of the layer, as this setting would allow the pooling layer to repeatedly take the same input segment into account for more than one output segment.

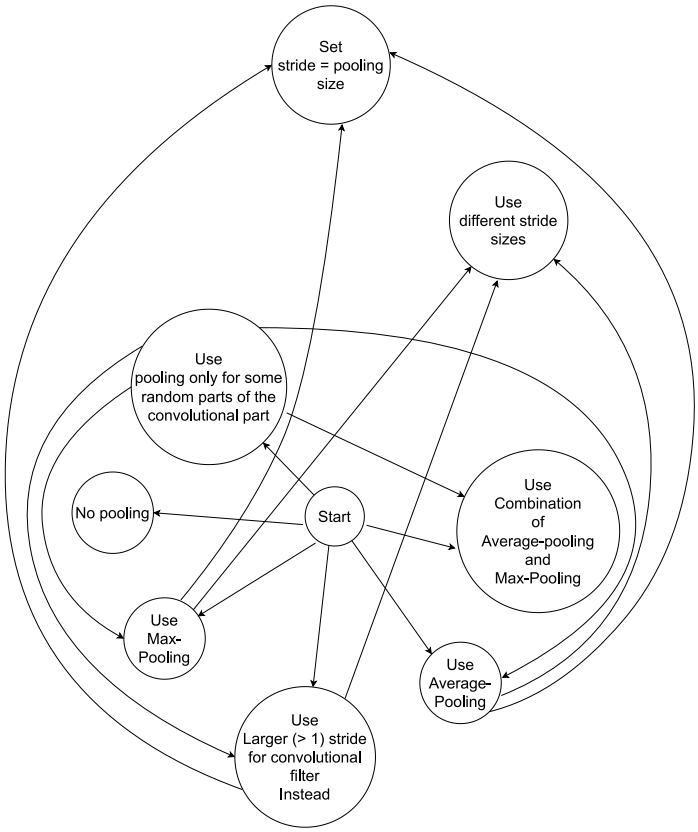


Figure 4.1: test design

We identified several recurring patterns and paradigms in terms of suggested architectures from previous work. In previous work, the most common pattern used for CNN architectures is the input-convolutional-flatten-dense-output pattern. However, some of the more recent trends in the design of CNNs, suggest the incorporation of a GAP layer which aims to either replace the commonly used flat layer, and its output is often directly fed into the final layer of the network, replacing the need for any dense layers in between the convolutional part and the output layer altogether. We want to compare the adequacy of these different paradigms for the SCA domain and how they compare to each other; hence we investigate all three paradigms and see how they compare.

We incorporated batch normalization as a constraint, as shown in [19] to significantly reduce the issue of internal (and external) covariant shift, which can occur when there is a shift in the distribution of the input data. Next to this, it seemed to address the issue of the exploding (or vanishing) gradient problem due to higher learning rates and the issue of getting stuck in local minima during the training process. Combined with the fact that batch-normalization adds very few trainable parameters to the model, we hypothesize that adding batch-normalization layers (as a constraint) to our deep learning architecture can only benefit the training process and generalization of the models using our proposed architectural structure.

It is a standard paradigm to gradually increase the number of filters as convolutional networks grow deeper, with the underlying idea to gradually detect more and more abstract features in deeper convolutional layers. In our architecture, we propose only to choose the number of filters once and multiply it with the number of the convolutional block in which it is incorporated. Our final architectural structure is depicted below in figure 4.2.

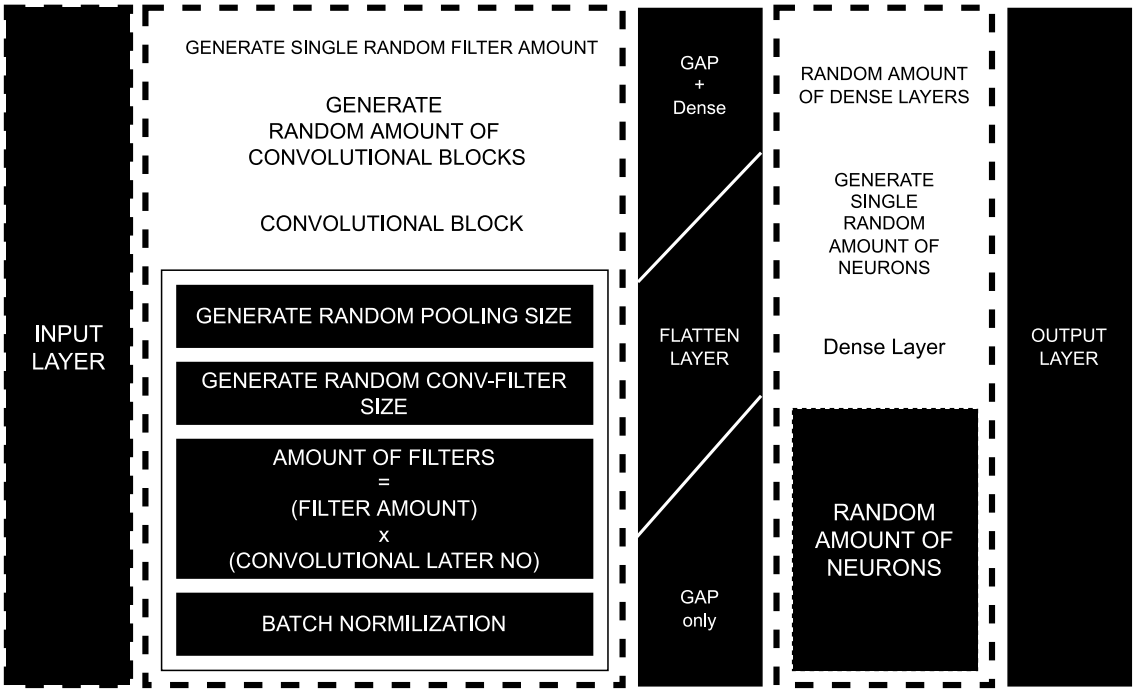


Figure 4.2: The stride and kernel size are generated separately and independently for each convolutional layer. The amount of filters, however, makes use of the same random number across all convolutional layers (which is multiplied by a certain amount for each consequent layer in order to ensure that the latter convolutional layers have more filters than previous convolutional layers). The number of convolutional blocks is random as well, as the amount of dense layers. The amount of neurons is the same for all dense layers.

4.3 Experimental setup

Our experimental environment setup is as follows: For our experiments we use the Tensor-flow[35] library along with the KERAS[8] package to train deep learning models and python's built-in $random$ library for generating hyper-parameter values according to our specified constraints (initially selected from table 4.1).

For each set of chosen hyper-parameters and model architecture, each model is trained 10 times to account for any undesired randomness in the set-up environment. We train the models for 50 epochs.

Hyper-parameter	min	max	step
Convolutional layers	1	2	1
Convolution filters	8	32	8
Convolution kernel size	2	20	2
Pooling size	2	20	2
Pooling stride	2	20	2
Dense (fully-connected) layers	1	3	1
Neurons (for dense or fully connected layers)	100	1000	100
Learning rate	0.0001	0.001	0.0001
Mini-batch	100	700	100

	Options
Pooling Type	max pooling, average pooling
Activation function (all layers)	ReLU, Tanh, ELU, or SELU

Table 4.1: Ranges used for random hyper-parameters where the differences with [58] are highlighted in green.

We train the models 10 times and use the following metrics to assess the model's potential candidacy. Each time we train a model, we calculate the model's average guessing entropy $\overline{GE}$ reached when using 1000 attack traces averaged over 100 attacks. We then train the model 10 times, take the average of the average GE reached over these 10 attempts, and define this metric as MG.

$$(\bar{MG}) = \frac{1}{10}\sum_{i=1}^{10} \bar{GE}_{attempt_i} \tag{4.1}$$

In the same matter, we define MN. Whereas $\bar{N}_{T_{GE}}$ is the average number of traces needed to reach a guessing entropy of less than 1 when using 1000 attack traces averaged over 100 attacks, MN is when we train the model 10 times and take the average $\bar{N}_{T_{GE}}$ reached over these 10 times.

$$(MN) = \frac{1}{10}\sum_{i=1}^{10} \bar{N}_{T_{GE_{attempt_i}}} \tag{4.2}$$

We define the minimum $\bar{N}_{T_{GE}}$ a model reached over the 10 times it was trained as $\bar{N}_{T_{GE_{minimum}}}$. When the algorithm selects a model based on its MG (and MN) as a possible candidate. We confirm these results by building and training this model one single time on a separate pc using the same amount of attack traces (1000 traces averaged over 100 attacks) and defined the $\bar{N}_{T_{GE}}$ and $\overline{GE}$ obtained as $\bar{N}_{T_{GE_{observed}}}$ and $\overline{GE}_{observed}$. Initially we set the algorithm to select candidates based on their MN and categorize them into: possibly converging (where $10 < MG < 50$), good candidates (where $1 < MG < 10$) and strong candidates (where $MG < 1$).

Initial experiments are performed on an HPC cluster equipped with a 1080 TI video card and 8 GB of RAM. Unless stated otherwise, all the experiments are conducted over a period of 7 days.

4.4 Results

4.4.1 Architectures for Hyper-parameter injection

In this section, we experiment with three different architectural patterns commonly used across deep learning domains and investigate their effect on model performance for side-channel datasets. Initially,

we use these architectures alongside the hyper-parameters selected from table 4.1.

The first pattern we investigate is the GAP design pattern (depicted in figure 4.3), where the architecture uses a Global Average Pooling layer [31] after the convolutional part of the network. As previously mentioned, the idea behind the GAP layer is to average all the feature maps extracted by the convolutional part of the network before doing any classification. For this particular experiment, we find multiple models that can retrieve the dataset's encryption key. The result of re-training the candidates found by the algorithm for 1 single trail can be observed in figure 4.6. These results indicate that combining a GAP layer with several dense layers before the final output can result in a beneficial architecture as it can produce good results when applied to side-channel datasets.

In [31] it is argued that global average pooling can prevent overfitting because it directly uses the output of the convolutional part and consequently feeds this to the softmax layer after averaging. Hence, it makes sense to use a similar architectural approach to theirs by removing the classification part and solely use the GAP layer between the convolutional part and the final layer. As this has worked well for many previous approaches, we assumed that this would work well in the case of side-channel analysis, as we suppose that the leakage extracted in the convolutional part of the network may be sufficient enough to make an appropriate classification in the softmax layer of the network.

When we compared the previously obtained results to those obtained by the architecture that solely uses a GAP layer, as depicted in figure 4.4, we observed a significant decrease in model efficiency relative to those depicted in figure 4.6, as no converging models are found for this particular construction.

We assumed that this might have to do with the low amount of features and convolutional layers drawn from our initial ranges. The reason for presuming this is because additional (and more abstract) features may be beneficial for the global pooling operation because there is more information to work with than when using shallow convolutional parts. In order to test out this theory, we adjusted the ranges to include more layers and more filters per layer before running any new experiments. For the experiments with more convolutional layers containing more filters, we did indeed notice a slight increase in overall model performance; however, these experiments did not show a significant enough improvement concerning this particular architectural pattern as, again, here, no model was found that reached a guessing entropy of 1.

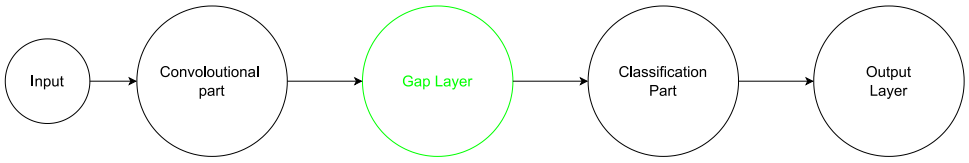


Figure 4.3: GAP + Dense architecture.

A natural hypothesis arising from these results is that the GAP layer itself does not contribute significantly to the results we obtained in figure 4.6. We confirm this by removing the GAP layer and only keeping the dense layers (as depicted in figure 4.5). For this particular construction we achieve far better results than those depicted in figure 4.6 as can be seen in figure 4.7. Hence we can confirm that the usage of GAP layers does not seem adequate for side-channel datasets and that the conventional approach using a flatten layer along with several interconnected dense layers is a far better option for SCA.

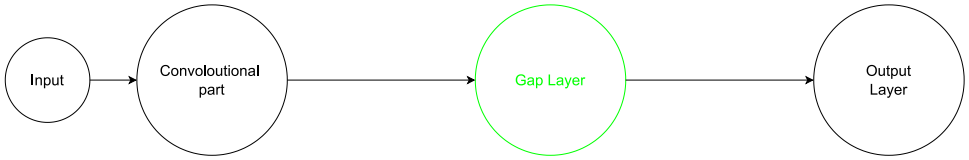


Figure 4.4: GAP only architecture.

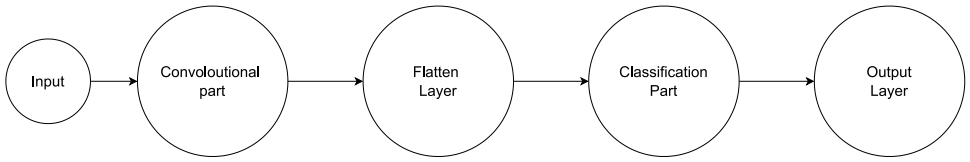


Figure 4.5: The conventional architecture.

We suspect this observed behavior is because the GAP layer averages information, resulting in the contamination of convenient features by averaging it with irrelevant feature maps. The consequence is that the classification part of the network is given much more noisy features than when using the flatten layer, which only flattens the input and preserves all the information as outputted by the convolutional part. Certain specific ranges may exist where this architecture may give better results; however, because not a single model converges below a guessing entropy of 50, we conclude that the GAP layer (whiteout consequent dense layers) does not aid in the detection exploitation of side channel information.

For completeness, we also show that the same algorithm also achieves extremely good results on a more trivial dataset such as Ascad $N = 0$, as displayed in figure 4.7b. The smallest $\hat{N}_{t_{GE}}$ (averaged over 10 trials) found for ASCAD $N = 0$ was 169, which is less than both [59] who achieved 191 and [41] who achieved 172. The smallest MN found for ASCAD $N = 50$ was 397 (where the minimum $\hat{N}_{t_{GE}}$ reached within the 10 trials was 314), where [59] achieved 244 and [41] achieved 313.

Due to these results, we recommend the more conventional structure for convolutional neural networks depicted in figure 4.5 and continue with this particular set-up for the remainder of our chapter on experimental results.

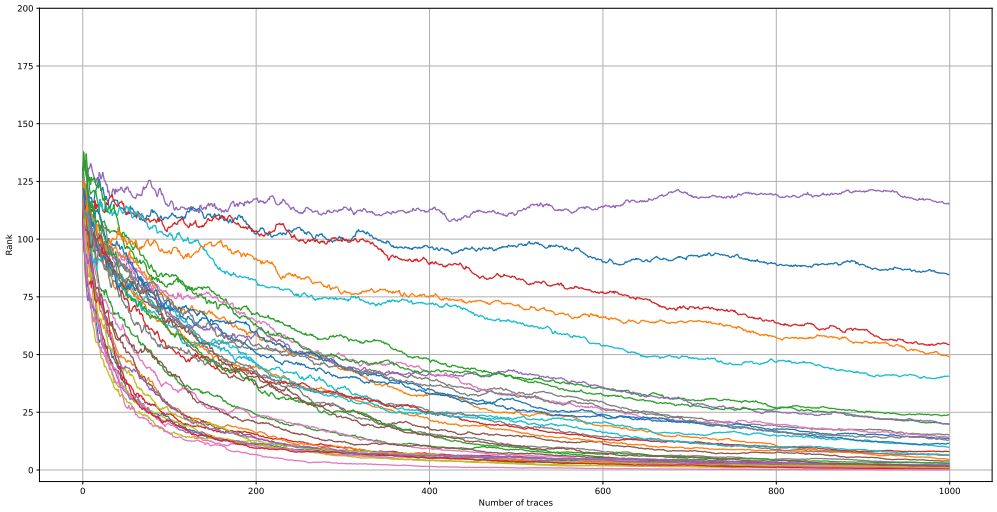
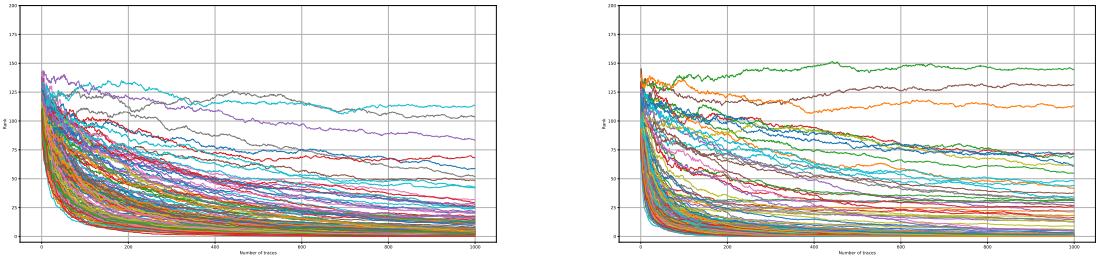


Figure 4.6: Architecture using a common GAP approach as depicted in figure 4.3. Results were obtained using the ASCAD N=50 database and the hyper-parameters ranges presented in table 4.1. Initially, 38 converging models were found where: 34 were potential candidates and 4 were good candidates. When retraining the models (for one trial), we found 4 strong candidates.



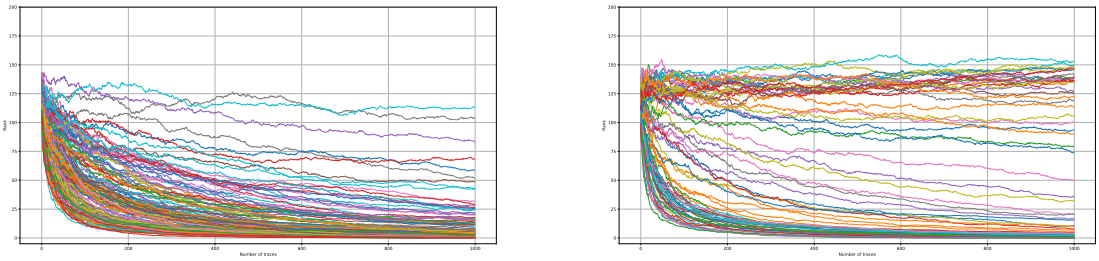
(a) Ascad N=50.

(b) Ascad N=0.

Figure 4.7: Results for running the setup depicted in 4.5 along with random hyper-parameters with the ranges of table 4.1. In 4.7a: re-training of the initially found 145 converging (guessing entropy <= 50) models: Initially 46 potential, 74 good, and 25 strong candidates. When retraining the initially found 145 models for one trial: 30 strong candidates are found, 4.7a shows the result of retraining the initially found 145 models. In 4.7b for ASCAD N=0 initially, 192 converging models were found: 39 were potential candidates, 24 were good candidates, and 128 were strong candidates. When retraining all the initially found 192 models for one trial: 146 strong candidates are found 4.7b shows the result of retraining the initially found 192 models.

4.4.2 Max-pooling vs Average-pooling

We found strong candidates using both average pooling and max-pooling, where the results when using average pooling results were significantly better. As shown in figure 4.8, even-tough we find both potential and good candidates, the initial run did not find any strong candidates; however, we did find strong candidates when retraining the models for one single trial.



(a) Average pooling.

(b) Max pooling.

Figure 4.8: Comparison between average pooling and max-pooling: in 4.8a Average pooling achieved 30 strong candidates during re-training, whereas in 4.8b max-pooling only found 16 strong candidates.

The main reason for this difference could be attributed to the fact that when using average pooling, we take all neighboring values into account that fall within the filter's windows range and retain all relevant information, where only the magnitude of this information is altered. In contrast, max-pooling omits all values except those that are relatively high compared to its neighbors, with the risk of losing valuable information because low channel values can also contain leakage concerning the relevant information. Due to the insights gained from these results, we continue the remainder of the experiments with the average-pooling layer as a fixed choice when injecting the random hyper-parameters for pooling size and stride. The set-up of the architecture random hyper-parameter generation is depicted in figure 4.2

4.4.3 Learning rate and mini-batch

In [59] the One Cycle Learning Rate Policy is applied during model training with the argument that it allows them to use very high learning rates due to the algorithm's ability to adjust the learning rate (based on the accuracy fluctuations) for every epoch during the learning process. In our experiments, we applied the One Cycle learning rate Policy to the models trained with the hyper-parameters selected from our ranges and compared the overall results with those that did not use the OCLP. We investigated its effect on the number of models found and their performance.

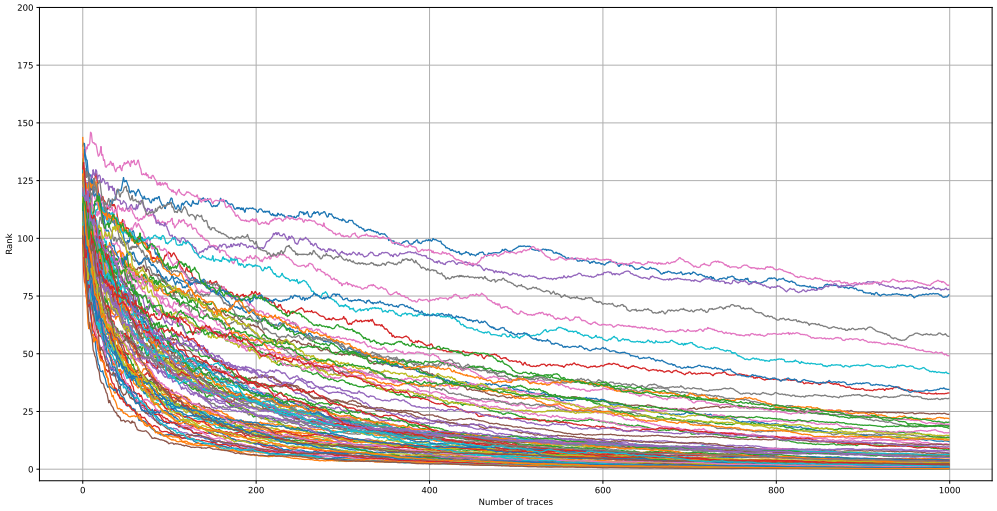


Figure 4.9: using the one cycle learning rate: the result of retraining the converging (averaged a guessing entropy of less than 50 over 10 trials) models found on ASCAD N=50: With 1000 attack traces averaged over 100 attacks,

The algorithm that used the One cycle learning rate found 97 converging (guessing entropy less than 50) models, out of which 12 reached a guessing entropy of less than 1 and 50 reached a guessing entropy between 10 and 1. The results of retraining these 97 models are shown in figure 4.9, where 25 models were found that reached a guessing entropy of less than 1. As can be observed in the comparison made in table 4.2, this is far less than the 145 converging models that were found without using the one-cycle-learning-rate.

Guessing entropy	With One-cycle learning rate	Without
Less than 1	12	25
Between 1 and 10	50	74
Between 10 and 50	33	46
Total	97	145

Table 4.2: One cycle learning rate.

The suspicion behind the poor performance of the algorithm when using the one-cycle-learning rate is that the total training time per model increases by a marginal (often negligible amount) due to the presence of an extra callback during each epoch in order to update the learning rate during the training phase. For the overall algorithm, these marginal computation times may add up significantly in the long run and may, in turn, result in fewer models per time period. Another possible reason is that our ranges for the learning rate may be too small in order to fully take advantage of the benefits provided by the OCLP algorithm; hence, it may be beneficial if we adjust our ranges concerning the learning rate when using this algorithm. Because we observe consistently good results with a fixed learning rate, we believe that the selected range is adequate. These results were expected as 0.001has given good results across multiple domains where deep learning has been applied; hence is often considered the default choice when tuning time is limited. Moreover, because the learning rate is arguably the most critical hyperparameter to tune, we do not think any adjustments would be advantageous as an increased range could result in highly fluctuating results. Nevertheless, as we did not see any apparent benefit in our experiments, we continued the remainder of our research in this chapter without the OCLP, as we found more architectures to inspect when this is not activated.

Hence for the remainder of our experiments, we do not use the one-cycle-learning rate when training the models with randomly generated hyper-parameters from our pre-defined ranges.

Concerning the batch size, we found that models with both small and large mini-batches could reach a guessing entropy of 1; however, we found that the best performing models used a batch size of > 200

during their training. We assume this is because a larger mini-batch can enable faster computations (because we have fewer steps per epoch) while reducing the noise in the gradient [21], allowing the network to learn easier. The variations in the gradient are eliminated because we use a more significant portion of the samples, and hence the gradient becomes less subject to high variance. Taking this point into account, we argue that it is more likely for a model to converge when using a larger batch size. Of course, it is possible to choose a batch size that may cause the learning algorithm to get stuck in local minima; however, we did not observe this in our experiments because multiple models are found with a batch size of 700. The best performing models found on $ASCADN = 0$ used a mini-batch size of 100, whereas the best performing models on $ASCADN = 50$ used a mini-batch size between 400 and 600. This observation might suggest that larger batch sizes may be more interesting for non-trivial datasets.

4.4.4 Activation function

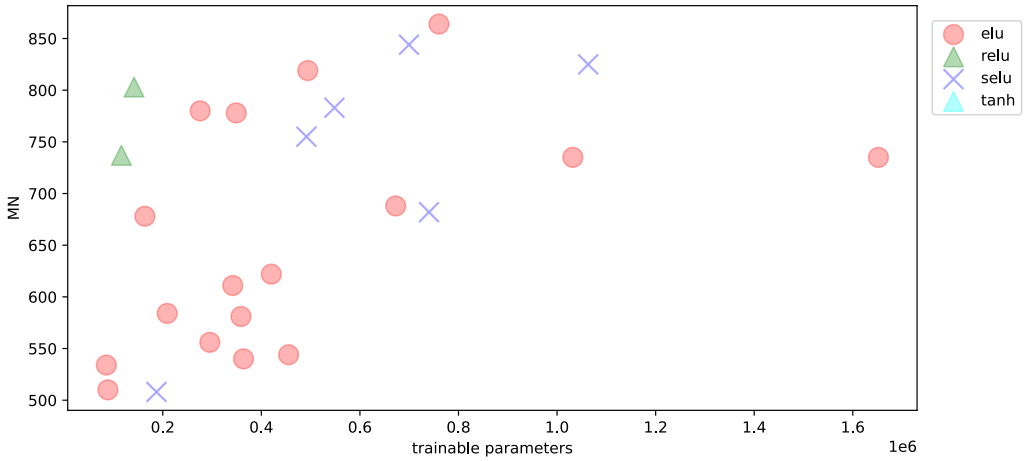


Figure 4.10: Distribution of the models (that were re-trained figure 4.7a) found by the algorithm where MG < 1 in terms of their MN and amount of $trainable - parameters$ for ASCAD n=50.

When further analyzing the models from figure 4.7a, we see an interesting pattern concerning the activation functions that the converging models employ. We depicted the activation functions and model parameter-count in figure 4.10, where we clearly see that the converging models tend to favor the SeLU, ReLU, and ELU activation functions over the Tanh functions. In [59] the $SELU$ activation-function is recommended; however, figure 4.10 shows that ELU was more common amongst the models selected by the algorithm.

Parameters	Activation-function	Ntge observed	Ntge min	MN	MG
187192	selu	495	398	508	0.112
88604	elu	710	405	510	0.11
85500	elu	875	391	534	0.127
363844	elu	506	322	540	0.186
455336	elu	773	412	544	0.115
295456	elu	418	409	556	0.136
358648	elu	593	404	581	0.16
209076	elu	525	498	584	0.162
342060	elu	664	417	611	0.431

Table 4.3

Looking more closely at the activation functions, we see that their ranges (depicted in figure 4.4) show that Tanh, ELU, and SELU each contain negative values, whereas the RELU activation function

only contains positive values. According to [9] the ELU activation function gives good results because it contains negative values, which allow the ELU activation function to guide the mean activation function toward 0, which could explain the good results obtained compared to the ReLU activation function. However, when taking the Tanh activation function into account, we see that this activation function also contains negative values but does not seem to contain any converging models for the side-channel dataset. Further investigation into the differences in the mathematics behind the activation functions shows that compared to other activation functions, the Tanh activation function has one major drawback concerning the training process of models, namely, when the input x of the activation function is positive, we do not get the (scaled) identity x but a value between 0 and 1 which can result in a small derivative (gradient). This phenomenon can cause the gradient to become too small when x is large and cause the gradient to disappear (which is often referred to as the vanishing gradient problem).

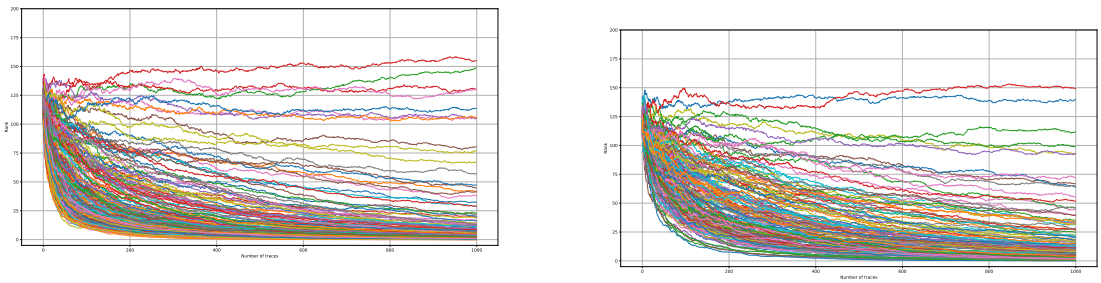
Note that ELU and ReLU do not have this problem because the positive values of x result in x (in the case of SELU in λx where $\lambda > 1$), as can be seen in figure 4.4.

af	minimum value	maximum value
relu	0	x
elu	$a(e^x - 1)$	x
selu	$\lambda\alpha(e^x - 1)$	λx
Tanh	-1	1

Table 4.4: ranges of the activation functions used in the experiments.

Hence we conclude that Tanh is not a good activation function when constructing DNN for side channel analysis because it is more prone to the vanishing gradient problem than the RELU, ELU, and SELU activation functions. We confirm these results by repeating the experiment multiple times for each (fixed) activation function. The results when fixing the activation function are given below.

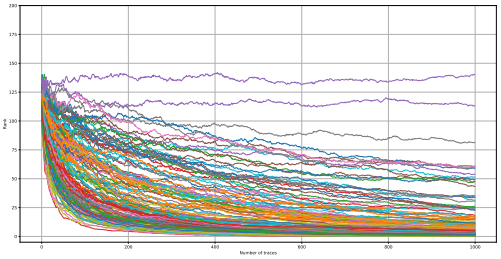
The results when keeping ReLU, ELU, SELU, and Tanh fixed confirm our hypothesis. Note that we argue against any form of manual pre-processing (before training the model) in this thesis because we would like the convolutional part to do our processing for us. Both [] and [] used pre-processing for specific datasets, where the data values were normalized between 0 and 1. We leave research surrounding pre-processing (for example, choosing normalization ranges of (0,1) or (-1,1)) for future work. We further attribute the differences between the ELU and SELU activation functions (where positive values are linear in x) to the scaling factor $\lambda > 1$, which was suggested to work well with the $lecun - normal$ for weight initialization, however seeing as we make use of the he-normal weight initializer, its does not seem to outperform ELU.



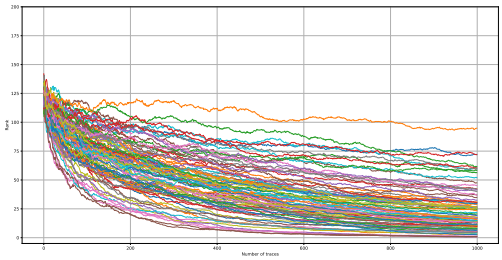
(a) $N = 50$. 59 Potentially-converging, 94 good-candidates, 55 strong-candidates. retraining 74 strong candidates.

(b) N=100

Figure 4.11: ELU.



(a) $N = 50$. 64 Potentially-converging, 74 good-candidates, 19 strong-candidates. When retraining 37 strong candidates.



(b) N=100

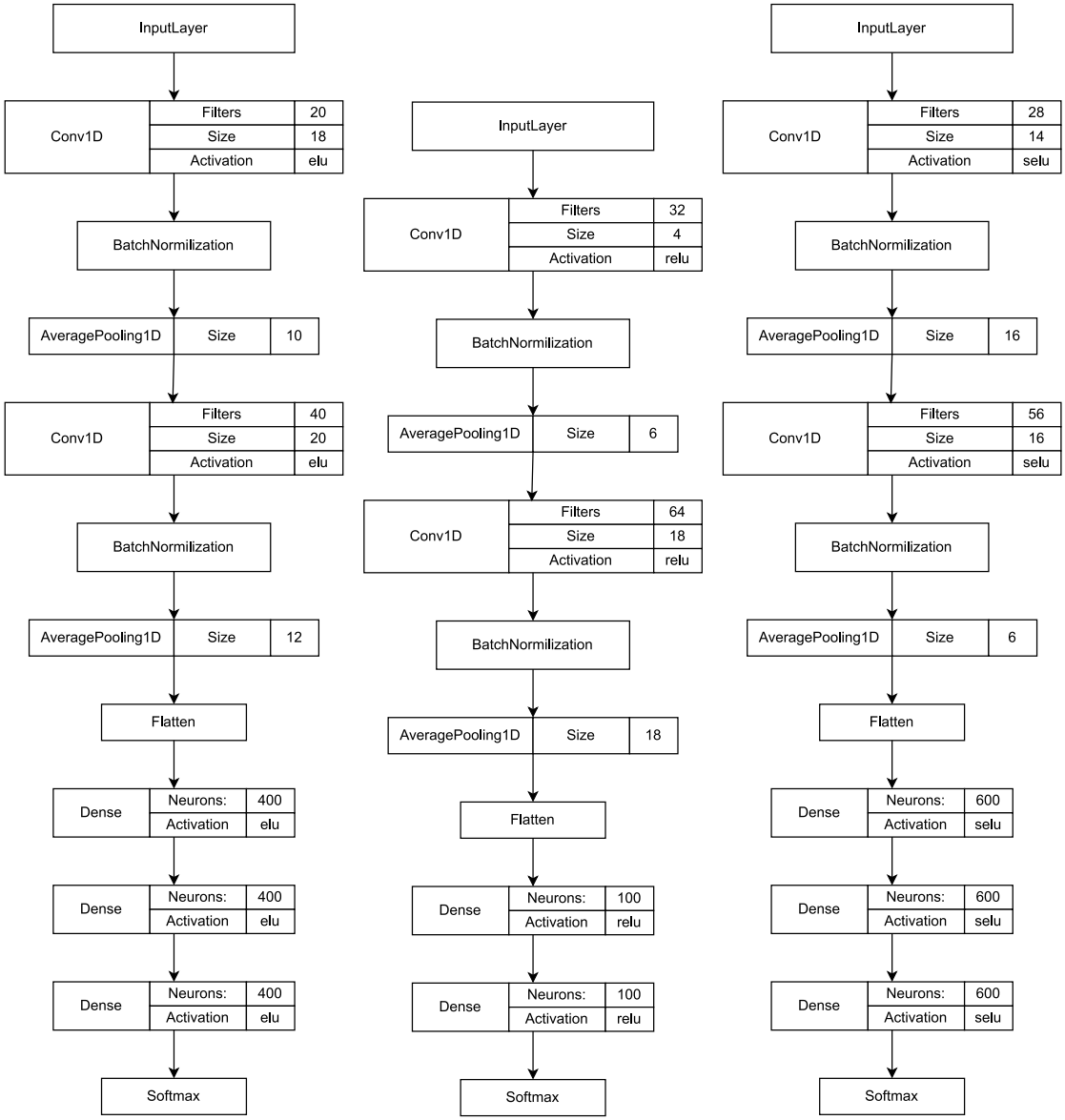
Figure 4.12: $SELU$.

No converging models for Tanh were found, which is supported by some of the findings presented in [58] and in figure 4.10.

As we can see, our hypothesis regarding the activation functions is confirmed. If we look at the models found in more detail, we can see that based on both the number of models as well as the performance of the models, we achieve outstanding results which confirm our hypothesis regarding both the hyper-parameters as well as the architectural constraints we discussed in our approach section of this chapter. We can see the top 10 models for each of the activation functions depicted in table 4.5. If we look closer at the top models for each activation function, we see that they have a large variance in their hyper-parameter values, as shown in figure 4.13. Due to this observation, we analyze the remaining hyper-parameters on the construction of the models found in more detail in the next section.

	ReLu		ELU		SELU	
Rank	parameters	$N_{T_{GE}}$	parameters	$N_{T_{GE}}$	patameters	$N_{T_{GE}}$
1	111736	461	69540	326	1136588	435
2	72272	489	63036	387	1650388	492
3	98260	652	520396	407	2335540	509
4	60020	688	2715484	431	404324	514
5	84536	719	514984	446	2745048	534
6	2732864	787	301300	452	168232	553
7	60052	798	1317300	469	714256	580
8	618624	805	545624	498	974432	582
9	1075604	867	788884	508	1541400	613
10	480192	872	301628	529	900600	637

Table 4.5: Top 10 performing models for activation functions ReLU, ELU and SELU on ASCAD $N = 50$.



(a) elu: with mini-batch 400 and learning-rate: 0.0003.

(b) relu: with mini-batch 300 and learning-rate: 0.0005.

(c) selu: with mini-batch 200 and learning-rate: 0.001.

Figure 4.13: Comparing the top models found during the initial round of experimentation.

4.4.5 Range assessment

Convolutional filter sizes

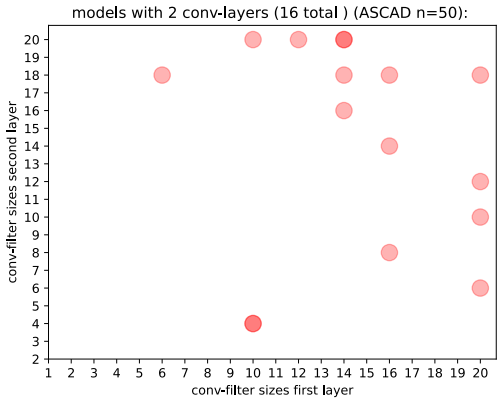
When analyzing the convolutional filter sizes of the models found, we find that they are widely spread out in the selected ranges; however, we did notice a higher number of strong candidates with a larger convolutional filter size, as can be observed in figure 4.14. We suspect that this phenomenon has to do with the fact that a larger convolutional filter has a larger perceptive field and hence can extract useful features from desynchronized traces more easily than smaller filters.

Smaller filter sizes can detect local features. Larger filter sizes can detect global features. Hence, it stands to argue that we may want a more global view of the traces when traces are misaligned. However, when misalignment is not present, larger filter sizes are not extremely useful as, in this particular scenario, we are more interested in local features.

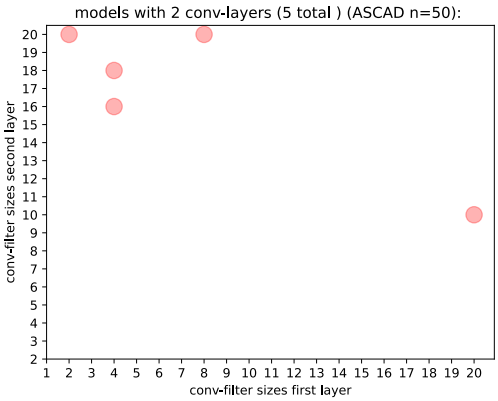
We investigated this phenomenon further by increasing the range for the convolutional filters from $[2, 20]$ to $[20, 40]$ (figure B.1), and $[40, 65]$ (figure B.2). As we can see in figure B.1 and figure B.2, the

increase in the range also caused an increase in the number of strong candidates; however, when we look more closely at the performance of the models, we see that the performance gain is almost negligible. Because of the negligible performance gain, we conclude that the ranges $[2, 20]$ are sufficient when solely considering the $N_{T_{GE}}$ metric. We suspect that the increase in strong candidates may have to do with the dataset used and the counter-measures employed by the encryption algorithm, which we investigate further in chapter 5.

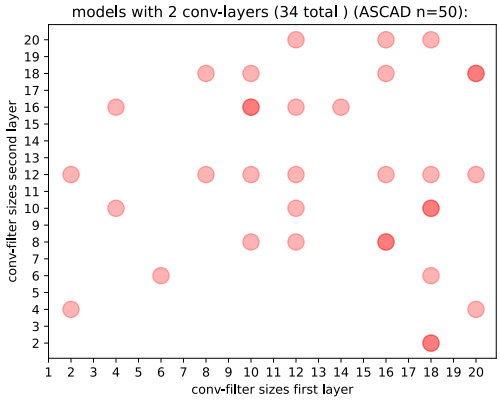
One conclusion that we can draw from these experiments is that outside of the fact that our ranges provide enough strong candidates, models with larger convolutional filter sizes train well on SCA datasets as opposed to other domains such as image recognition (where only small filter sizes are used). When training the models with a more extensive convolutional filter size range, we also noticed that the memory usage was higher, and thus this could make a difference if we would require larger models (which so far has not been the case for the side-channel domain). The overall increase in training time was negligible for the individual models. Even when increasing the number of convolutional layers, we did not notice that the convolutional filter size significantly impacted the overall model performance or the training time.



(a) selu



(b) relu



(c) elu

Figure 4.14: Assessing the filter sizes of the converging models: ranges $[2, 20]$.

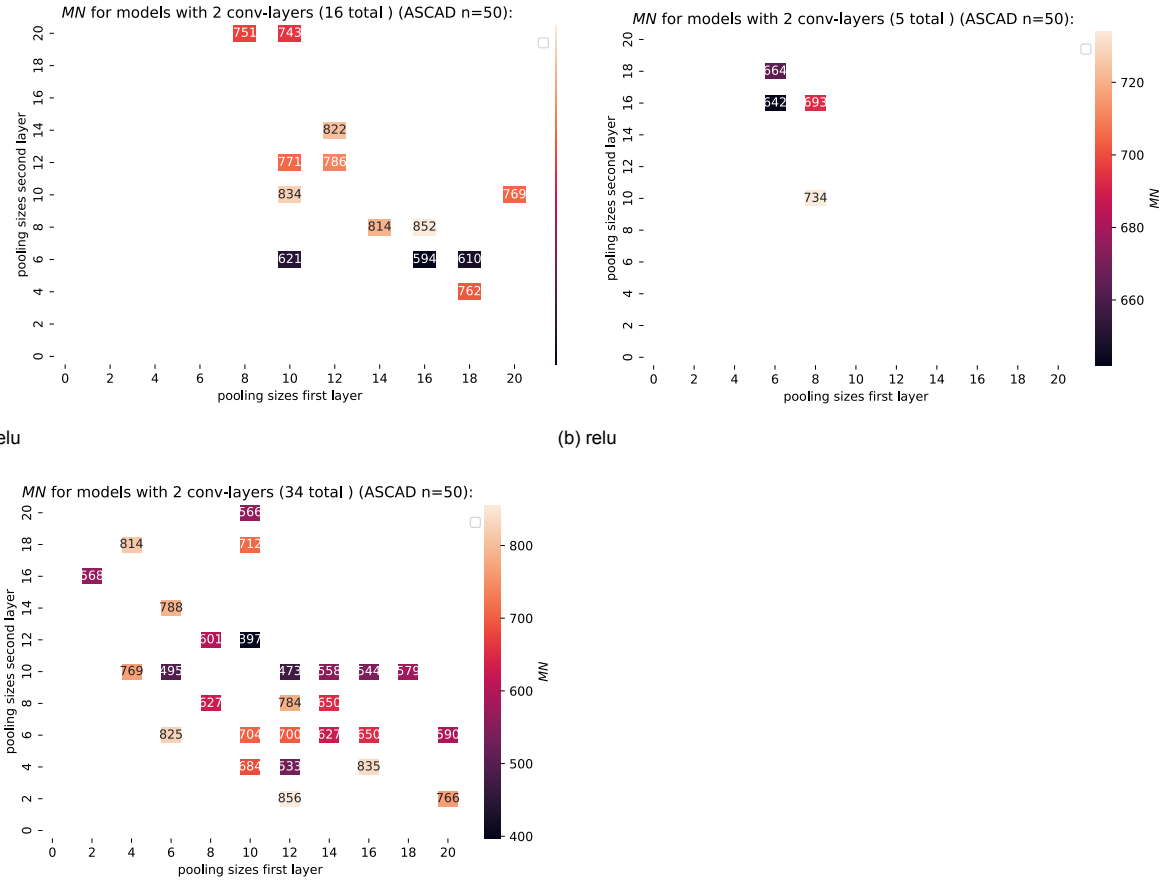
It is possible that even though smaller filters can detect more local and larger ones more global features, both can be used by the classification part to ultimately construct the same abstract information that may be required to ensure the correct classification of a trace. This possible theory lends itself to the fact that it was shown in [47] how multiple sequential smaller convolutional filters (without spatial pooling in between) can replace larger ones (e.g., two sequential $3X3$ filters can replace a $5X5$ filter, and two $5X5$ filters can replace a $7X7$ filter). However, contrary to their architecture, because we chose to incorporate a pooling (as well as a batch normalization) layer in between sequential convolutional layers, we may need to consider the pooling layer's effect when assessing the model's feature-extraction

mechanism.

If we consider that larger convolutional filters can detect more global features and smaller ones more local features, we also have to take into account what the pooling filter will do to the extracted features. The pooling layer can increase the network's ability to detect more shift-invariant features; hence, we argue that many of the global features that a larger convolutional filter may detect can also be detected by a smaller filter provided that it is combined with a pooling layer containing the correct pooling filter size. This is another possible theory on why the use of different convolutional filter sizes may result in the same performance; however, because multiple factors are involved, we do not immediately eliminate the effect of other hyperparameters on the choice of the filter size, and further empirical studies may give better insights into this phenomenon.

Pooling filter sizes

We investigated the effect of the pooling layer (where we constrained the size and stride to be of the same size) and plotted these values against the model performance. As shown in figure 4.15, we see that different combinations of sizes yield the same level of performance. We also see that filter sizes are widely spread over our range selection, which leads us to conclude that the ranges we defined are adequate for this specific dataset when combined with our architectural constraints.



(a) selu

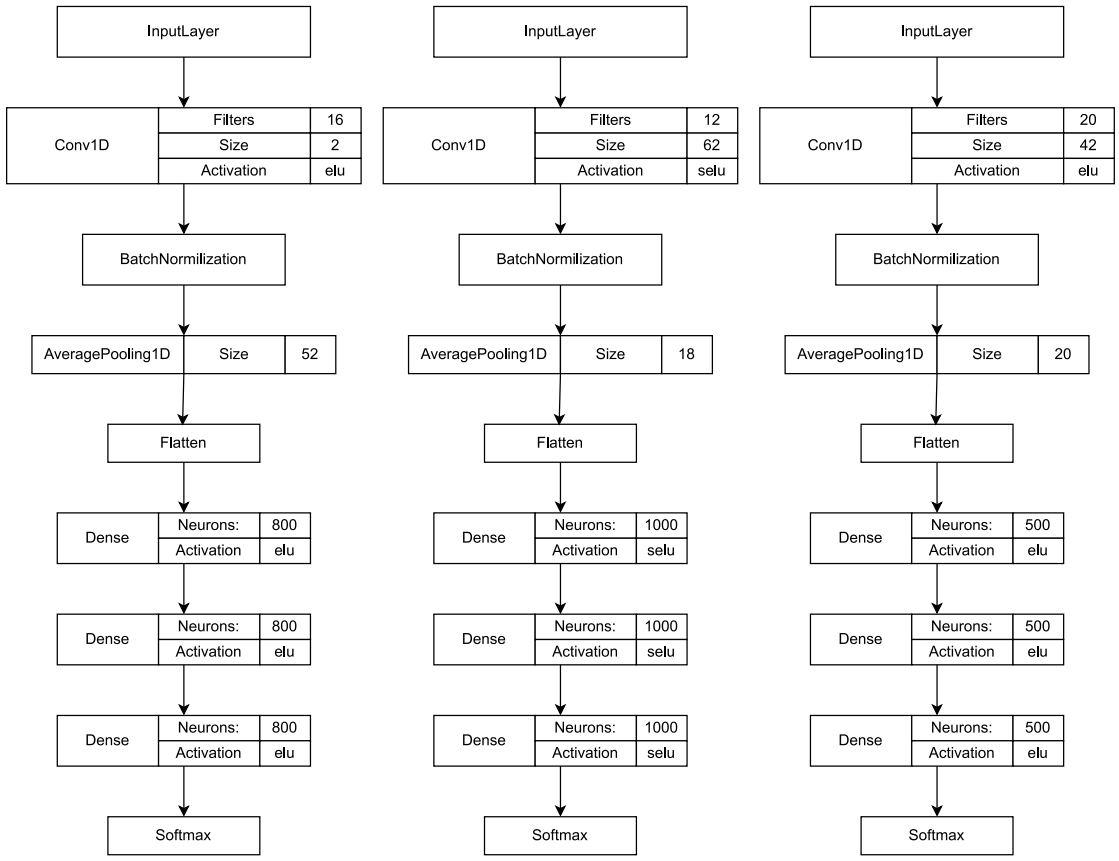
(b) relu

(c) relu

Figure 4.15: Assessing the pooling sizes of the converging models.

Another interesting thing to point out is that when increasing the pooling ranges from $[2, 20]$ to $[20, 35]$ (while keeping other ranges the same), we only found strong candidates that solely consist of 1 convolutional layer for ELU, even though the ranges for the number of convolutional layers is 2. We see this same phenomenon for the SeLU activation function, where we find 51 good candidates with one convolutional layer and only 16 with two convolutional layers. As we can see from the experiments

depicted in figure B.3 and B.4, the convolutional filter size can be small and still be able to detect leakage when the pooling size is large. A more detailed example of such a scenario is given in figure 4.16. This observation would suggest that the pooling size may be far more significant than the convolutional filter size for certain datasets. As we also notice an increase in the number of models found when increasing the filter ranges, we hypothesize that the choice of the convolutional filter size, as well as the choice of the pooling filter size, may be dataset-dependent and may have more to do with dataset properties as opposed to other hyperparameters settings. We will assess the pooling element further in chapter 5, where we also consider dataset features.



(a) mini-batch 500 and learning-rate: 0.0003, min $N_{T_{GE}}$: 343, avg $N_{T_{GE}}$: 439.

(b) mini-batch 200 and learning-rate: 0.001, min $N_{T_{GE}}$: 437, avg $N_{T_{GE}}$: 659.

(c) mini-batch 400 and learning-rate: 0.0008, min $N_{T_{GE}}$: 378, avg $N_{T_{GE}}$: 561.

Figure 4.16: Models with single convolutional layer (ASCAD $N = 50$): Example where a model with a small convolutional-filter size of 2 (figure 4.16a) along with a large pooling size, outperforms models with a large convolutional-filter size (fig 4.16b and 4.16c). This observation suggests that there may be a difference in the level of importance when comparing convolutional size to pooling filter size.

Classification part

When inspecting the number of neurons used in the dense layers of the models found, we see that they are evenly spread out and find well-performing models regardless of the number of neurons chosen. When inspecting the number of neurons used in the dense layers of the models found by the reinforcement learning methodology [41], we notice that often the amount of neurons found is relatively tiny $<= 15$ with a maximum value of 20, the same counts for the models in [59], whereas we use a maximum of 1000. Hence we attribute this observation to the fact that our range selection for the classification part is more than capable of approximating the underlying distribution function associated with the leakage. We could choose to lower them; however, because models in the field of SCA tend to be small in size, we argue that our selected ranges are a proper choice as the number of neurons does not add a significant drawback to the training time and because other future SCA data-sets may require more capacity.

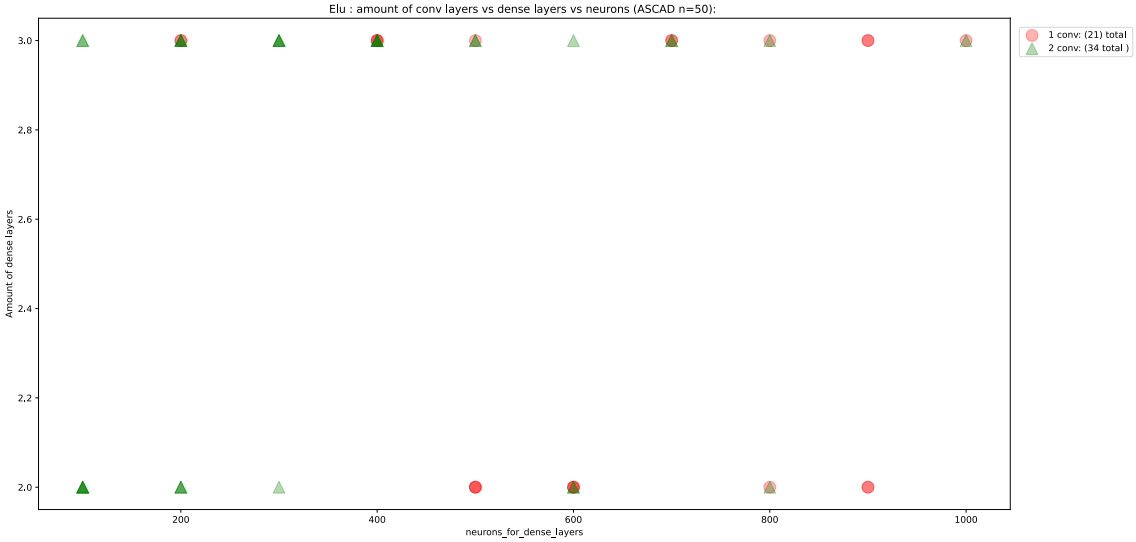


Figure 4.17: Elu: Amount of dense layers vs amount of neurons (for models with both 1 and 2 convolutional layers).

4.4.6 Deeper networks

Even though we obtain competitive results with models that only contain either 1 or 2 convolutional layers, we pay attention to the fact that the models from [41] use at least 3 convolutional layers. Furthermore, the only models from [41] with which the results were reproducible were models that contained either 4 or 5 convolutional layers. In this section, we aim to investigate models' performance when using an increased number of convolutional layers.

When extending the ranges for the number of convolutional layers, we are faced with the possibility that, due to our selection of pooling ranges, certain combinations of random hyper-parameter selections could result in a model that produces an output of size 0, which could happen if the pooling sizes used in the convolutional-part of the network are too large. Hence, those specific combinations of hyper-parameter values can produce an invalid CNN model.

We demonstrate this problem using the input and output sizes of the convolutional part for the $ASCAD$ databases depicted in figure 4.18. The $ASCAD$ database contains traces of length 700, and when passed to a pooling layer of size x, the output of this pooling layer is $700 * \frac{1}{x}$ and if this output is again passed to a pooling layer, it is again divided by pooling size x (since we use a pooling-stride equal to the filter-size of that pooling layer) until the last pooling layer is reached. If this value becomes less than one at any point, the input for the next layer is considered invalid. When a trace has passed through the last convolutional block, we refer to its dimensions as T_{LC}.

Note that when we used a range of 2 for the number of convolutional layers, we were not faced with this problem because the lowest value that could be for T_{LC} was $700 * \frac{1}{20} * \frac{1}{20} = 1,75$.

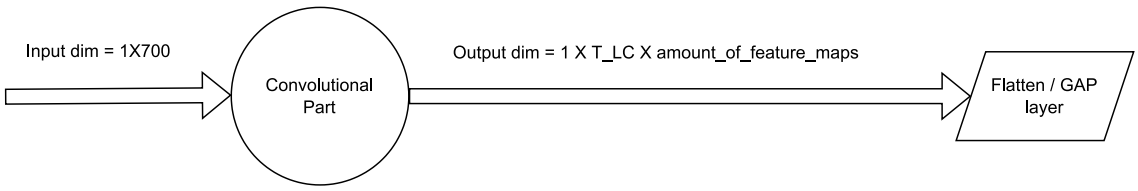


Figure 4.18: The value of T_{LC}.

To prevent this from happening, we can either adjust the ranges of the pooling layers or only select the models where the T_{LC} value is above a specific value. We chose to do the latter. We also hypoth-

esize that the value of T_{LC} may have some relation to how well the network performs, and hence we investigate this by repeating the experiments for different ranges. The results can these experiments be observed in table 4.6. Although we do see an increase in the performance of the models that have more convolutional layers, there is currently not a clear indication of the value of T_{LC} having a significant impact on the performance of the models, as can be observed in figure []. Hence from this experiment, we can conclude that the amount of features is not as relevant as the type of features that the classification part receives from the convolutional part. We investigate this phenomenon further in chapter 6, where we also consider the dataset features.

	range [1,2]			range [3,4]			range [5,7]		
Model	Parameters	MN	Ntge min	Parameters	MN	Ntge min	Parameters	MN	Ntge min
1	520396	397	314	321728	315	174	730064	337	235
2	179804	473	352	361432	344	275	2249480	349	294
3	78836	483	381	2120668	362	275	562264	393	254
4	252128	495	387	2056332	410	330	1357816	407	284
5	1576760	509	436	123468	458	352	2524236	407	250
6	169196	533	425	83144	466	297	167720	416	306
7	514984	544	412	1828880	474	301	1813704	419	309
8	67948	558	308	714976	495	326	358156	423	331
9	63036	566	307	131892	509	366	187648	436	320
10	117852	568	380	377168	523	460	3130444	471	323

Table 4.6: Deeper models: increasing the number of layers for the architectural structure presented in figure 4.2.

4.5 Discussion

Although the GAP layer has worked well in other areas [31] where Deep learning has been applied, in the experiments performed with our current setting with specific ranges and structural constraints, it does not seem to perform well with side channel data. We presume this is because much information is omitted as we average over all the feature maps. The omitting of certain information in the field of SCA may have a less desirable effect than it does on other fields where DL is applied (such as image recognition) because there is an active attempt to hide the exploitable information. The exploitable information may be small at any given time point of the trace as it may also contain other information such as background noise and other irrelevant computations. Hence, any vital information extracted through one filter can become polluted by those extracted by a different one.

Furthermore, we see the same effect when comparing max-pooling to average pooling. We observe that when substituting the average pooling layer with one that uses max-pooling, the performance of the overall models is slightly decreased. Again here, we assume that this is because the max-pooling layer only retains the larger values and omits contained by the window filter at the time of pooling, and hence has a higher dismissive rate than average-pooling layers (where information is only averaged, which leads to less information loss).

In our experiments, we also observed that the pooling filter size might have a higher significance when it comes to model performance as opposed to that of the convolutional filter size. We observe that we achieve similar performance with different filter sizes across multiple layers, where we see that, even though we find more models with a large filter, their performance is not significantly higher than the models using smaller filter sizes. We assume that the larger filter size's ability to detect more global features may have a more beneficial effect on datasets with some form of random delay, as we can obtain a more abstract few of the properties, as opposed to datasets where there is no random delay and the detection of local features may be more beneficial. The pooling layer, however, can significantly enhance a model's ability to detect shift-invariant features and hence may make it easier for models to detect any sort of feature even when using smaller filter sizes.

Concerning the number of hidden layers and neurons within these layers, we know that the universal approximation theorem [18] states that a feedforward network with only one hidden layer and a finite number of neurons can approximate any measurable continuous function. Our experiments found that our original ranges gave good results without any clear indication of a preference for 2 or 3 hidden layers. Also, when looking at the number of neurons per layer, we did not see any clear indication of choosing one over the other as we found similar performance for models with fewer neurons as those with more. This result was expected as previous research used a relatively low amount of neurons in their experiments with models that showed convergence. Contrary to a common assumption that models using a high amount of neurons in their dense layers were more likely to overfit due to their high capacity, we did not find any evidence of this during our experiments, which indicates that the high capacity of our ranges is still a good selection.

Contrary to the universal approximation theorem, it was shown in several works [52] [53] that there may be significant benefits for a deeper classification part as opposed to that of a shallow one; however, this may be dataset dependent and may need to be empirically determined. Taking this into account, we also did some experiments with ranges that used a deeper classification part, where we set the number of layers between 5 and 10 with fewer neurons per layer. Also, for these experiments, we only noticed a slight difference in results regarding the number of models found. We did not see any significant differences in model performance for the experiments conducted so far with the databases selected. These observations indicate that, at least for these databases, three layers may be more than sufficient.

When using a deeper convolutional part, we notice an increase in the overall number of models found and the individual models' performance. We assume this is because we can abstract more feature information and have deeper layers learn weights that can adjust features from previous layers, making them more useful for the classification part of the network. Again, this may also be dataset-dependent and is an element we look more closely at in the next chapter.

In this chapter, we have compared different architectural approaches and assessed their adequacy for the side-channel domain. Using the insights gained from our comparison, we developed practical architectural constraints for model construction. Furthermore, we have identified effective ranges that work well with our architectural constraints and produce effective convolutional neural networks for the side-channel analysis domain. The benefit of our approach lies in the way we assess our ranges. As stated in section 4.3, the algorithm trains a model 10 times before any selection procedure starts, which has the benefit that candidates selected by the algorithm perform well, even in the presence of undesired randomness, and ensures the reproducibility of their results and ensures our range assessment is robust (without any form of pre-processing). Not every existing methodology does this; whereas we take the average performance over 10 trials into account, models selected by the methodology in [41] do not. Yes, the methodology used in [41] produces good results; however, this only happens occasionally. As depicted in appendix A, we attempted to reproduce the results from the top-10 models that were found by the reinforcement learning methodology and only found 3 of the top 10 models showing signs of conversion (even-tough pre-processing), where one particular model showed strong convergence, as can be observed in Appendix A (table A.2).

Hence, because our strategy consistently produces a high number of high-performing models with reproducible results, we conclude that both our range assessment and architectural assessment are fair and that the ranges and architectural constraints are indeed valid and effective for constructing Convolutional neural networks for the side-channel analysis domain. Note that our assessment strategy can easily be applied to any side channel analysis dataset. In the next chapter, we look at the dataset attributes and interpret-ability methods with the intent to identify what the relationship between hyper-parameters and different dataset characteristics looks like.

Hyper-parameter	min	max	step
Convolutional layers	1	2	1
Convolution filters	8	32	8
Convolution kernel size	2	20	2
Pooling size	2	20	2
Pooling stride	2	20	2
Dense (fully-connected) layers	1	3	1
Neurons (for dense or fully connected layers)	100	1000	100
Learning rate	0.0001	0.001	0.0001
Mini-batch	100	700	100

	Options
Pooling Type	average pooling
Activation function (all layers)	ReLU, ELU, or SELU

Table 4.7: Ranges used for random hyper-parameters where the differences with [58] are highlighted in green.

Chapter 5

Attribution methods

Many deep learning architectures have currently been treated as black box solutions where one only analyzes the output of the final layer and computes an error relative to the expected value. In case of poor performance (e.g., high error rate), it has been a traditional approach to adjust hyperparameters or input data until the error rate has become sufficiently small. It has been scarce in the past that one tries to interpret individual components of the deep learning architecture itself. There are mainly two reasons why interpretability has been lacking so far; one is a lack of domain knowledge, and the other is a lack of deep learning knowledge, as it is scarce to find an expert in both categories. Another factor to take into account is that when dealing with deep learning models, there may be thousands (in some cases millions) of features that are extracted, which can make it a cucumbersome task to assess each feature individually in terms of their meaning and relevance, and figure out where the fault lies. This is often why often times automated empirical methods (e.g. reinforcement learning, genetic algorithms) are used to fine-tune model hyper parameters concerning their decision making process; however these approaches do not solve the black-box lemma.

Evaluating machine learning models by only using empirical methods has been prone to producing models that exhibit unexpected behavior, especially when models were given unseen data in unanticipated scenarios; this has led to surprising results in the past in both moral and technical settings.

In this chapter, we investigate the applicability of attribution methods in the side-channel analysis domain and see whether these can be used to gain more insights into the models decision-making process, and, in turn, whether these can be exploited (within the time frame of this thesis) in order to generate high-performing CNNs for side-channel datasets. Specifically we aim to answer research question 2. The chapter is set up as follows: first, we give a short motivation in section 5.1 concerning our approach, in section 5.2, we compare the saliency maps of different high-performing models in order to answer the question of whether or not different CNNs look at the same part of the input in their classification procedure and investigate whether a high performing model can make use of data-segments other than those that are indicated to have high leakage according to common leakage assessment tools (such as SNR), for its decision-making process. In section 5.3, we investigate the extent to which the hyper-parameters values of a model can be determined through knowledge of dataset-specific features and the use of visual attribution methods, after which we conclude this chapter with a discussion in section 5.4.

5.1 Motivation

Attribution methods are often used to gain insights into a neural network's decision-making process by determining the input areas that influence a model's output and by what percentage. It has been commonly used in image-recognition models to determine whether the pixels belonging to an object are used to classify it or whether the pixels in the background contribute more to the determining factor. Hence they are often used as a tool to enhance both the interpretability and explainability of a model's decision-making process, which has often led to the construction of more robust models due to insights gained on the relationship between deep-learning architectural components (such as hyper-parameters) and input contributions toward object classification. As the use of deep learning in SCA has

shown to be quite effective on several accounts [3] [41], it has grown into one of the more mainstream methods for performing side-channel analysis attacks in somewhat practical settings. However, even though it has been an effective strategy in the side-channel domain, even when counter-measures are present, it is more often used as a black-box method to solve the key-retrieval process for side-channel data. There have been few attempts made [59] where the model's inner-workings were taken into account in terms of interpret-ability and explain-ability; however, these claims were shown to be somewhat lacking at best [56]. Moreover, as we have seen from the previous two chapters, the models that can solve a side-channel dataset may vary in terms of the hyper-parameters used. Although typical attribution methods have been used in the past for SCA [16] [36], they were used as leakage detection tools for SCA datasets and the localization of POIs, as opposed to increasing model performance on SCA-datasets (or aiding in its design). Hence in this chapter, we aim to determine whether a relationship can be established between the areas of the input contributing the most toward class probabilities and hyper-parameter values; and seek to exploit this information for generating high-performing CNNs for side-channel attacks.

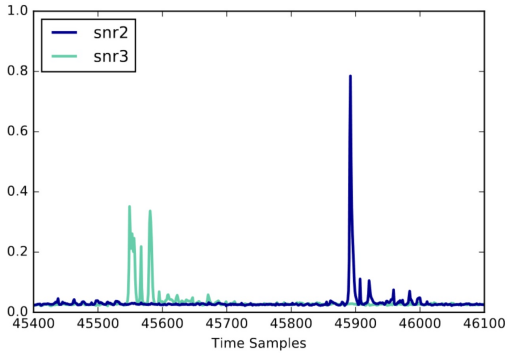
5.2 Exploitable trace properties

We use the reduced traces for the ASCAD [3] database. Each of the traces within this database contains 700 samples corresponding to the interval $[45400..46100]$ of the original traces. The database contains $50,000$ profiling and $10,000$ attack traces. The intermediate values for this interval that show leakage according to the SNR are given in table 5.1.

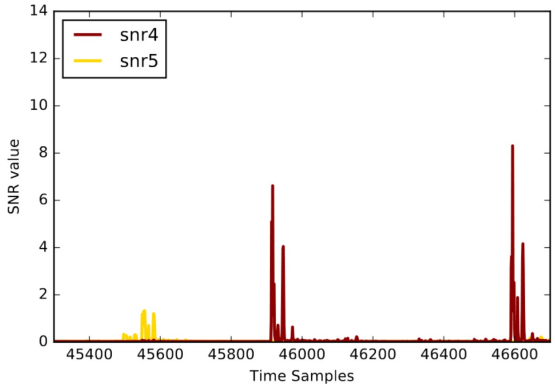
Name	Type	Definition of the target variable Z
snr2	masked sbox output	$sbox(p[3] \oplus k[3]) \oplus r_{out}$
snr3	common sbox output mask	r_{out}
snr4	masked sbox output in linear parts	$sbox(p[3] \oplus k[3]) \oplus r[3]$
snr5	sbox output mask in linear parts	$r[3]$

Table 5.1: SNR for different intermediate values given in [3].

The corresponding SNR values are given in figure 5.1.



(a) snr2=sbox$(p[3] \oplus k[3]) \oplus r_{out}$, snr3=$r_{out}$.



(b) snr4=sbox$(p[3] \oplus k[3]) \oplus r[3]$, snr5=$r[3]$.

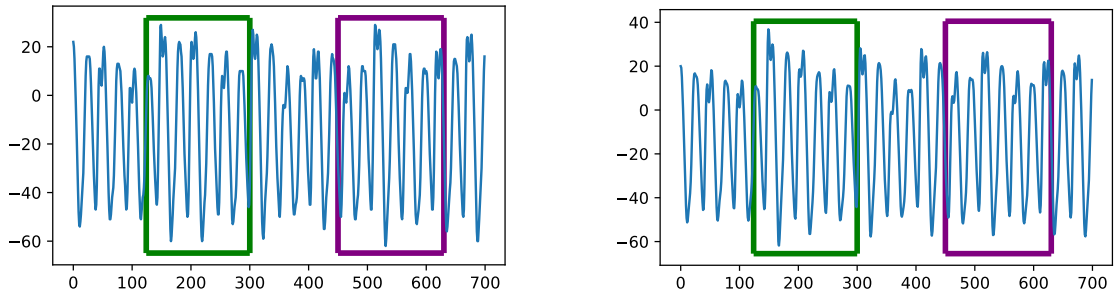
Figure 5.1: SNR for the processing of intermediate values present in the interval [45400..46100] [3].

When inspecting figure 5.1 we observe that the SNR detects high leakage in the interval $[45530..45700]$ for the intermediate value r_{out} and high leakage in the interval $[45850..46030]$ for the intermediate value $(sbox(p[3] \oplus k[3]) \oplus r_{out})$. We also observe leakage in the interval $[45500..45600]$ for intermediate value $r[3]$ and leakage in the intervals $[45900..46000]$ and $[46550..47550]$ for the intermediate value

$(sbox(p[3] \oplus k[3]) \oplus r[3])$. We insist that the interval $[46550..47550]$ is not present in the reduced traces and is mentioned for completeness rather than its relevance for the reduced traces.

The attack on the ASCAD database is set up to target the third byte of the encryption key where corresponding labels in this database contain the value of $z = (sbox(p[3] \otimes k[3]) \otimes rout)$. Because of this, it could stand to argue that any particular model exploiting the SNR leakage for this particular target variable would focus on the areas within these traces, where $snr2$ and $snr3$ have relatively high values.

An example of a single trace taken from the interval $[45400..46100]$ is given in figure 5.2a and the average of the $50,000$ profiling traces is given in figure 5.2b.



(a) single profiling trace.

(b) average profiling trace.

Figure 5.2: Traces from ASCAD $N = 0$.

As we can see from the profiling traces depicted in figure 5.2, we do not observe any particular pattern concerning the locations where leakage occurs according to the SNR from figure 5.1a. However, if we were to assume that this is indeed the area that the model should devote more attention to, as opposed to other areas of the signal, we could adjust the model in such a way so that the saliency map has high values for that specific part of the signal (we can verify this by plotting the saliency map of that model).

For the attack on the ASCAD database, we use the model initially introduced in [59] to attack the $ASCAD$ dataset with a desynchronization counter-measure of $N = 50$. However, in this scenario, we use it to attack the synchronized ($N = 0$) traces. The performance of this model on this dataset containing synchronized traces can be observed in figure 5.3.

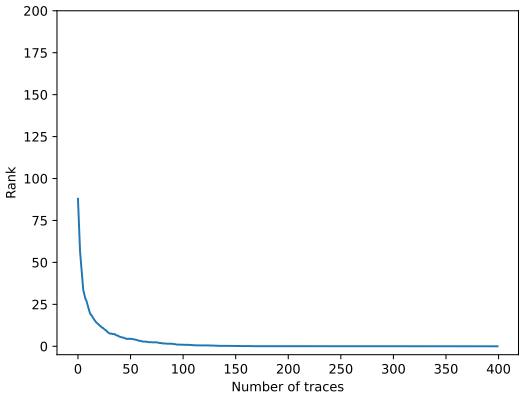
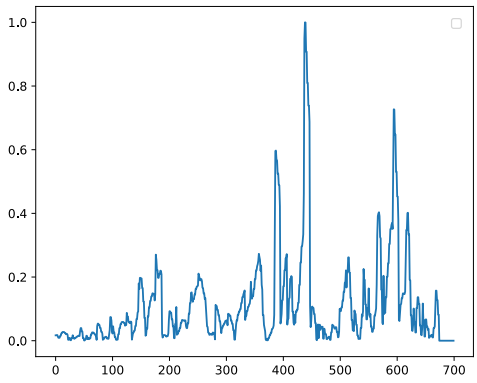
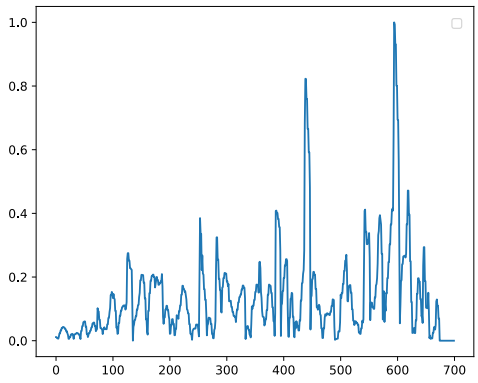


Figure 5.3: Performance on ASCAD $N = 0$, accuracy = 0.009700 and $N_{T_{GE}} = 266$.

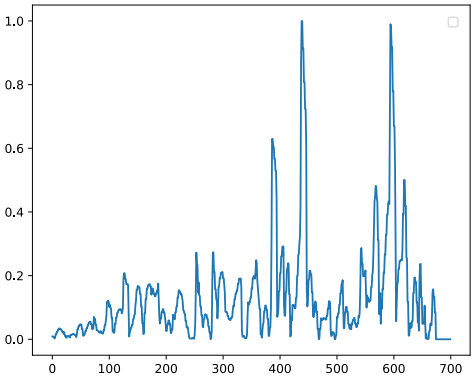
As we can see in figure 5.3, the model performs well on $ASCAD$ $N = 0$, and hence we are interested to see what areas of the input trace attribute more value to when the target variable is correctly classified. We do this by calculating and inspecting the model's vanilla saliency map, which can be observed in figure 5.4.



(a) saliency map single trace.



(b) Average saliency map over profiling traces.



(c) Average saliency map over attack traces.

Figure 5.4: Saliency map obtained for the model with the performance depicted in figure 5.3: Here we see that the areas two highest peaks are in the same vicinity of $snr2$ as opposed to that of $snr3$, and that there is also a significant amount of saliency attributed to areas where the singal-to-noise ratio does not detect any leakage.

When computing the saliency map, we see the significant peaks in the intervals $[124, 300]$, where the SNR shows leakage for $(p[3] \oplus k[3]) \oplus r_{out}$, and $[450..630]$ (where SNR shows leakage for r_{out}) which explains the model's performance.

We also see several peaks spread across the signal where the SNR shows only low leakage. These peaks could imply that there is information in these areas used by the model that is relevant for a proper decision-making process and that the SNR fails to detect the importance of the leakage in these areas. Another possibility is that the information in these areas is irrelevant concerning the actual leakage and that the fault lies in the models decision making process. If we assume that the peaks derived from the SNR are the areas to which the model should assign more significant weight and higher saliency, we would like to adjust the model's hyperparameters so that the convolutional filter assigns high values to these specific areas of the signal.

Before using the hyper-parameters to adjust the model's saliency map, we would like to test our theory by adjusting the traces so that only the areas deemed relevant by the SNR depicted in figure 5.1a remain. We do this by multiplying the signal trace with a vector places zeros at the areas where no relevant leakage was depicted. The calculation vector and the resulting trace is given in figure 5.5.

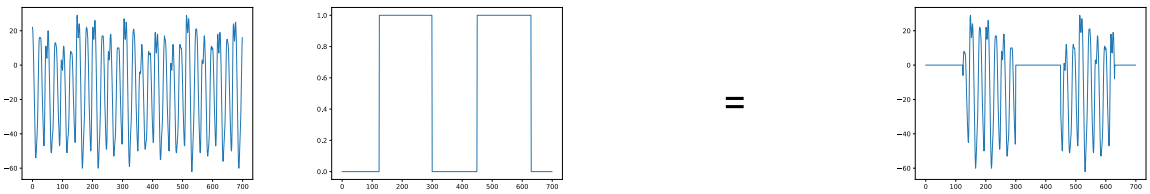


Figure 5.5: Adjusting the trace to eliminate areas deemed irrelevant by SNR depicted in 5.1a. Here we only keep the trace areas where the values for the snr2 and snr3 measurements are significantly higher.

When the model makes use of the altered traces (as depicted in figure 5.5), we observe that the model continues to perform well in terms of its $N_{T_{GE}}$, and if we inspect the accuracy at each epoch, we see that no over-fitting occurs. The resulting saliency maps for the altered traces are shown in figure 5.10.

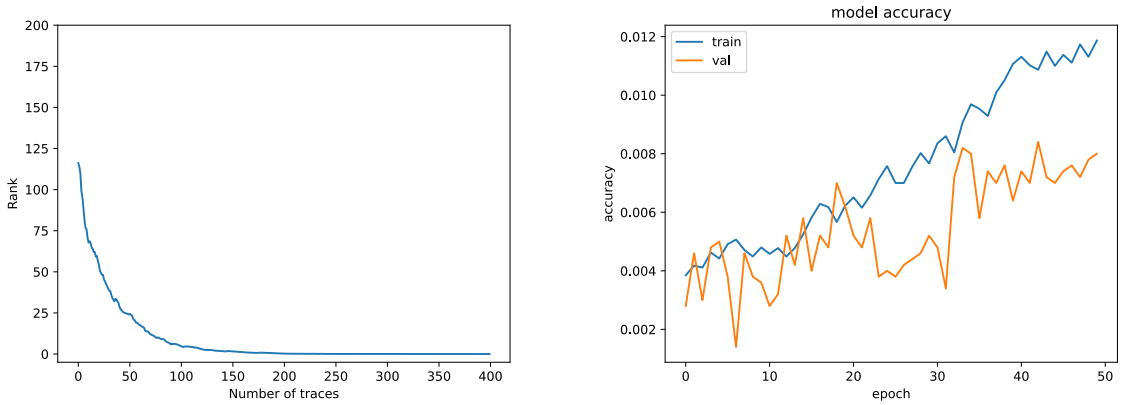
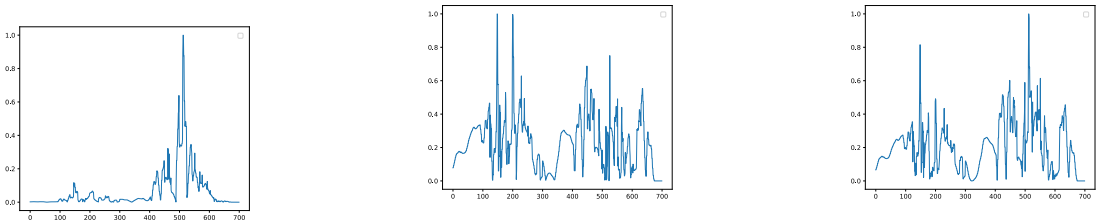


Figure 5.6: Model performance on altered traces (in the form of those presented in figure 5.5) of $ASCAD\ N = 0$, accuracy = 0.006800 and $N_{T_{GE}} = 253$. Here we see that we can still retrieve the correct key even though we use the altered traces.

As we can see in figure 5.10 the saliency maps continue to use the areas that have been set to zeros in the altered traces. This phenomenon may have to do with the $bias$ trainable parameter, which can turn zero values into non-zero values, and due to the pooling and convolution operations that use zero-valued neighbors.



(a) saliency map single trace.

(b) Average saliency map over profiling traces.

(c) Average saliency map over attack traces.

Figure 5.7: Saliency map obtained with the model-performance depicted in figure 5.6 on the altered traces (in the form figure 5.5: here we see that even tough parts of the original trace have been eliminated there are instances where high saliency is attributed to these eliminated parts. We also see that the average saliency map over the profiling traces is fairly identical to that of the attack traces with the exception of two peaks in the area of $snr3$ and one peak $snr2$), we also observe relativily high variance when inspecting the saliency map for individual traces.

We also notice in figure 5.7b that there is a high saliency just outside of the area deemed relevant by the SNR; this would suggest that although the model performs relatively well on this dataset (in terms of its $N_{T_{GE}}$), this model still has significant room for improvement.

When reducing the signal even further by omitting the area deemed relevant by the SNR in terms of r_{out}, we observe that the model does not converge to a guessing entropy of 0 as depicted in figure 5.8. Furthermore, if we inspect the accuracy for each epoch for the validation set, we also see that the model fails to learn a pattern w.r.t. the labels.

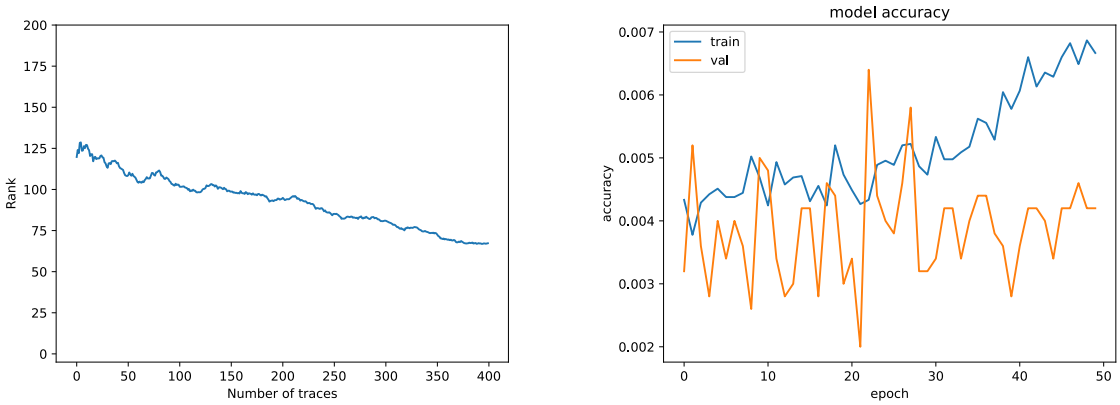


Figure 5.8: Model-performance on traces where the leakage detected by the SNR w.r.t. $(p[3] \oplus k[3]) \oplus r_{out}$ has been removed) and only the high leakage for r_{out} is kept (as displayed in figure 5.9.

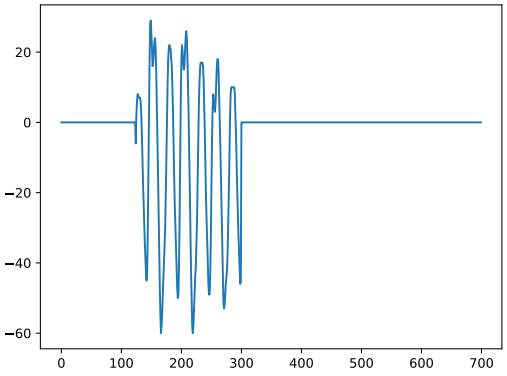
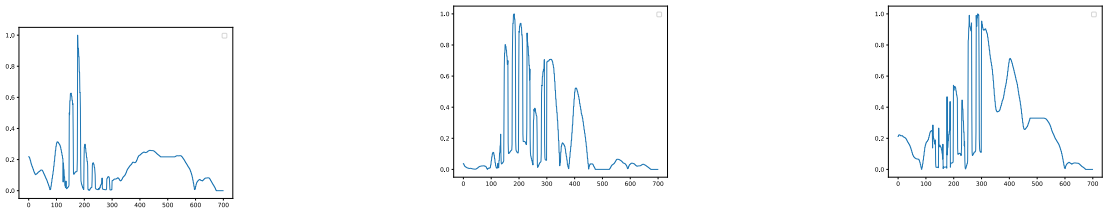


Figure 5.9: trace were only the high leakage detected by the SNR w.r.t. r_{out} is kept.



(a) saliency-map single trace.

(b) Average saliency map over profiling traces.

(c) Average saliency map over attack traces.

Figure 5.10: Saliency map obtained for the performance depicted in figure 5.8 on the altered traces (in the form figure 5.9).

Interestingly when we only remove the higher SNR values for the intermediate value sbox$(p[3] \oplus k[3]) \oplus r_{out}$ as depicted in figure 5.11, we see that the model is still able to use some of the leakages that have extremely low SNR, however not enough to retrieve the correct AES-key. This observation is a testament that the SNR's lower leakages may still serve as a valuable information source for the

feature extraction part of the CNN model. When we combine this leakage with the leakages for $rout$ we see that the model shows strong convergence in the beginning; however, because the most critical information concerning $k[3]$ has been removed, the line runs parallel to the x-axes after reaching a guessing entropy of 25. The strong convergence shows how more minor leakages within a trace can be highly beneficial for the feature extraction process, as the performance in figure 5.11 drastically improves on that of figure 5.8. This particular observation also draws attention to the fact that when a model attributes high saliency to lower leakages, this may not necessarily indicate a fault in the architecture, as these areas can still contribute toward the convergence of a model.

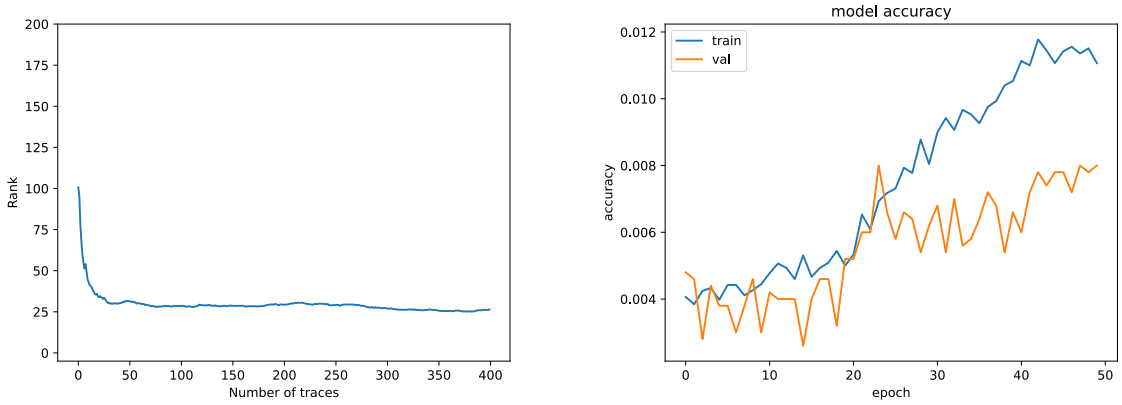


Figure 5.11: Model-performance on traces where the high leakage of the intermediate value sbox$(p[3] \oplus k[3]) \oplus r_{out}$ has been removed), and only r_{out} along with lower leakages (from sbox$(p[3] \oplus k[3]) \oplus r_{out}$) are kept (as displayed in figure 5.12. This figure shows that the model is able to extract information from the (significantly) lower leakages as the model does show some form of convergence as opposed to figure 5.8.

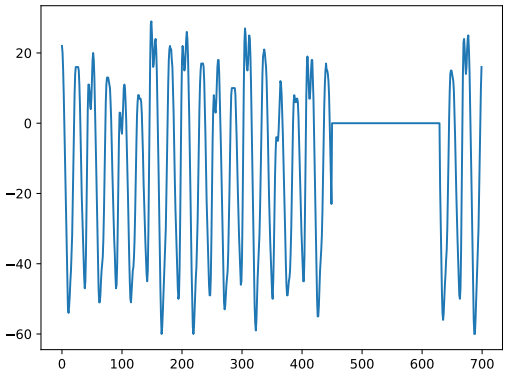


Figure 5.12: trace were only the high SNR-leakage of the intermediate value sbox$(p[3] \oplus k[3]) \oplus r_{out}$ is removed. The high SNR-leakage of r_{out} is kept along with the (extremely) low SNR-leakages related to sbox$(p[3] \oplus k[3]) \oplus r_{out}$ (which may contain a high amount of background noise).

From this section, we can conclude that the SNR can be used to approximate the areas where a deep-learning model can find sufficient for the AES-key retrieval with a short amount of traces. In the following section, we use these findings to adjust model hyper-parameters so that the locations with high SNR are given high attribution concerning the vanilla saliency map.

5.3 Tuning and assessing convolutional layers

We use the model depicted in figure 5.13 for the experiments in this section. We use this model because it only has one convolutional layer and has and requires more than 300 traces to reach a guessing entropy of 0 (and hence has possible room for improvement). The fact that this model has a single convolutional layer allows us to inspect this single convolutional layer first as we move toward deeper (e.g., more convolutional layers) models. We also note that based on this model's validation accuracy, we can observe that it is still learning as the validation accuracy is still on the rise at 50 epochs, and hence it is possible that the model will have better performance if we train the model for more epochs. To account for undesired randomness, experiments in this section have been repeated at least ten times, and their results can be observed in Appendix B. Various visualization methods are commonly used for convolutional layers to depict their effect on the input. In this chapter, we will use Gradient-based class activation maps [45] (Grad Cams) and saliency maps to assess the model's ability to detect the leakages indicated by the SNR.

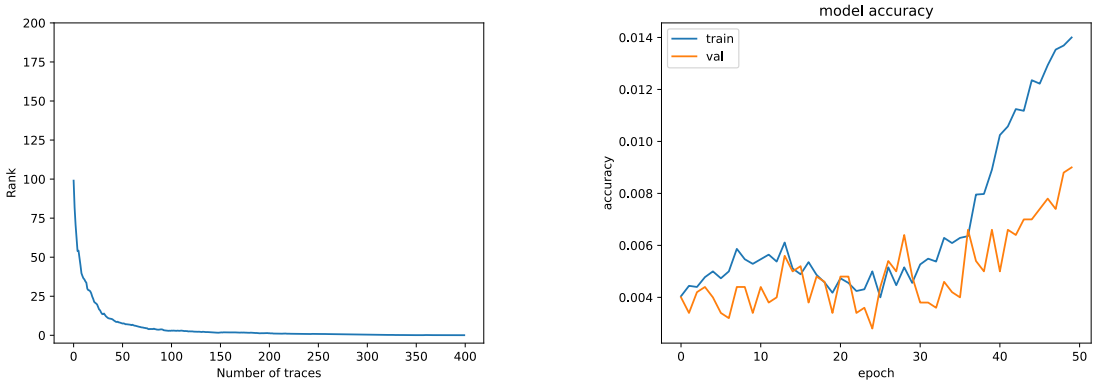


Figure 5.13: Performance of the unaltered model used in this section.

Unlike saliency maps that look at the relationship between the final class prediction and an input trace, Grad-CAMs aims to show the relationship between the output of the last convolutional layer and a class prediction. Hence in our case, we could use a grad-cam to inspect the relationship between the prediction of the correct label and the output of our last convolutional layer. The reason for inspecting the gradient of a particular class only for the last convolutional layer (as opposed to previous convolutional layers) is because the last convolutional layer captures high-level features which make up the final prediction, and hence are more relevant w.r.t what area is actually used to make the prediction (as opposed to earlier convolutional layers that extract local features that may or may not be used in the final prediction of a class).

The average grad-cam for the model depicted in figure 5.13 is depicted in figure 5.14.

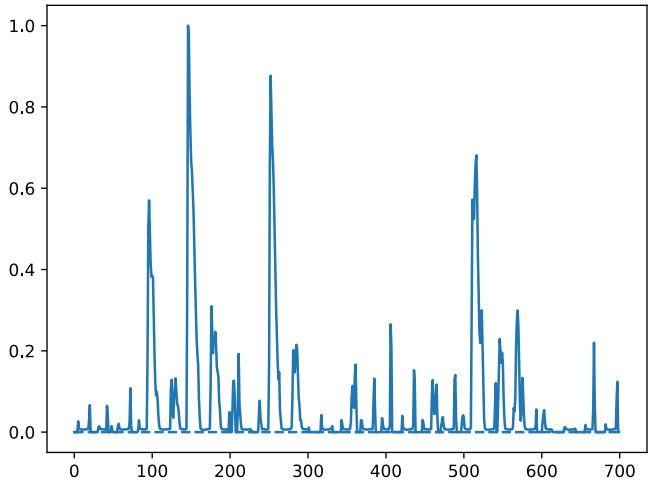


Figure 5.14: Average Grad-CAM over profiling traces. Contrary to the saliency map, the GRAD CAM only shows the saliency that is attributed via the last (in this case, there is only one) convolutional layer. Here the GRAD CAM looks similar to the saliency map as they both attribute higher saliency to the areas that fall within the higher SNR leakage, indicating that feature extraction of the model is successful.

As we can see from figure 5.14 the model does indeed use features that are present in the $[124..300]$ and $[400..630]$ ranges. This observation could be one of the reasons why this model performs relatively well with only one convolutional layer. The saliency map for this model is given in figure 5.15.

When inspecting the saliency map, we observe differences and similarities with the Grad-CAM peaks. The differences can be attributed to the fact that the grad-cam only shows the peak information for the convolutional part of the model, whereas the saliency map takes the entire model into account (convolutional-part as well as the classification part). Hence, using both Grad-CAMs and saliency maps, we can identify which part of the model is (or not) using the features extracted from the SNR-detected leakage.

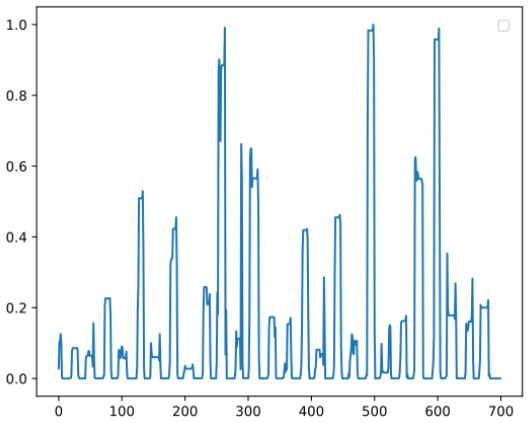


Figure 5.15: Average saliency-map over profiling traces. Here we see that the saliency map is reasonably similar to the GRAD CAM, with some minor differences. The similarities indicate that the classification part of the model makes good use of the features extracted by the convolutional part in the decision-making process. In contrast, the differences indicate that either the convolutional part or the classification part of the model is faulty. The extent of the differences indicates the magnitude of the faults.

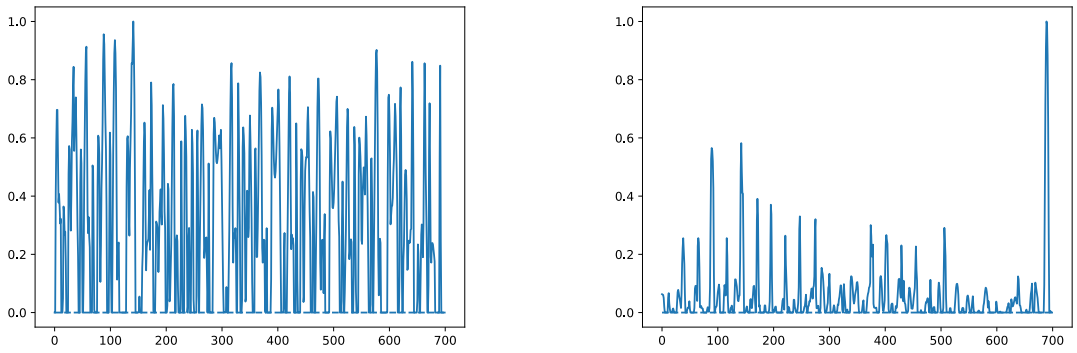
The convolutional layer that the model depicted in figure 5.13 uses has three relevant hyperparameters: filter size, amount of filters, and stride. Each of these three hyper-parameters influences the structure of the average grad-cam.

5.3.1 Filter size and pooling size

When selecting a filter size, it stands to argue that it should be small enough to accurately focus on relevant information without noise from neighboring samples where there is no exploitable information. When inspecting the SNR values from figure 5.1a, we would like to extract features from the ranges $[124…300]$ and $[450…630]$ and not from background noise (although this can sometimes be beneficial as was depicted in figure 5.11). However, the question of how to precisely tune this specific hyper-parameter is still not answered by the GRAD CAM. Our first intuition was that the convolutional filter could produce good results when a size is selected that is less than or equal to the subset of the trace where SNR values are relatively high. However, even though this particular statement may be true for many occasions, it does not cancel out the usage of larger filter sizes. We did several experiments with high to extremely-high filter sizes and again found that we can obtain similar (and often also better) results compared to models using small filter sizes (which is in line with some of the observations made in chapter 4).

When selecting the filter size, Zaid et al. introduce a concept called entanglement, where Zaid *et al.* argues that the larger the filter, the more spread out the relevant information is in the convoluted samples. In his work, Zaid *et al.* proceeds to argue that this may lead to bad performance and labels the phenomenon of two convoluted samples sharing the same relevant information as ”entanglement” with the argument that this might lead to a performance decrease. Even though a larger filter size may be able to cause the spreading of relevant information, we found no evidence of this leading to a performance decrease (in terms of $N_{T_{GE}}$). Decreases in performance are highly dependent on other hyperparameter combinations.

When inspecting the Grad-Cam for different filter sizes, we found that this assumption is not valid for many examples, as shown in figure 5.16.



(a) filter size 200.

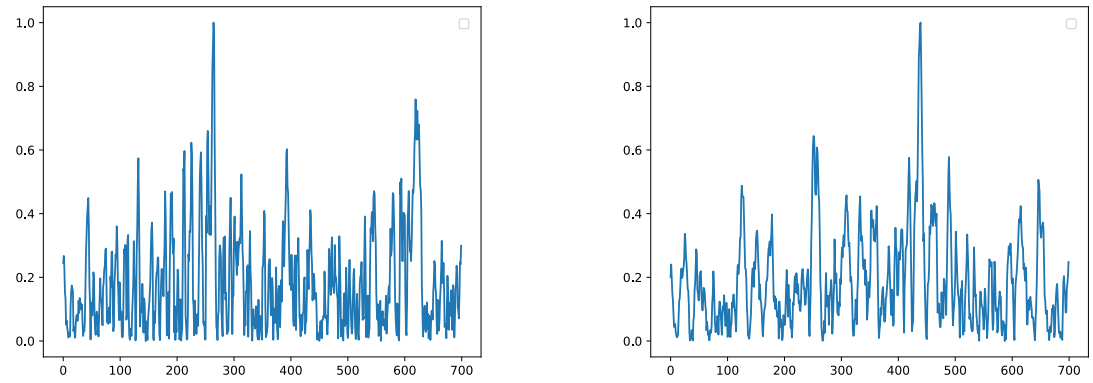
(b) filter size 400.

Figure 5.16: Grad-CAM for filter sizes 200 and 400.

When using larger filters, we notice that even though the information is more spread out compared to the usage of smaller filters (which is also not always the case), we can still detect POIs in line with SNR values from figure 5.1a. This observation can be attributed to the fact that global features (where more neighbors are taken into account) can often still accurately represent exploitable leakage. We can observe this in figure 5.17 where even though we use a large filter size to extract features, the classification part (saliency-map) is still able to accurately extract the leakage without significantly affecting the model’s performance. When inspecting the GRAD CAMs in figure 5.16, it may be tempting to say that the convolutional layers do not contribute because of the spread out information; however, this would be an invalid argument because if we remove this (single) convolutional layer, the model does not converge at all. As a matter of fact the model with a filter size of 400 performs slightly better than the model with an original filter size of 5. Furthermore, we also have to point out the difference in the spread-out factor, which is significantly lower with a filter size of 400 as opposed to that of a filter size of 200, even though it is twice its size (as can be observed in figure 5.16).

On a side note, we also notice that the larger the filter size, the more sporadic the GRAD CAM starts to act; we assume this is because we have a high amount of weights which can learn a high number

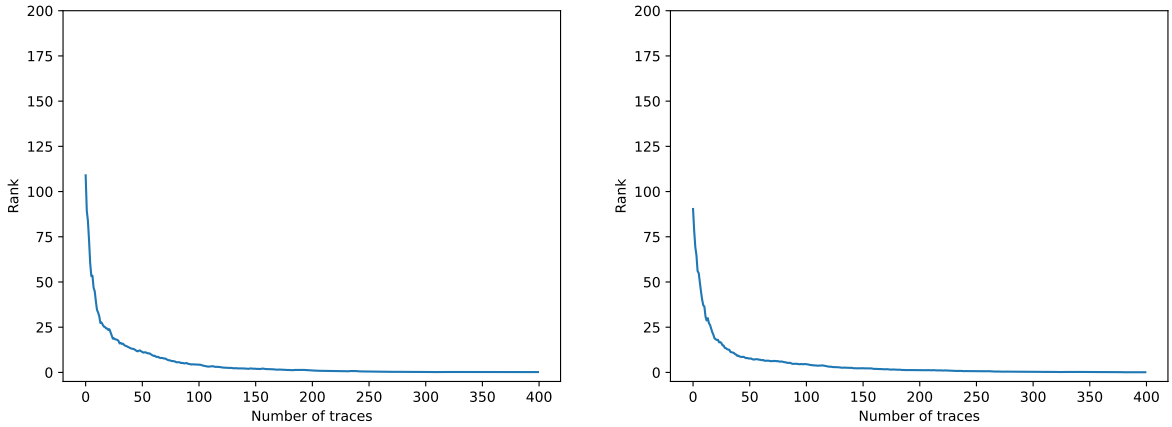
of different combinations of features where many of these combinations can be equally exploited by the classification part of the model. The other factor that needs to be taken into account is that model capacity can not only increase with the addition of layers or neurons but also with the choice of the filter size, which can make a model more prone to overfitting and challenging to train.



(a) filter size 200.

(b) filter size 400.

Figure 5.17: Saliencymap for filter sizes 200 and 400.



(a) filter size 200.

(b) filter size 400.

Figure 5.18: Performance with filter sizes 200 and 400.

Filter size in image recognition is often set to be low in order to increase performance rather than its ability to extract features accurately. Outside of performance the choice is not extremely indicative w.r.t. to feature extraction in the SCA domain (as can be seen in figure 5.17).

Models used in image recognition and natural language processing tend to be quite large (e.g., VGGNet) and contain a high amount of trainable parameters (138 million parameters), where the training time can span multiple weeks for one single model even with high resources. However, models used in the SCA domain tend to be relatively small; hence, the choice of a large filter size is less impactful in terms of performance relative to other domains.

5.3.2 Amount of convolutional layers and amount of filters

The typical idea behind the number of filters and layers in CNN models is to detect local features in the first convolutional layers and higher, more abstract features in the deeper layers of the convolutional

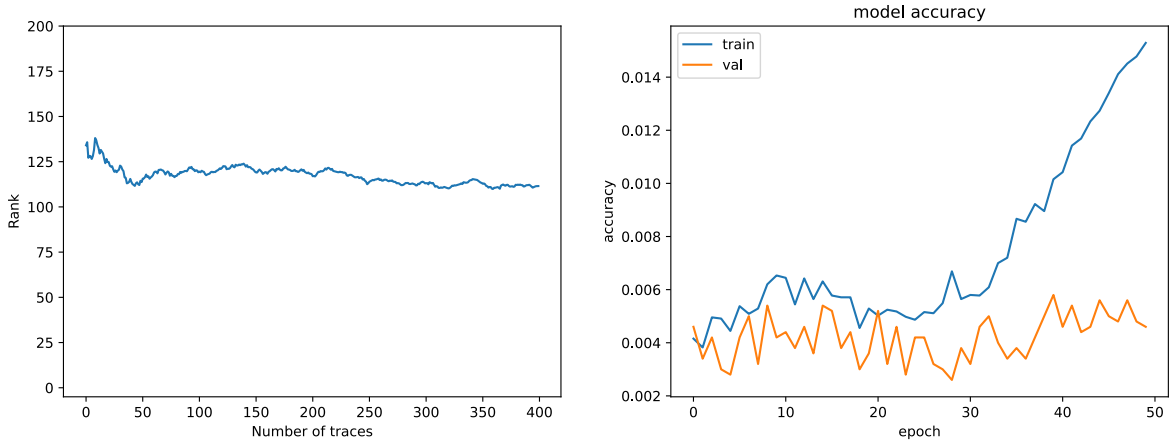
network. In this section, we investigate the effect of an increase in the number of layers in the SCA domain concerning the choice of dataset.

We start by looking at the initial $ASCAD\ N = 0$ dataset, which only contains synchronized traces. When adding layers to our model choice, there is no apparent increase in performance when no desynchronization is present. This observation is expected as the first convolutional layer successfully detects the POI that the SNR identified for this dataset. We can observe the effect of adding more layers to the model in table 5.2, which shows that deeper models do not have a significant impact on this particular scenario. We think that the difference in performance for models containing 8 or more layers in this particular scenario can be attributed to the reduction of input dimension, as opposed to the level of abstraction introduced by adding more layers. This is because the increase is only marginal, and we do not observe any increase when adding the first 6 layers.

Amount of Layers	1	2	3	4	5	6	7	8	9
Ntge	269	279	270	257	280	233	262	189	191

Table 5.2: The effect of adding layers for synchronized traces, when adding more layers we ensure the model's validity by adjusting pooling size accross all layers. Here we only notice a difference in the models containing 8 or 9 layers.

When desynchronization is present, information is spread differently across different traces. When this is the case, we can see that the model does not perform at the same level as it does on the synchronized traces, as shown in figure 5.19. We see that the model does not converge w.r.t. the guessing entropy, and when investigating the training process of the model more closely in figure 5.19a, we also see that it just remembers the traces it has seen before (over-fitting) and fails to learn any specific pattern from the traces and hence is unable to generalize to any new or unseen traces.



(a) Model performance on the attack traces (Guessing entropy).

(b) Evaluation of the model's training process.

Figure 5.19: Performance of the model used for synchronized traces on a dataset with the desynchronization counter-measure.

Looking more closely at this counter-measure, depicted in figure 5.20, we can see why the model fails to detect the leakage indicated by the SNR. This particular counter-measure shifts the intermediate values by suspending the calculation of intermediate values by a random amount of time. The random amount of delay in desynchronization is usually drawn from a specific range between 0 and a max-value m. Because the desynchronization is drawn and applied to each trace individually, the leakages are spread out differently for each trace.

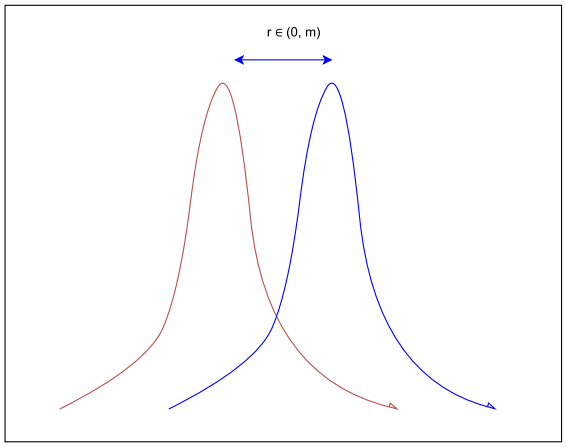


Figure 5.20: desynchronizing traces, where r is randomly drawn from the interval $(0, m)$, for each trace individually, where m is a pre-defined fixed-number that indicates the maximum shift-value each trace can be shifted.

To circumvent this counter-measure, we need to extract features regardless of their location in the input. To achieve this effect, we propose taking advantage of the pooling operation, which can facilitate the detection of spread-out information over the input. Pooling decreases the input size by averaging input features within a specific window size. Hence we can eliminate this effect by ensuring that our network focuses on abstract rather than local features. However, if we set the pooling size window to be too large, we expose the network to be prone to information loss due to including too much irrelevant information (noise) in the output. To eliminate the possibility, we hypothesize that using a small window size for pooling with multiple repetitions can eliminate any level of desynchronization. We assume that a deep enough network will result in an output containing the correct POI by applying a convolution operation (hence extracting features) after every small squeeze, gradually compressing the leakages while extracting the relevant information between two consecutive compressions.

As discussed in section 4.4.2, average pooling is preferred over max-pooling because average pooling retains the information from nearby neighbors and allows the next convolutional layer to decide which compressed value may be of more relevance. In contrast, max-pooling only retains the maximum value compared to its neighbors and throws away the lesser values, consequently making the feature extraction in sequential layers less effective.

To test this hypothesis, we decrease the average pooling to a minimal size of 2 and increase the number of layers in the convolutional part to 7. Note that this does not significantly increase the number of trainable parameters because the filter size is minimal.

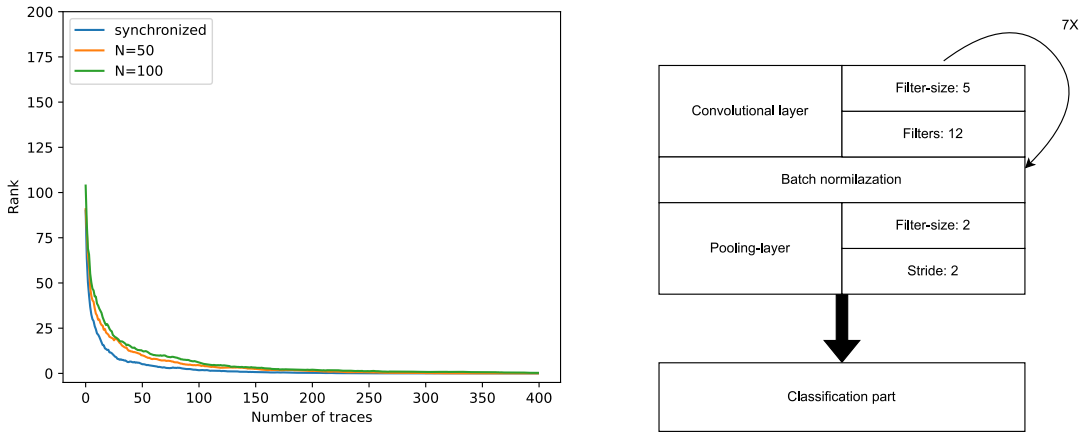


Figure 5.21: performance of deeper models for the extraction of abstract features. Because the filter size is small for each layer, it results in a meager amount of $267,700$ trainable parameters and can be trained within minutes.

In figure 5.21, we can observe the consequence of repeatedly squeezing information while re-extracting features after every squeeze through a follow-up convolutional layer. It shows that the effect of the desynchronization counter-measure is completely eliminated as the model performs similarly on all the datasets regardless of the counter-measures present, which in turn, confirms our hypotheses.

The saliency maps of this model indicate a clear shift in the peaks when desynchronization is introduced which indicates that the model is appropriately identifying the leakages even though the leakage is shifted differently over the time interval for every trace.

Increasing the number of filters also allowed the models to converge better. We assumed this is because adding filters allows for more unique feature maps as each filter learns a different feature, which, in turn, can help detect higher levels of desynchronization or traces where information is spread out and that are generally noisy. The effect of increasing the number of filters seemed to benefit $ASCAD$ $N = 50$ and $ASCAD$ $N = 100$ more than it did $ASCAD$ $N = 0$.

5.4 Discussion

In this chapter, we see that saliency maps and Grad-CAMs can be valuable tools to identify which part of the network is performing poorly and may require adjustment for proper POI detection when combined with other leakage area assessment tools (e.g., SNR). Furthermore, we analyzed the different aspects of the convolutional network that determine the shape of the Grad-CAMs, saliency maps, and its ability to detect leakages in general, such as filter size, pooling size, and amount of convolutional layers. When side-channel-specific counter-measures are present, we see that pooling layers are adequate to bypass these counter-measures; especially when using deeper models, we can see that by using small filter sizes for pooling, we can gradually eliminate any level of desynchronization. We also observe that filter size does not seem to affect model performance (in terms of the $N_{T_{GE}}$) as much as other hyperparameters such as pooling. However, it does seem to affect the number of resources required to train a model because larger filters can cause a significant increase in trainable parameters. However, because models in the side-channel analysis domain are relatively small compared to the models used in other domains such as image recognition, filter-size selection for SCA is less critical for model construction. In the case of SCA, we are faced with the fact that other than the SNR, which can give insights into leakage areas, there are currently not many feature properties that have been properly identified from a human-interpretability standpoint as opposed to, for example, the field of image recognition where one could have edge detection, object shape, etc. One of the main reasons for this is that there is no active attempt to conceal input structure from visual inspection in these other domains. Hence known POI is still the primary trait used in SCA techniques to assess relevant input locations; hence we are currently still focused on the POI identified by other leakage assessment tools (such as SNR) as our main facet when assessing the dataset.

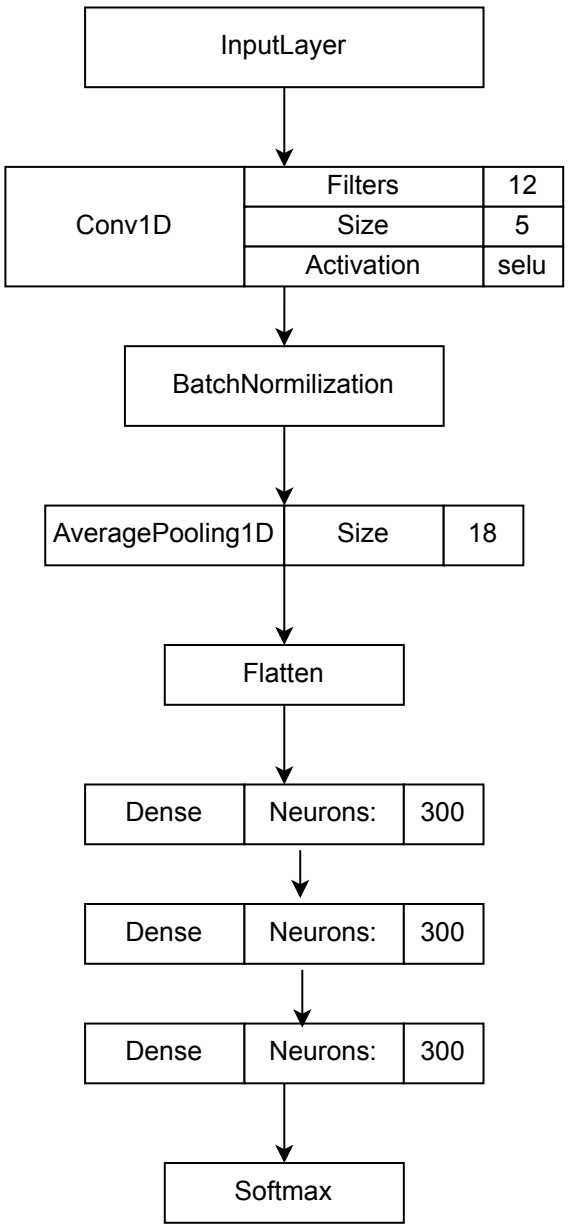


Figure 5.22: unaltered model used on $ASCADN = 0$.

Chapter 6

New methodology

Due to the high amount of radically different models that work well on side-channel datasets and the high combinations of hyper-parameters that work well with our structural constraints, the construction of a methodology for building a CNN is not trivial. The high number of hyper-parameter value combinations that may result in good models indicate that there may be a high number of undiscovered possible methodologies.

However, we did manage to gain usefull insights in chapters 4 and 5; we build on these specific insights in this chapter. We use the architectural constraints from chapter 4 and the hyper-parameter constraints. We argue that attribution methods are currently also a method to constrain things further as it constraints the focus on either the convolutional or the classification part. Here we further introduce constraints by limiting the hyper-parameter selection in a step-by-step fashion, and we attempt to construct a new methodology that can be used to build a convolutional neural network that performs well on SCA datasets and can compete with both state-of-the-art [59] [56] [60] methodologies, as well as other recently presented methodologies [41].

We present the methodology as a sequence of steps to apply to an existing side channel data set. Our methodology can be applied to any unseen side channel data set for CNN construction. We assess the performance of the produced models in terms of guessing entropy and trainable parameters (often referred to as complexity) only for unseen data sets.

6.1 Considerations

The empirical observations from chapters 4 and 5 show that the models that can detect leakage on the ASCAD dataset do not necessarily have to share any close similarities (as a matter of fact, they can be highly different) in terms of their hyperparameter combination when using our architectural constraints. Based on these observations, we developed several intuitions and hypotheses surrounding their construction. These hypotheses were specifically developed for models using our architectural constraints and hyper-parameter ranges.

The first hypothesis we made was that for every two hyper-parameter values drawn from our ranges, there exists at least one set of hyper-parameter values for all other hyper-parameters that allow the convolutional architecture to converge toward a guessing entropy of 1. We tested our hypothesis by varying two hyper-parameters while freezing all other hyper-parameter values. We did this for both convolutional architectures that were already able to exploit the leakage and retrieve cipher-key as well as architectures that were not.

Our first experiment varied the learning rate and mini-batch size values. For this experiment, we did not observe multiple value combinations leading to any converging models. Even in models that were already able to retrieve the cipher-key, the slight variations for these hyperparameters resulted in a drastic decrease in model efficiency. Hence finding a different learning rate did not seem trivial.

We assume this is because the learning rate hyper-parameter can arguably be considered the most significant value for most CNN architectures. This significance is because the learning rate determines how we approach local or global minima of the loss function for our specific architecture. Hence, small or significant changes in the learning rate can either cause the learning algorithm to miss any minima completely or not reach them. On a side note, when taking the mini-batch by itself, most models gave a similar performance as those prior to the adjustments.

For our hypothesis, the convolutional and filter pooling size were the other two hyper-parameters we varied for different models. Contrary to the experiments done for the combination of learning rate and mini-batch, we found multiple combinations of pooling and convolutional filter sizes that gave similar results. For these combinations, the models exhibited the same behavior (sometimes either slightly better or worse) for different convolutional and pooling filter sizes (both individually and collectively) even when we varied them with a large step size. This behavior is similar to what we observed in chapters 4 and 5. We assume this is because the convolutional filter can detect (more or less) the same features with different sizes, and different pooling filter sizes can still allow the detection of the significant shift-invariant features. Which combinations of filter-sizes result in the same features is something that, at least with the current tools, has to be verified empirically. Another issue to point out is that due to the high amount of filters in the convolutional part, it is currently still impractical to analyze each feature individually. We also stress that if we only use the current tools, we still do not know what these features represent, making them even more challenging to analyze. Nevertheless, we know that a convolutional architecture has multiple combinations of convolutional filters possible that will result in the same performance level, which confirms our hypotheses, at least for these two hyper-parameters.

The last two hyper-parameters we varied were the number of dense layers and the number of neurons for dense layers. For this particular case, we found that the classification part had different effects depending on the model's classification part. There was no clear pattern concerning the classification part other than that more dense layers with less (50 or less) performed better than fewer dense layers with more neurons on a few occasions.

Concerning the number of convolutional layers, we saw both in chapters 4 and 5 that it can be beneficial to use deeper models for distinct scenarios. In chapter 5, we saw that the benefit of deeper models is essentially dataset-dependent. We saw that if a dataset has a desynchronization (or random delay) countermeasure, deeper models can be highly beneficial, whereas, for datasets with no such countermeasures, there was almost no benefit to using deeper models. Hence this particular parameter is dataset dependent.

These were the most relevant observations we took into account when building our methodology.

6.2 Methodology

Based on the previous considerations and observations, we constructed the following methodology as a sequence of simple steps that allow us to create an efficient CNN architecture for virtually any SCA dataset.

Because the learning rate can be considered the most critical hyper-parameter and arguably the most difficult to tune: we recommend using an adaptive learning rate. Granted, in our experiments in chapter 4, the one-cycle learning rate was not valuable as we did not find more models, primarily due to the minor computational drawback. However, in the case of our methodology, we will constantly be tweaking the same model, so it would be nice to have as few hyper-parameters as possible to tweak. The adaptive learning rate allows us to focus on other hyper-parameters as it actively attempts to prevent overshooting or slowly approaching local or global minima. For the adaptive learning rate we use 0.001 as the starting factor as experiments in chapter 4 give good results for this setting.

Our methodology distinguishes between datasets with a random-delay countermeasure and those that do not. However, our starting approach will be the same for both cases.

We start with determining the number of convolutional layers. For this step, we choose an arbitrary small starting filter size of 5 (although 2 is also acceptable) because we found no particular evidence of choosing one filter size over the other. The convolutional filter size of 5 is an appropriate starting value as it takes only a few neighboring values into account and can always be adjusted if our methodology requires us to do so. For the pooling filter, we start with the smallest possible value of 2. For this step, we choose a base classification part of 3 dense layers, each containing 100 neurons. We chose this base model because both our experiments as well as previous literature show that dense layers do not require many neurons to detect leakages for SCA datasets, and hence 100 neurons are more than enough capacity to represent the underlying distribution function for SCA datasets.

We keep the number of filters for any convolutional layer fixed at 12, as previous experiments showed that we do not need to increase the filters as the models grow deeper as long as the total amount of filters are sufficiently able to extract relevant features.

First, train the model with one single convolutional layer and then continue training a new model by gradually convolutional layers. Inspect the convergence for each model. At this point, it should be clear how many layers our model would need when using a pooling size of 2. If the traces are significant in length, we recommend starting with a larger pooling size. When a random delay is present, we can decrease the number of convolutional layers needed by gradually increasing the pooling size. If there is no random delay present this step should confirm that only a low number of convolutional layers is needed.

If the traces are significant in length and the dataset contains a random delay, we recommend starting with a larger pooling size. When a random delay is present, we can decrease the number of convolutional layers needed by gradually increasing the pooling size.

At this point, we can start tuning the convolutional filter size; however, before we do so, we first calculate the SNR for the dataset and use attribution methods first to see whether it is the convolutional part or the classification part that needs the most tweaking.

Tweaking the filter size: As we have seen many times throughout this thesis, the filter size can have different values across all layers, and the architecture can still achieve the same performance level. In this methodology, we decided that this specific hyper-parameter can be adjusted empirically (using any method) as there is no specific preference for the convolutional filter. We start with a small convolutional filter and can gradually increase the size and observe the effect this has on the performance. We note that larger filter sizes can detect more global features, and smaller filter sizes can detect more local features (where there is also a significant overlap in the features both can detect). This note means that for datasets with a random delay, it may sometimes be beneficial to choose a larger filter; however, it is currently still something that has to be decided empirically.

Tweaking the classification part: As discussed in chapter 4, we did not notice any specific preference for the classification part. However, in almost all cases, any classification part that employs neurons in the 100 range seemed to have sufficient capacity for SCA datasets. In order to further improve the classification, we propose to gradually explore adding more dense layers with fewer neurons, as this was shown in [52] [53] to be more beneficial in certain scenarios as opposed to adding more neurons to the existing hidden layers.

Although not necessary for our methodology, we mention the optional step of optimizing the number of trainable parameters. We do not think this step is necessary because of the relative size of models used in the side-channel domain and because the step is relatively trivial. We also do not actively perform this step in our methodology assessment.

6.3 Results

6.3.1 DPA V4

This dataset has a masking countermeasure and does not incorporate any form of random delay; hence we know beforehand that we do not require high pooling or a high amount of convolutional layers. We observe in this step of our methodology that one layer is sufficient for obtaining state-of-the-art performance (in terms of methodology). The results of these experiments can be observed in figure 6.1.

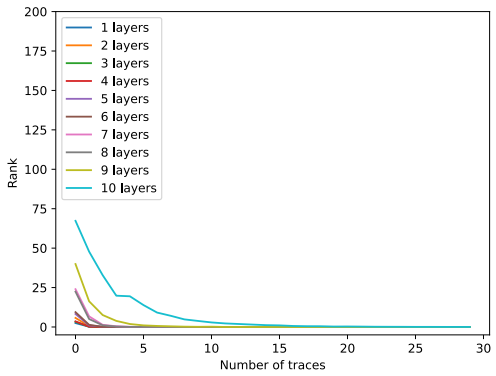


Figure 6.1: Applying our methodology to the DPA v4 dataset. Here we see that this dataset is more trivial than others as it only has a first order counter-measure, were the traces contain almost no noise.

We also observe the ill effect that adding more layers can have on a dataset without desynchronization; the more layers are added after the second one, the less efficient the model performs on this specific dataset. For models with 1 to 4 convolutional layers, we observe similar performance, whereas starting from models with 5 layers and onward, we notice a tremendous drop in efficiency. We obtain an $NTge$ of 1, and further improvement is not needed as this is already state-of-the-art performance. Although not needed by our methodology, we can also decrease the number of parameters required by reducing the number of filters from 12 to 1, without any signs of performance decrease. We assume this is because this particular dataset has a known mask value; hence, feature extraction is not a necessity. The model selected through this methodology is depicted in figure C.1.

6.3.2 AES RD

This dataset has a random delay countermeasure; hence we increase the number of convolutional layers, and after each one, we add a pooling layer with a small pooling size to eliminate this countermeasure. We tested our methodology on this dataset and found that we get state-of-the-art results starting from layer 8, depending on the pooling filter size used. We stress that the amount of layers does not affect the number of parameters present in the model because we are using small convolutional filter sizes while decreasing the input size. We can decrease the number of layers by gradually increasing the pooling size until no further improvement is observed. The results of these experiments can be observed in figure 6.2.

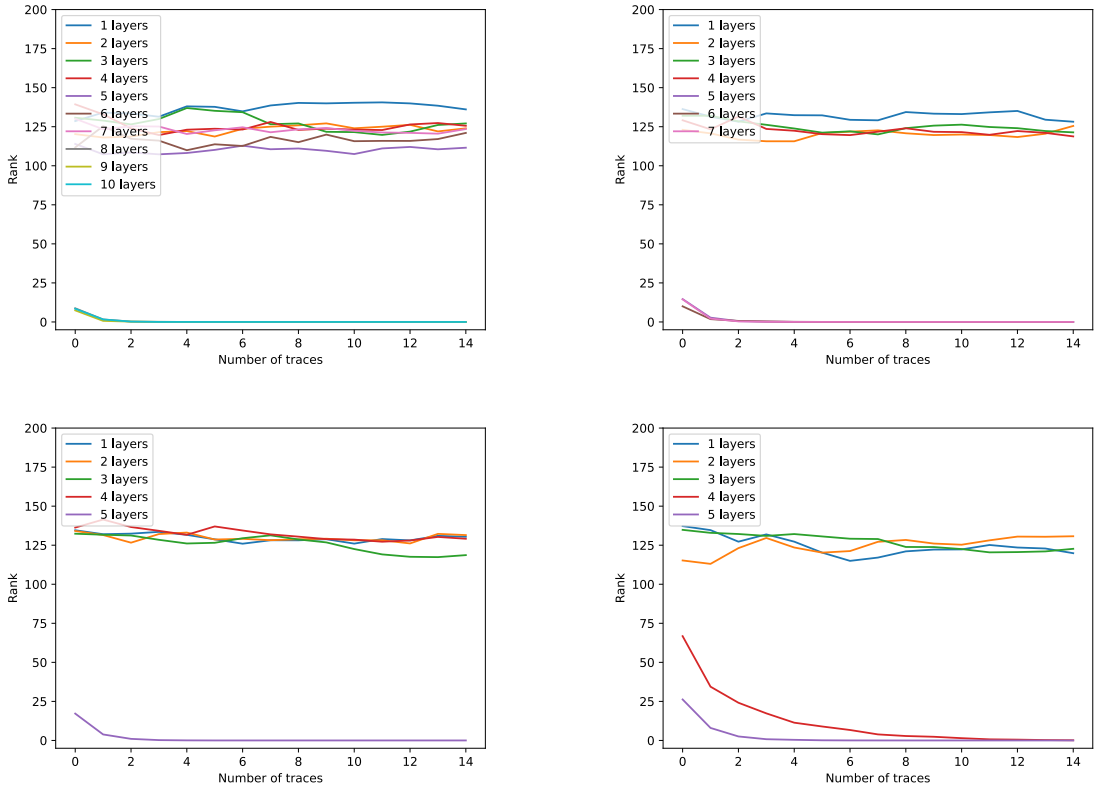


Figure 6.2: Performance obtained trough our methodology on the AES RD dataset which contains a random delay countermeasure.

From these results, we can see that a top-bottom approach might be better than a bottom-up one as we can start with high-performing architectures until we come across an architecture that does not perform well. Also, note that the models have a low amount of trainable parameters because we keep the number of filters low. From our previous experiments, we know that we do not need to follow the standard paradigm of increasing the number of filters as the network grows deeper; hence we use the same amount of filters throughout all convolutional layers for these experiments, which makes it easier to control the number of parameters our models exhibit. We observe how the sequential pooling can completely eliminate the desynchronization counter-measure and how deeper models reach good results where the more shallow models struggle. We find multiple models to select from when we use our methodology. We also observe that as we gradually increase the pooling size (resulting in fewer layers), we reach a point where the pooling size is too large. For this dataset, the model selected through our methodology is depicted in figure C.2.

6.3.3 AES HD

The AES HD dataset sets itself apart from other SCA datasets as it contains a property that this thesis has not yet encountered, namely, the property of extremely noisy features/traces. It is also considered one of the more difficult public datasets in the SCA domain. We initially find no converging models when we apply our methodology for this specific dataset during the first stage (as depicted in figure 6.3). However, as we gradually increase the pooling-filter size (resulting in fewer layers), we detect that the model consisting of only two layers starts to show more convergence relative to the other architectures (figure 6.3); however, we are still unable to retrieve the cipher key. Here things become interesting as it gives us an adequate opportunity

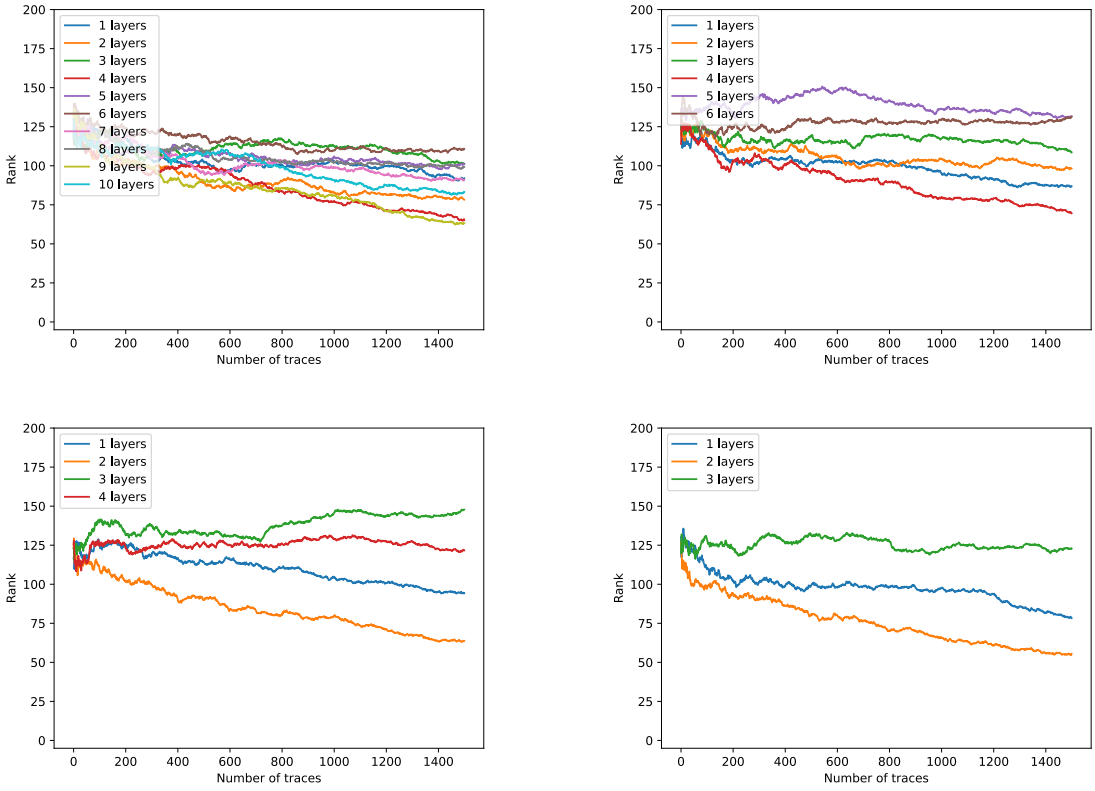


Figure 6.3: AES HD: Initially during the first stage of our methodology no models are found for the AES HD dataset. During the first stage of our methodology, we gradually increase the pooling size. When reaching a pooling-filter size of 5, the methodology shows that selecting an architecture with only two convolutional layers may be beneficial.

to use attribution methods. When using the GRAD-CAM and saliency map, we notice that the convolutional part can detect the leakage adequately and that the fault lies in the classification part, which requires more tuning.

Figure 6.4: AES HD: To figure out which part of the network needs to be tuned we use the attribution methods: GRAD CAM and saliency map. The GRAD CAM in this scenario is pulled together over an input size of different dimensions than the original trace because we use the output of the last convolutional layer at which the input has already been pooled several times; however, if we super impose the GRAD CAM over the original input trace dimensions, we see that it is in-line with the SNR value for this specific dataset (as displayed in figure 6.5).

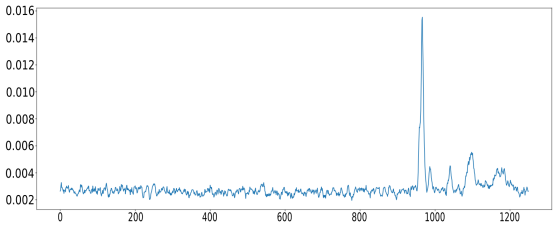


Figure 6.5: SNR for the AES HD dataset.

After figuring out which part of the model most likely needs adjusting, we move toward the step of adjusting the classification part (which consisted of a base of 3 dense layers with 100 neurons). From our expirements in chapter 4, we know that often times more dense layers with less neurons can give better performance, which is in-line with [18]. Hence, we decide to use 10 dense layers instead of 3 and start of with 10 neurons per layer. Here we notice a great improvement but not quite enough to retrieve cypher key, we gradually increase the amount of neurons per layer by a factor of 3 and find that 30 gives extremely good results. Note that for this part of the methodology any method can be used as we are only dealing with two hyper-parameters: amount of dense layers, and amount of neurons.

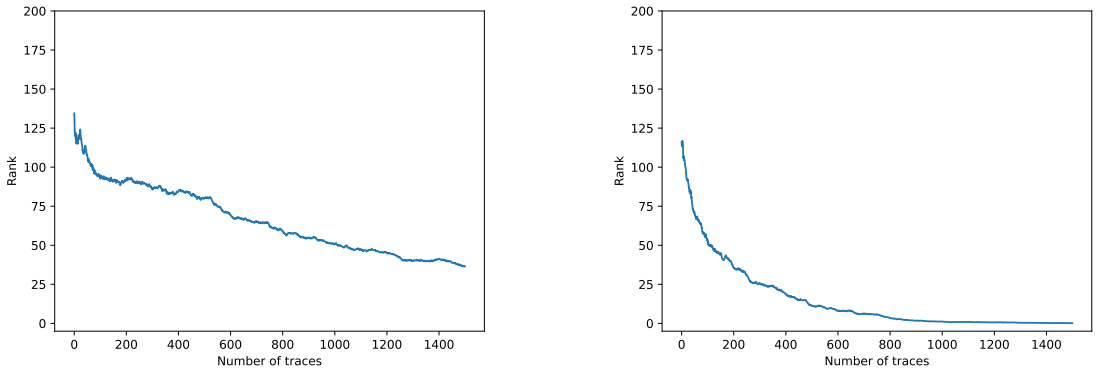


Figure 6.6: AES HD: There is a large improvement going from a shallow classification part toward a deeper classification part.

To tune this part of the architecture, we use our observation from the previous section and use more layers with fewer neurons. As we can see from figure 6.6, this significantly increased our model's efficiency, and we can now retrieve the cipher-key with $N_{t_{ge}} = 1000$, which is state-of-the-art performance (in terms of methodolgy). As we had previously observed, the non-trivial datasets work well with larger minibatch sizes; hence we also use a larger minibatch size in this scenario.

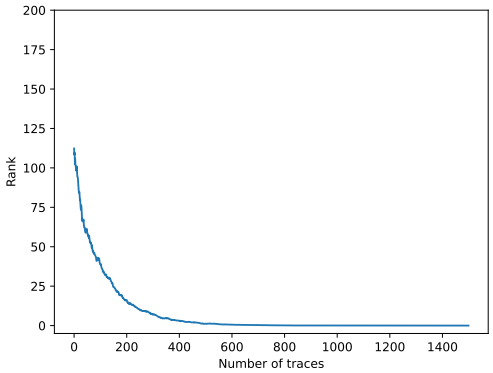


Figure 6.7: Doubling the number of filters for this model resulted in a further performance increase.

Altough it is not needed we found that the model converges even better when doubling the amount of filters from 12 to 24 as depicted in figure 6.7. The final model for this particular scenario is depicted in figure C.3.

6.3.4 Comparison with random search

In order to further assess the effectiveness of our methodology we did a comparison with random search, where we ran a search algorithm for 7 days on an HPC cluster that employed a 1080 TI video-card. The results show that random search can be used on more trivial datasets such as AES RD, however not on AES HD as the algorithm did not find any converging model for this particular dataset. In order to make a fair comparison we only did a random search using our architectural constraints and hyper-parameter ranges instead of a fully random search (as this is more likely to fail).

For $AES\quad RD$ we found 125 models that converged toward a guessing entropy of 1. Out of these models we found 30 where able to reach this with only 2 traces. Similarly we found multiple models for DPA V4 as it is more trivial. This shows that random search is a viable option for these specific datasets; however, we argue that this may not always work for an attacker because of time constraints. We also stress that random search may often require considerable resources, espacially if the dataset is non-trivial.

For $AES\quad HD$ we did not find any models that were able to converge toward a guessing entropy of 1. Furthermore we also found only few models that were able to pass a guessing entropy of 50, which is an indication of the difficulty of this dataset and also clearly shows the extend of the methodology's efficiency.

We also run random search for the ASCAD $N = 100$ dataset, which is one of the least trivial from the ASCAD collection. For $ASCAD\ N = 100$ we find only 13 models that converge toward a guessing entropy of 1. However, looking more closely at the performance of these models, we see that these models require anywhere between 579 and 898 traces to reach a guessing entropy of 1; hence, for this scenario, the best model requires almost 3 times more traces than the models generated by our methodology.

6.4 Discussion

Note that we cannot use blind Random search or Bayesian optimization as these methods will not produce good results when a dataset is difficult (e.g., contains desynchronized traces or noisy features). Our method differs because we combine what we have learned in chapters 4 and 5. We have learned that the proper assessment and identification of the ranges is more insightful as it shows that different values for the same hyper-parameter can lead to the same performance level. We also learned that as opposed to the suggestions that were given by Zaid et al. regarding the concept of "entanglement", it is virtually non-existent in the field of side-channel analysis when taking kernel size into account. We want to use attribution methods to gain insights into how specific hyper-parameters need to be adjusted based on the data set used; however, as we have seen in chapter 5, this is currently (at the time of writing) not possible with the attribution methods we explored so far. The extent of the use of attribution methods is in the evaluation of a network's ability to detect leakage points, which often can also be misleading because these attribution methods are based on the accuracy metric and not on the guessing-entropy metric (which is preferred in the SCA domain as it was shown in [39]). Hence the black-box method is still an issue with these attribution methods. This is also why no existing methodology currently uses attribution methods and dataset features in constructing a network. These methods have currently only been used to show that the model works and succeeds, attributing high saliency to SNR peaks (as was done by Zaid et al.) and not for the selection of hyper-parameters. Showing that a model succeeds in detecting leakage does not help us determine hyper-parameter values. Combining this with the fact that the assessment of the individual features found by a CNN for SCA datasets can currently only be beneficial if we know what these features mean makes the construction of a methodology for CNNs thus the more challenging. Hence these observations make

it clear that when it comes to selecting hyper-parameters values such as kernel size for a new and unseen dataset, this still has to be done empirically.

The only proper consistency when assessing datasets concerning specific countermeasures and how we can tackle them. If we have a desynchronization countermeasure, we can take this countermeasure by either adjusting the pooling size or gradually pooling with an extremely small pooling size whilst doing our feature extraction in between these pooling operations. Hence our methodology builds on this fact. When there is no such countermeasure (or when it is eliminated by pooling), we are left with the detection of helpful features with many combinations we currently can not select a priori with the current tools at hand. However, we did manage to identify the constraints for the architecture of CNNs that work well with our identified ranges; we build on these specific insights in this chapter. We use the architectural constraints from chapter 4 and the hyper-parameter constraints. We argue that attribution methods are currently also a method to constrain things further as it constraints the focus on either the convolutional or the classification part. Here we further introduce constraints by limiting the hyper-parameter selection in a step-by-step fashion. This construction is based on the observation that multiple values for a single hyper-parameter are possible. Hence we theorize that for every partial combination of hyper-parameters of one part of the network, there exists at least a significantly large set of combinations in the other part of the network that significantly improves the overall performance. Hence for a combination of mini-batch and learning rate selected from our ranges, there exists a combination of convolutional-filter size and pooling size, which is the idea that our methodology builds on.

Assessing deep learning models with attribution methods for the purpose of explain-ability is currently (at the time of writing) still a non-trivial task, as even though we can visualize the parts of the input that had the most influence on the decision-making process of the network, we are currently still not at a point where we can say that the features that were detected by the network represent a certain aspect of the feature trace with regard to their meaning. However, this is not an issue for our methodology, as we still are able to leverage the fact that by depicting the areas of the trace that are most influential according to a certain part of the network and comparing them to SNR values, we are able to gain insights into which parts of the model may require more adjustments. By taking dataset features and properties (e.g. countermeasures) into account we are able to build a methodology that can be applied to any unseen dataset.

The results show that the proposed methodology outperforms any existing methodology for deep learning architectures in the side channel analysis domain concerning the architectures it produces. Even though it is outside of the scope of this thesis, It would be interesting to see how other interpretation ability methods could be applied to improve the methodology concerning other hyper-parameters, particularly those involving the classification part, such as dense layers and amount of neurons. Another interesting research question for future work would be to investigate whether we could leverage the saliency map to guide the training algorithm. A possible way to do this could be by introducing some form of penalty when the saliency map is not in sync with the SNR for the training set or, in the case of reinforcement learning, introducing a reward when the saliency map and SNR values are more aligned. It would also be interesting to see how other automated feature extraction methods such as copy-to-input auto-encoders compare to convolutional layers. The last suggestion is particularly interesting because when using copy-to-input auto-encoders, we can say something about the feature relevance because we can observe whether they are required to reconstruct the input, allowing us to assess them before actually adding an auto-encoder to the network.

Chapter 7

Conclusions

7.1 Research questions

Research question 1

Can we identify structural constraints and hyper-parameter ranges that produce high-performing CNNs and aid in the design and construction of CNNs in the field of side-channel analysis?

Answer: Yes, we can; in chapter 4, we showed that by selecting ranges through research and experimentation, we were able to produce a list of hyper-parameter ranges and architectural constraints, resulting in a helpful design paradigm for the design and construction of CNNs that perform well on side-channel data. When comparing current existing methodologies to our design paradigm, we also observed that this method compares exceptionally well to current existing methodologies. We find several high-performing models within a relatively short time, whereas other methodologies such as reinforcement learning require days to weeks to find models without much guarantee of their reproducibility (due to, for example, covariant shift and undesired randomness). We also see that it outperforms other existing methods such as Bayesian optimization and genetic algorithms in both model robustness and performance of generated models.

Research question 2

Can attribution methods from other domains be used to aid in the design of deep learning models for SCA?

In chapter 5, we showed that attribution methods such as saliency-maps and GRAD-CAMS can be instrumental in the design of CNNs because when combining these methods with leakage assessment tools such as SNR, we can identify the faulty part of the network, which can further aid in the elimination of which hyper-parameters may need adjustment. We also showed that deeper models effectively reduce the effect of common hiding counter-measures such as desynchronization but did not show much performance gains in the absence of such counter-measures. Furthermore, the attribution methods also allowed us to observe the effect of convolutional filter size for the SCA domain, which showed little importance compared to other hyper-parameters such as pooling and amount of filters when considering side channel datasets. We also showed that large filter sizes could work quite well in the SCA domain and because the SCA architectures tend to be relatively small compared to other domains, this is a viable option for constructing and designing CNNs in the side channel analysis domain when necessary.

Research question 3

Can we derive a new methodology for the construction of CNNs that gives good results and is easy to follow?

Yes. In chapter 6, we provided a methodology in the form of a sequence of steps that help construct Convolutional neural networks and can be applied easily to a side-channel dataset. We assessed this methodology by applying these steps to a new and unseen dataset, which resulted in constructing a

convolutional network that performs better than those provided by current state-of-the-art methodologies, making our proposed methodology the new state-of-art.

7.2 Summary of Contributions

- We have provided a combination of suitable hyper-parameter ranges and constraints on the architectural design for CNNs for SCA. This combination is a compelling and feasible approach for the side-channel analysis domain, primarily because side-channel data does not require immense models as opposed to other domains such as image recognition and natural language processing.
- Furthermore, we have shown that combining this strategy with attribution methods such as GRAD-CAMs and saliency maps allows us to further reduce model-design time by identifying the faulty part of the deep learning architecture, which allows us to eliminate further hyper-parameters that may require adjusting.
- We have provided a methodology that is easy to follow and aids in the construction and design of CNNs for the SCA domain. Our suggested methodology is currently the most robust in terms of the number of models it can construct and the reproducibility (robustness) of the performance of generated models, as opposed to other methods such as reinforcement learning and genetic algorithms. As far as we know, at the time of writing, the proposed methodology outperforms all other SCA methodologies and can be considered state-of-the-art in the SCA domain.

7.3 Limitations and Future work

Although we can often identify which part of the network is faulty by using attribution methods, it has not yet shown the ability to give further detail on which hyper-parameter of this faulty needs adjusting. This flaw is especially the case with the classification part of a network. If the classification part is identified as faulty, attribution methods do not show whether neurons need to be added or removed (and from which layer) and if layers need to be added to improve the model. In the case of the convolutional part, this is currently also true; however, through observed characteristics of the dataset (e.g., hiding counter-measures), we can still gain some intuition on which hyper-parameters of the convolutional part may increase performance; however, these present intuitions have not yet completely eliminated the need for empirical experimentation of trial-and-error in order to identify the correct values for these parameters and to obtain an optimal model.

It would be interesting to see how other methods, such as reinforcement learning and genetic algorithms, would perform if we restrain the reward function toward structures that employ our constraints and hyper-parameter ranges and how their efficiency would compare to our methodology. Another interesting research question for future work would be to see if we could somehow introduce a penalty on the cost function of the learning algorithm based on the SNR of the dataset and the saliency map of the model in order to ensure that the saliency map contributes high saliency to the areas that show high leakage and low values to the areas that do not.

Bibliography

[1] Nadhem J. Al Fardan and Kenneth G. Paterson. “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”. In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

[2] Bowen Baker et al. “Designing Neural Network Architectures using Reinforcement Learning”. In: *CoRR* abs/1611.02167 (2016). arXiv: 1611.02167. URL: http://arxiv.org/abs/1611.02167.

[3] Ryad Benadjila et al. “Deep learning for side-channel analysis and introduction to ASCAD database”. In: *Journal of Cryptographic Engineering* 10 (June 2020). DOI: 10.1007/s13389-019-00220-8.

[4] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13.1 (Feb. 2012), pp. 281–305. ISSN: 1532-4435.

[5] Shivam Bhasin et al. “Analysis and Improvements of the DPA Contest v4 Implementation”. In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont. Cham: Springer International Publishing, 2014, pp. 201–218. ISBN: 978-3-319-12060-7.

[6] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. “Convolutional Neural Networks with Data Augmentation against Jitter-Based Countermeasures.” In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference*. Taipei, Taiwan, Sept. 2017. URL: https://hal.archives-ouvertes.fr/hal-01661212.

[7] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 13–28. ISBN: 978-3-540-36400-9.

[8] François Chollet et al. *Keras*. https://keras.io. 2015.

[9] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2015. DOI: 10.48550/ARXIV.1511.07289. URL: https://arxiv.org/abs/1511.07289.

[10] Jean-Sébastien Coron and Ilya Kizhvatov. “An Efficient Method for Random Delay Generation in Embedded Software”. In: *Cryptographic Hardware and Embedded Systems - CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 156–170. ISBN: 978-3-642-04138-9.

[11] Jean-Sébastien Coron and Ilya Kizhvatov. “An Efficient Method for Random Delay Generation in Embedded Software”. In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 156–170. DOI: 10.1007/978-3-642-04138-9_12. URL: https://www.iacr.org/archive/ches2009/57470156/57470156.pdf.

[12] Jean-Sébastien Coron and Ilya Kizhvatov. “Analysis and Improvement of the Random Delay Countermeasure of CHES 2009”. In: *Cryptographic Hardware and Embedded Systems, CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 95–109. ISBN: 978-3-642-15031-9.

[13] Joan Daemen and Vincent Rijmen. “The Design of Rijndael: AES - The Advanced Encryption Standard”. In: 2002.

[14] A. Eiben and Jim Smith. *Introduction To Evolutionary Computing*. Vol. 45. Jan. 2003. ISBN: 978-3-642-07285-7. DOI: 10.1007/978-3-662-05094-1.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. "Deep Neural Network Attribution Methods for Leakage Analysis and Symmetric Key Recovery". In: *Selected Areas in Cryptography – SAC 2019*. Ed. by Kenneth G. Paterson and Douglas Stebila. Cham: Springer International Publishing, 2020, pp. 645–666. ISBN: 978-3-030-38471-5.

[17] Geoffrey E. Hinton. "A Practical Guide to Training Restricted Boltzmann Machines". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 599–619. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_32. URL: https://doi.org/10.1007/978-3-642-35289-8_32.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[19] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456.

[20] Maikel Kerkhof et al. *Focus is Key to Success: A Focal Loss Function for Deep Learning-based Side-channel Analysis*. Cryptology ePrint Archive, Report 2021/1408. https://ia.cr/2021/1408. 2021.

[21] Nitish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: (Sept. 2016).

[22] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836.

[23] Jaehun Kim et al. "Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-channel Analysis". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (May 2019), pp. 148–179. DOI: 10.46586/tches.v2019.i3.148-179.

[24] Günter Klambauer et al. "Self-Normalizing Neural Networks". In: (2017). DOI: 10.48550/ARXIV.1706.02515. URL: https://arxiv.org/abs/1706.02515.

[25] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[26] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[27] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397. ISBN: 3540663479.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[29] Hugo Larochelle et al. "An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation". In: *Proceedings of the 24th International Conference on Machine Learning*. ICML '07. Corvalis, Oregon, USA: Association for Computing Machinery, 2007, pp. 473–480. ISBN: 9781595937933. DOI: 10.1145/1273496.1273556. URL: https://doi.org/10.1145/1273496.1273556.

[30] Huimin Li, Marina Krček, and Guilherme Perin. “A Comparison of Weight Initializers in Deep Learning-Based Side-Channel Analysis”. In: *Applied Cryptography and Network Security Workshops*. Ed. by Jianying Zhou et al. Cham: Springer International Publishing, 2020, pp. 126–143. ISBN: 978-3-030-61638-0.

[31] Min Lin, Qiang Chen, and Shuicheng Yan. “Network In Network”. In: *CoRR* abs/1312.4400 (2014).

[32] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Commun. ACM* 63.6 (May 2020), pp. 46–56. ISSN: 0001-0782. DOI: 10.1145/3357033. URL: https://doi.org/10.1145/3357033.

[33] Xiangjun Lu et al. “Pay Attention to Raw Traces: A Deep Learning Architecture for End-to-End Profiling Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, Issue 3 (2021), pp. 235–274. DOI: 10.46586/tches.v2021.i3.235-274. URL: https://tches.iacr.org/index.php/TCHES/article/view/8974.

[34] Stefan Mangard, Maria Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. English. 1st ed. XXIII, 337 S. Springer, 2007. ISBN: 0-387-30857-1.

[35] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[36] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. “Gradient Visualization for General Characterization in Profiling Attacks”. In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Ilia Polian and Marc Stöttinger. Cham: Springer International Publishing, 2019, pp. 145–167. ISBN: 978-3-030-16350-1.

[37] Loïc Masure and Rémi Strullu. *Side Channel Analysis against the ANSSI’s protected AES implementation on ARM*. Cryptology ePrint Archive, Report 2021/592. https://ia.cr/2021/592. 2021.

[38] Guilherme Perin, Lichao Wu, and Stjepan Picek. *Gambling for Success: The Lottery Ticket Hypothesis in Deep Learning-based SCA*. Cryptology ePrint Archive, Report 2021/197. https://ia.cr/2021/197. 2021.

[39] Stjepan Picek et al. “The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations”. In: *Transactions on Cryptographic Hardware and Embedded Systems* 2019 (Nov. 2018). DOI: 10.13154/tches.v2019.i1.209-237.

[40] Bastian Richter, Alexander Wild, and Amir Moradi. *Automated Probe Repositioning for On-Die EM Measurements*. Cryptology ePrint Archive, Report 2019/923. https://ia.cr/2019/923. 2019.

[41] Jorai Rijsdijk et al. “Reinforcement Learning for Hyperparameter Tuning in Deep Learning-based Side-channel Analysis”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, Issue 3 (2021), pp. 677–707. DOI: 10.46586/tches.v2021.i3.677-707. URL: https://tches.iacr.org/index.php/TCHES/article/view/8989.

[42] Sebastian Risi, Joel Lehman, and Kenneth O. Stanley. “Evolving the Placement and Density of Neurons in the Hyperneat Substrate”. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’10. Portland, Oregon, USA: Association for Computing Machinery, 2010, pp. 563–570. ISBN: 9781450300728. DOI: 10.1145/1830483.1830589. URL: https://doi.org/10.1145/1830483.1830589.

[43] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *ArXiv* abs/1505.04597 (2015).

[44] Florian Schroff, Dmitry Kalenichenko, and James Philbin. “FaceNet: A unified embedding for face recognition and clustering”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 815–823.

[45] Ramprasaath R. Selvaraju et al. “Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization”. In: *CoRR* abs/1610.02391 (2016). arXiv: 1610.02391. URL: http://arxiv.org/abs/1610.02391.

[46] P.Y. Simard, D. Steinkraus, and J.C. Platt. “Best practices for convolutional neural networks applied to visual document analysis”. In: *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* 2003, pp. 958–963. DOI: 10.1109/ICDAR.2003.1227801.

[47] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2015).

[48] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. “A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks”. In: *Advances in Cryptology - EUROCRYPT 2009*. Ed. by Antoine Joux. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 443–461. ISBN: 978-3-642-01001-9.

[49] Kenneth Stanley, David D’Ambrosio, and Jason Gauci. “A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks”. In: *Artificial life* 15 (Feb. 2009), pp. 185–212. DOI: 10.1162/artl.2009.15.2.15202.

[50] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evol. Comput.* 10.2 (June 2002), pp. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. URL: https://doi.org/10.1162/106365602320169811.

[51] Biaoshuai Tao and Hongjun Wu. “Improving the Biclique Cryptanalysis of AES”. In: *Information Security and Privacy*. Ed. by Ernest Foo and Douglas Stebila. Cham: Springer International Publishing, 2015, pp. 39–56. ISBN: 978-3-319-19962-7.

[52] Matus Telgarsky. “Benefits of Depth in Neural Networks”. In: *COLT*. 2016.

[53] Matus Telgarsky. “Representation Benefits of Deep Feedforward Networks”. In: *ArXiv* abs/1509.08101 (2015).

[54] Phillip Verbancsics and Josh Harguess. *Generative NeuroEvolution for Deep Learning*. 2013. DOI: 10.48550/ARXIV.1312.5355. URL: https://arxiv.org/abs/1312.5355.

[55] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning*. 1992, pp. 229–256.

[56] Lennert Wouters et al. “Revisiting a Methodology for Efficient CNN Architectures in Profiling Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, Issue 3 (2020), pp. 147–168. DOI: 10.13154/tches.v2020.i3.147-168. URL: https://tches.iacr.org/index.php/TCHES/article/view/8586.

[57] Lichao Wu and Guilherme Perin. “On the Importance of Pooling Layer Tuning for Profiling Side-Channel Analysis”. In: *Applied Cryptography and Network Security Workshops*. Ed. by Jianying Zhou et al. Cham: Springer International Publishing, 2021, pp. 114–132. ISBN: 978-3-030-81645-2.

[58] Lichao Wu, Guilherme Perin, and Stjepan Picek. *I Choose You: Automated Hyperparameter Tuning for Deep Learning-based Side-channel Analysis*. Cryptology ePrint Archive, Report 2020/1293. https://ia.cr/2020/1293. 2020.

[59] Gabriel Zaid et al. *Methodology for Efficient CNN Architectures in Profiling Attacks – Extended Version*. Cryptology ePrint Archive, Report 2019/803. https://ia.cr/2019/803. 2019.

[60] Gabriel Zaid et al. *Understanding Methodology for Efficient CNN Architectures in Profiling Attacks*. Cryptology ePrint Archive, Report 2020/757. https://ia.cr/2020/757. 2020.

Appendix A

Methodology comparison

Ranking in [41]	$ASCAD\ N = 50\ RS$ Value CNN Architecture	$\bar{N}_{T_{GE}}$ according to[41]	parameters
1	[C(4,50,1), P(50,4), C(2,50,1), P(2,2), C(2,50,1), P(50,50), GAP(1), FC(4), SM(256)]	313	2,100
2	[C(16,25,1), P(50,4), C(2,2,1), P(4,2), C(8,25,1), P(4,2), C(4,25,1), P(25,25), GAP(1), FC(2), SM(256)]	333	2,472
3	[C(4,50,1), P(50,4), C(2,50,1), P(2,2), C(2,50,1), P(50,50), FLAT(50), FC(15), FC(15), SM(256)]	372	5,189
4	[C(8,25,1), P(50,4), C(16,50,1), BN, P(50,7), C(8,3,1), BN, P(4,4), C(128,2,1), P(2,2), C(32,1,1), GAP(32), FC(15), FC(4), SM(256)]	335	15,207
5	[C(4,50,1), P(25,2), C(4,3,1), P(50,7), C(8,25,1), P(2,2), C(16,3,1), P(7,4), C(8,3,1), GAP(8), FC(4), SM(256)]	403	3,172
6	[C(8,2,1), P(50,4), C(2,50,1), P(2,2), C(2,50,1), P(50,50), FLAT(50), FC(20), FC(20), FC(2), SM(256)]	548	2,318
7	[C(2,3,1), P(50,4), C(64,3,1), P(2,2), C(8,50,1), P(25,7), FLAT(63), FC(2), FC(2), FC(2), SM(256)]	480	26,990
8	[C(4,50,1), P(50,4), C(2,50,1), P(2,2), C(2,50,1), P(50,50), FLAT(50), FC(2), SM(256)]	734	1,582
9	[C(8,25,1), P(50,2), C(16,2,1), P(50,25), C(2,1,1), P(2,2), C(32,3,1), P(2,2), C(8,2,1), GAP(8), FC(10), FC(10), FC(4), SM(256)]	705	2,782
10	[C(4,25,1), P(50,4), C(2,1,1), P(25,25), C(8,2,1), P(2,2), C(64,2,1), P(2,2), FLAT(2), FC(4), FC(2), FC(2), SM(256)]	704	2,286

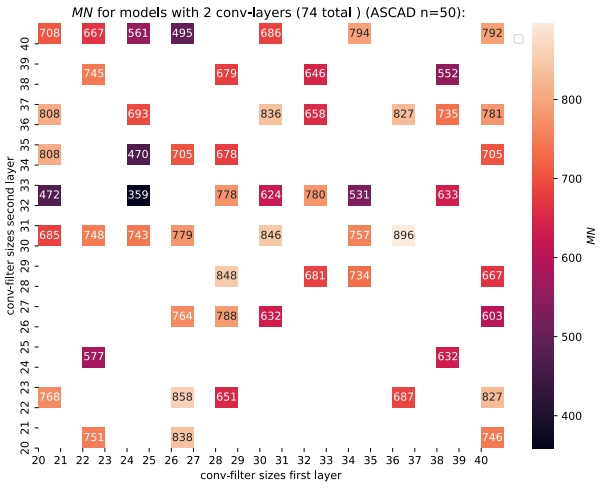
Table A.1: Jorai Rijsdijk's[41] results using the Reinforcement-learning Methodology from [41].

Ranking in [41]	$\bar{N}_{T_{GE}}$ over 10 trials	$\overline{GE}$ reached over 10 trials	MG	MN
1	[-,-,-,-,-,-,-,-,-,-]	[127.17, 114.95, 123.61, 131.95, 117.94, 121.89, 4.3, 107.03, 124.42, 118.79]	109.24	-
2	[-,-,-,-,-,923,-,282,244,-]	[90.46, 4.24, 0.0, 125.0, 117.77, 0.83, 16.23, 0.0, 0.0, 54.31]	40,884	483
3	[-,-,-,-,-,-,-,-,-,-]	[85.15, 101.3, 112.32, 37.52, 27.67, 68.04, 30.24, 26.98, 108.59, 102.94]	70.075	-
4	[219, 213, 353, 221, 210, 206, 219, 205, -,-]	[0.0,0.0,0.0,0.0,0.0, 0.0,0.0,0.0,25.99,27.08]	5.307	230,75
5	[441,-,-,726,177,-,330,373,341,-]	[0.02,35.36,8.83,0.44,0.0, 2.17,0.0,0.0,0.0,3.77]	5.059	398
6	[-,-,-,-,-,-,-,-,-,-]	[49.86,122.18,124.62,120.01,62.77, 95.29,115.52,108.21,28.97,120.24]	94,767	-
7	[744,-,-,-,-,-,-,-,-,-]	[0.36,118.92,126.33,124.86,125.47, 129.87,29.89,110.39,108.44,117.55]	99,208	-
8	[-,-,-,-,-,-,-,-,-,-]	[131.19,40.42,105.16,24.36,102.63, 119.78,125.5,111.15,21.09,41.22]	82.25	-
9	[-,-,-,-,-,-,-,-,-,-]	[25.6,75.01,27.53,24.61,12.21, 24.26,22.94,29.6,27.82,28.52]	29.81	-
10	[-,-,-,-,-,-,-,-,-,-]	[95.18,89.33,124.02,65.02,84.68, 118.36,73.91,124.64,102.02,95.5]	97,266	-

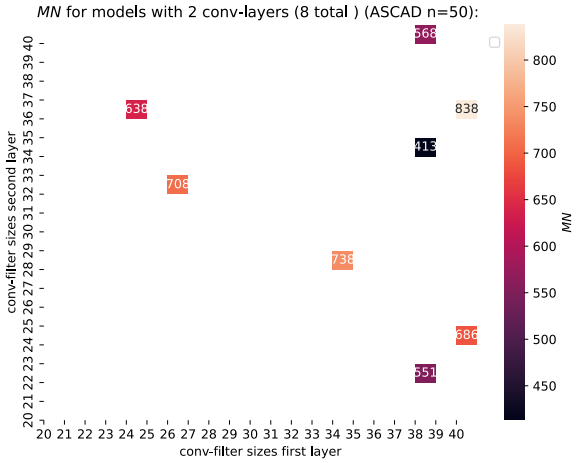
Table A.2: Reproducing Jorai Rijsdijk's[41] results, from table A.1, with the environment-setting from [41].

Appendix B

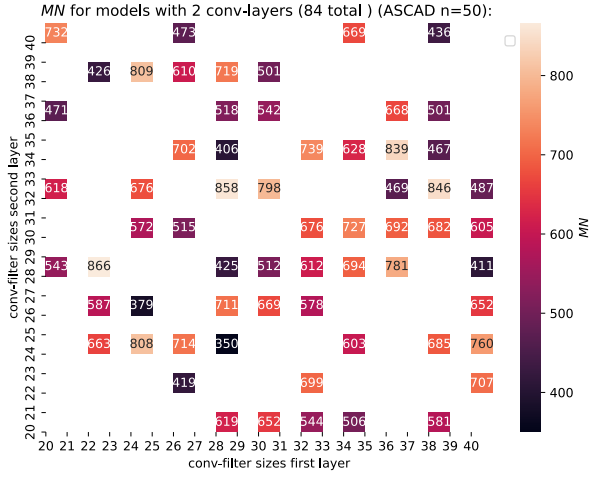
Increased filter sizes



(a) selu.

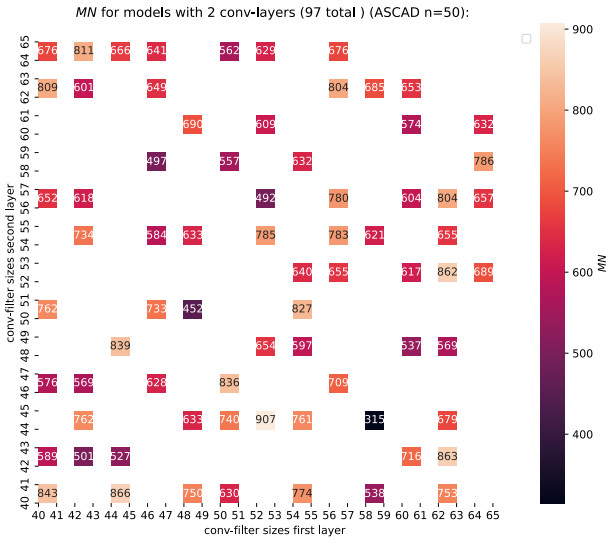


(b) relu.

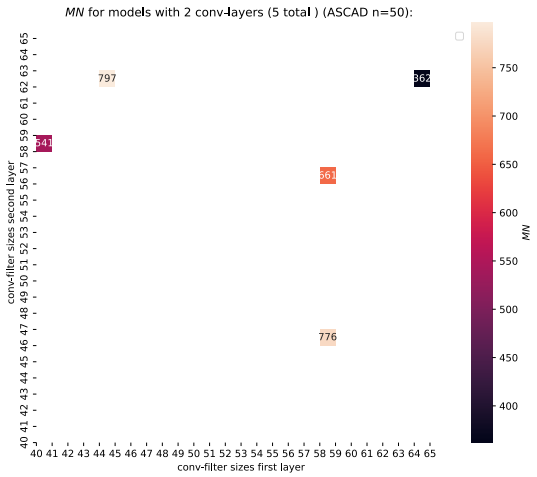


(c) elu.

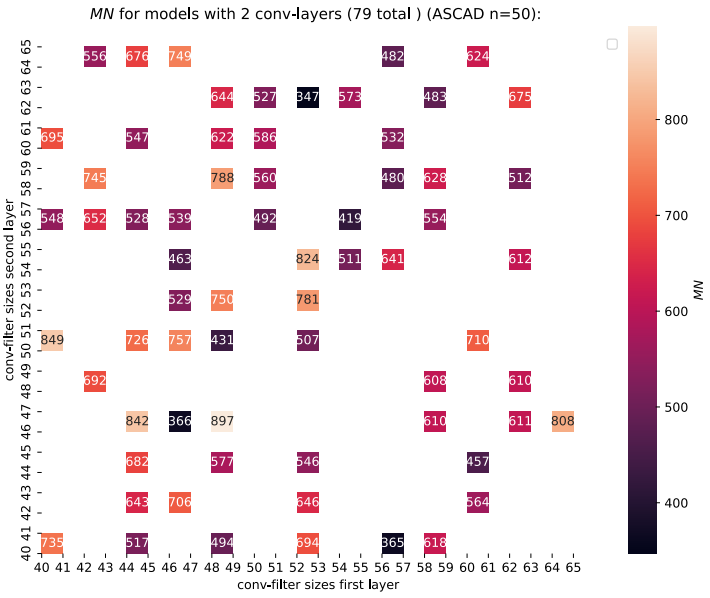
Figure B.1: Good candidates: Assessing the convolutional filter sizes: ranges $[20, 40]$.



(a) selu.

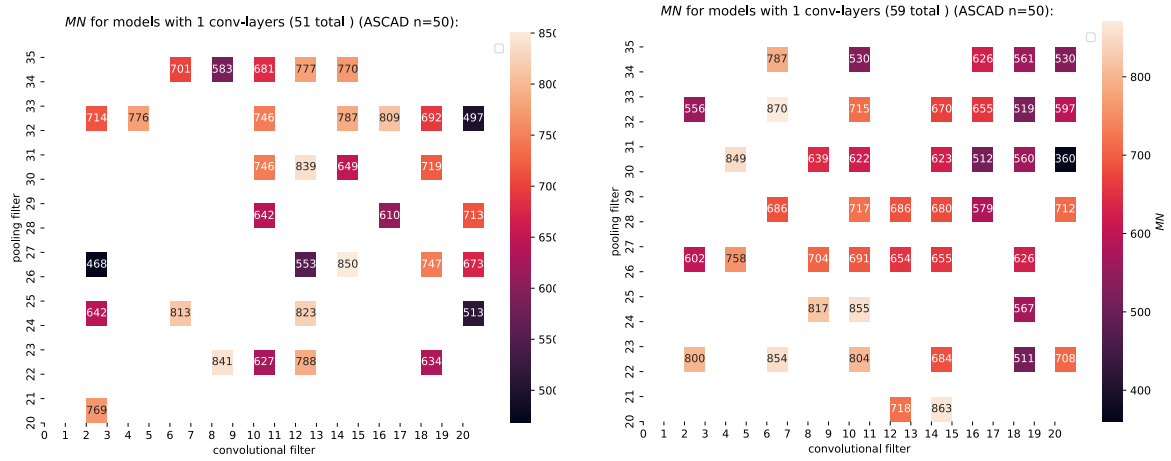


(b) relu.



(c) elu.

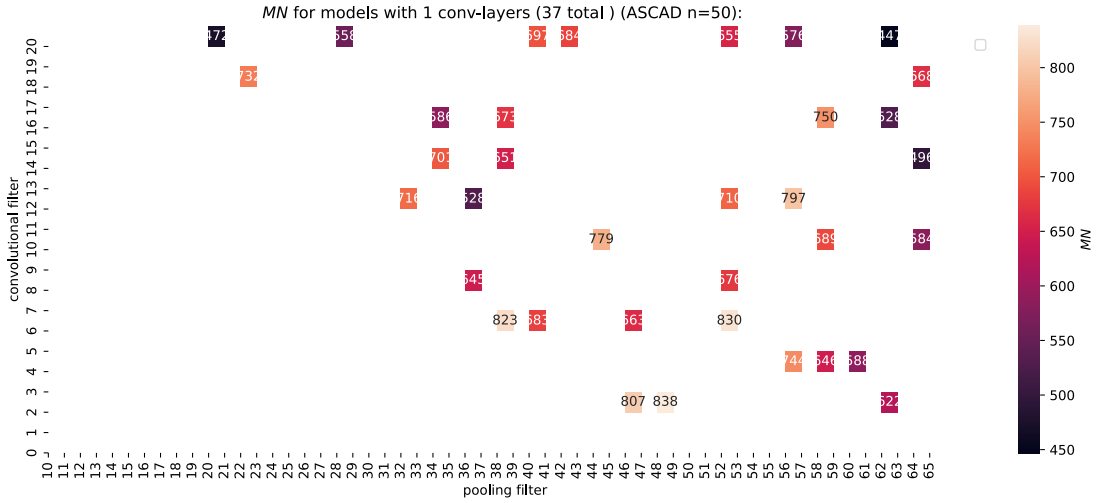
Figure B.2: Good candidates: Assessing the convolutional filter sizes: ranges $[40, 65]$.



(a) selu

(b) elu

Figure B.3: Good candidates with single layer: pooling-filter ranges $[20, 35]$.



(a) selu.

(b) relu.

(c) elu.

Figure B.4: Good candidates with single layer: pooling-filter ranges $[10, 65]$.

Appendix C

Model comparison

C.1 DPA v4

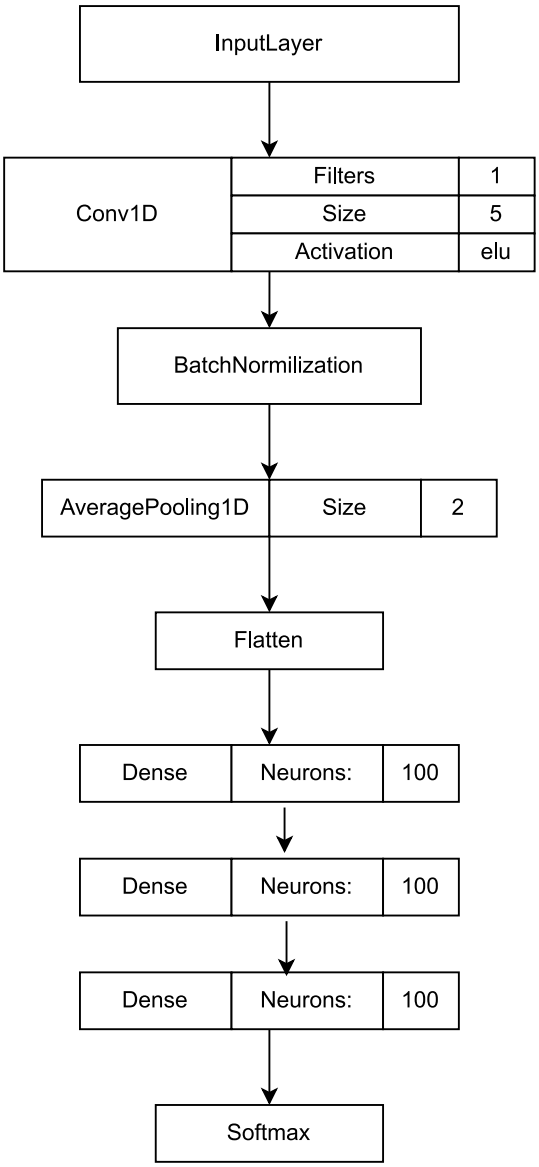


Figure C.1: $DPAv4$ dataset model (produced by our methodology), parameters: $246,163$, $N_{T_{GE}} = 1$.

C.2 AES RD

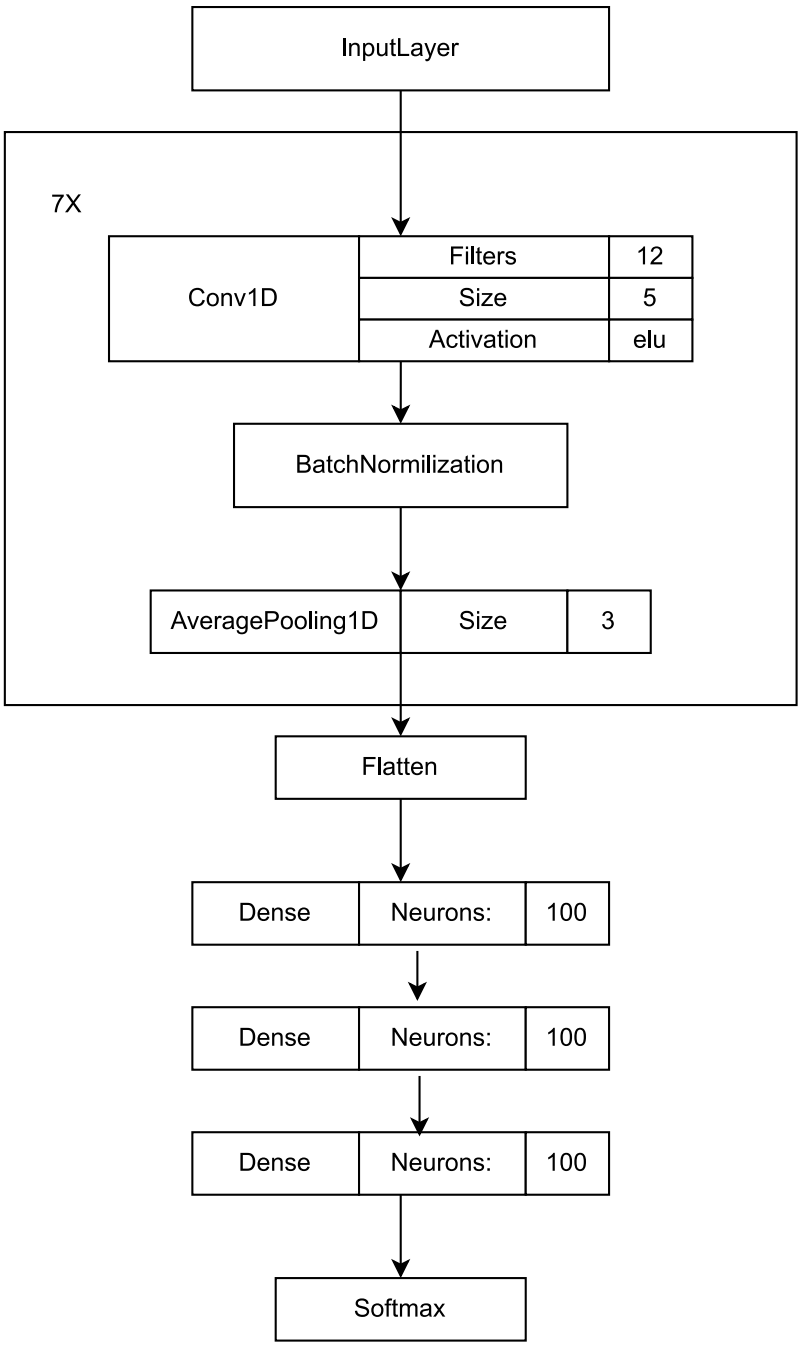


Figure C.2: $AES\ RD$ dataset model (produced by our methodology), parameters: $52,156$, $N_{T_{GE}} = 2$.

C.3 AES HD

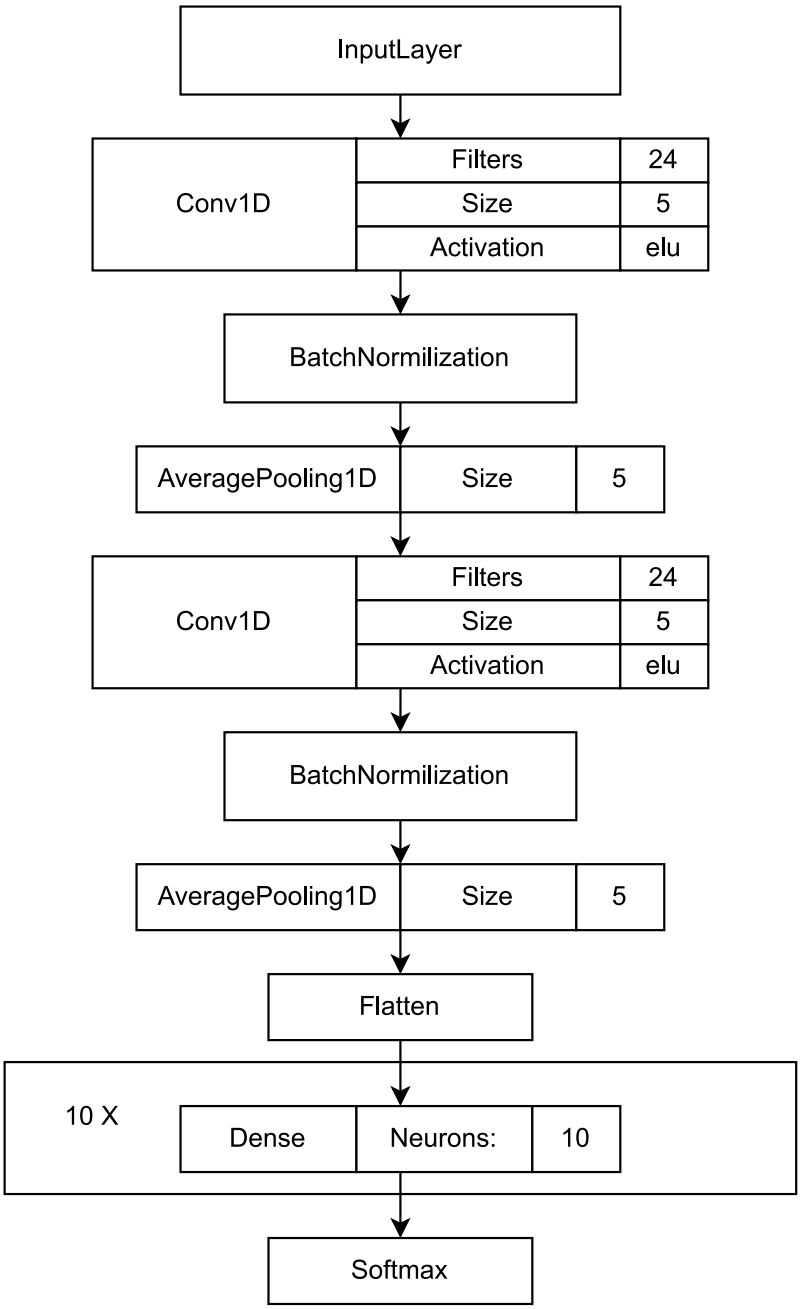


Figure C.3: $AES\ HD$ dataset model (produced by our methodology), parameters: $18,286$, $N_{T_{GE}} = 690$.